GIT'S GOLDEN RULES
(FOR TEAMS)

EMMA JANE HOGBIN WESTBY
@EMMAJANEHW
WWW.GITFORTEAMS.COM
WHO THINKS

GIT IS WEIRD AND HARD?
WHO THINKS

GIT IS EASY?
Who thinks only gits raise their hand when asked?
</CALIBRATION>
GIT IS GOOD...
AS A CONTENT TRACKER FOR TEXT FILES.

GIT IS GREAT
Git is very fast compared to centralised version control systems.
GIT IS NOT MAGIC.
A DEPENDENCY MANAGER.

GIT IS NOT
GIT DOES NOT HAVE ACCESS CONTROL.
WHOLE FILE SNAPSHOTS.
GIT BECOMES SLOWER AS YOUR HISTORY GETS VERY, VERY LARGE.
FOR VERY LARGE VALUES OF VERY LARGE

10,000+ COMMITS
GIT CAN GET UGLY...
IT'S NOT REALLY YOUR FAULT.
GIT IS WEIRD AND HARD BECAUSE
THE INTERNALS HAVE STRONG OPINIONS,
BUT THE INTERFACE DOES NOT.
WARNING!

CONTAINS OPINIONS
TODAY

GIT'S GOLDEN RULES

▸ Talk to your teammates.
▸ Separate your ideas.
▸ Be consistent.
▸ Include only what you need.
GOLDEN RULE #1

TALK TO YOUR TEAMMATES.
WITHOUT ACCESS CONTROLS, YOU MUST CODIFY GOVERNANCE
BASIC SETUP

committed work is then uploaded

push

update local copy of the repository

pull

Local Repository

commit

work is saved locally

choose a branch to begin working on

Local Repository

Centralized Code Hosting System
Forks roughly allow permissions without branch locking.
PULL REQUESTS

- Clone the repository
- Checkout new changes
- Commit changes
- Push changes to the repository
- Pull request via the web interface
TALK TO YOUR TEAMMATES.

- Map access, then map commands.
- Use branch locking or forks to control access.
GOLDEN RULE #2

SEPARATE YOUR IDEAS.
DOCUMENT AND USE A SINGLE BRANCHING STRATEGY
USE A CONVENTION (OR INVENT YOUR OWN … ENDING IN “FLOW”)

POPULAR BRANCHING CONVENTIONS

- State / Environment Branching (GitLab Flow)
- Branch-Per-Feature (GitHub Flow)
- Scheduled Release (GitFlow)
BRANCH-PER-FEATURE

GitHub Flow

master

integration

feature 1

feature 2

if no errors: merge to master

deploy

if there are errors: re-deploy master

deploy

merge to master

fix the errors then re-try deployment
SCHEDULED RELEASE

branch ->
merge

custom branches; usually ticket-focused

master
hotfix-1237-reins
release-1.0
develop
1234-fixing_links 1235-horse 1236-cart

v1.0.1
v1.0

initial release
release bug fixed
merge bug fix back into develop

GITFLOW
ONE BALL PER IDEA.
$ git rebase --interactive HEAD~n

Commit to whole ideas prior to merge
$ git merge --squash pr_branch

Convert conversations to conclusions at merge
SEPARATE YOUR IDEAS.

- Every commit and each branch should hold a coherent unit of work.
GOLDEN RULE #3

BE CONSISTENT.
DOCUMENT AND USE A MAINTENANCE STRATEGY.
# WHY THE FUSS?

## BECAUSE TIMTOWTDI

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pull</td>
<td>fetch + merge</td>
</tr>
<tr>
<td>pull --rebase=preserve</td>
<td>fetch + rebase</td>
</tr>
<tr>
<td>merge --no-ff</td>
<td>forces a merge commit object (&quot;true merge&quot;)</td>
</tr>
<tr>
<td>merge --ff-only</td>
<td>fast forward (graph looks like rebase)</td>
</tr>
<tr>
<td>merge --squash</td>
<td>compress commits to one; then merge</td>
</tr>
<tr>
<td>rebase</td>
<td>forward-port local commits</td>
</tr>
<tr>
<td>cherry-pick</td>
<td>merge individual commits</td>
</tr>
</tbody>
</table>
I want to combine work from two or more branches, should I merge or rebase?

Is this new work which you created specifically for your project?

Yes

Does your project have long-standing integration branches?

Yes

Are you attempting to update a long-standing branch with pre-approved work from a remote repository?

Yes

Rebase
It sounds like you need to bring your local copy of an integration branch up-to-date. It is appropriate to update the branch using a rebasing strategy by using `pull --rebase=preserve`.

No

Are you attempting to update a ticket branch from an long-standing branch in your project?

Yes

Rebase
It sounds like you need to bring your ticket branch up-to-date before sharing your work. It is appropriate to rebase.

No

Are you attempting to incorporate work from an upstream project?

Yes

It sounds like you might have just finished a peer code review. You should merge the approved work into the master branch.

No

If it breaks things will you remove the upstream work, or adjust your own code?

Yes

Rebase
It sounds like you're committed to having this work in place. You can safely rebase to add the new work from the upstream project.

No

Merge
It sounds like you might need to be able to easily undo this change. You should merge the updates, so that you can remove them easily if needed.

No

Are you attempting to incorporate work from an upstream project?

Yes

Merge
It sounds like you're trying to incorporate a feature branch into master. You should merge your work.

No

Are you attempting to update a long-standing branch with pre-approved work from a remote repository?

Yes

Rebase
It sounds like you need to bring your local copy of an integration branch up-to-date. It is appropriate to update the branch using a rebasing strategy by using `pull --rebase=preserve`.

No

Are you attempting to update a ticket branch from an long-standing branch in your project?

Yes

Rebase
It sounds like you need to bring your ticket branch up-to-date before sharing your work. It is appropriate to rebase.

No

If it breaks things will you remove the upstream work, or adjust your own code?

Yes

Rebase
It sounds like you're committed to having this work in place. You can safely rebase to add the new work from the upstream project.

No

Merge
It sounds like you might need to be able to easily undo this change. You should merge the updates, so that you can remove them easily if needed.
MERGE TO UPDATE IS EASY, BUT MESSY

before merge with no fast forward

updating a feature branch with merge, no fast forward

closing a feature branch with merge, no fast forward
UPDATE WITH REBASE

(IF YOU CARE)
BE CONSISTENT.

- Keep your history legible by having the team use a single strategy to update branches.
GOLDEN RULE #4

INCLUDE ONLY WHAT YOU NEED.
OUTSOURCE YOUR
DEPENDENCY MANAGEMENT
IF YOU MUST

INCLUDE EXTERNAL WORK

- Keep your "core" clean and track upstream work with named branches.

- Nest repositories without tracking by using subtrees (clone inside a clone).

- Git can track external repositories with submodules. There be dragons.
STORE AS MUCH AS YOU NEED, BUT NOT MORE.
ONE MEGA REPO

MONOLITH
MICROSERVICES

MANY ICKLE REPOS
BINARY FILES WILL GROW WITH EACH VERSION
USE OFF-SITE STORAGE

FOR VERY LARGE FILES
USE SHALLOW CLONES FOR FASTER DEPLOYMENTS

$ GIT CLONE --DEPTH [DEPTH] [REMOTE-URL]

$ GIT CLONE [URL] --BRANCH [BRANCH_NAME] --SINGLE-BRANCH [FOLDER]
GOLDEN RULE

INCLUDE ONLY WHAT YOU NEED.

- Outsource your dependency management.
- Break your repository into smaller service repositories when it's time.
- Binary files grow when versioned.
- Use shallow clones for faster deployments.
GIT’S GOLDEN RULES

- Talk to your teammates.
- Separate your ideas.
- Be consistent.
- Include only what you need.
RESEARCHERS

BIG REPOSITORIES

- How to Handle Big Repositories with Git
  https://www.atlassian.com/git/articles/how-to-handle-big-repositories-with-git/

- How do you handle your microservices
  https://news.ycombinator.com/item?id=9705098

- Organizing Microservices in a Single Repository
RESOURCES

DEPENDENCY MANAGEMENT

- How do you handle external dependencies?

- Paket for .NET and Mono
  http://fsprojects.github.io/Paket/

- Composer for PHP
  https://getcomposer.org/doc/00-intro.md

- Mastering Submodules
  https://medium.com/@porteneuve/mastering-git-submodules-34c65e940407
Do not version binaries in the repository; reference them from another location.

- git-annex - https://git-annex.branchable.com/
- git-bigfiles - http://caca.zoy.org/wiki/git-bigfiles
- GLFS - https://git-lfs.github.com ** start here
- http://blogs.atlassian.com/2014/05/handle-big-repositories-git/