Section 15: Rebasing: The Scariest Git Command?

139. What Really Matters In This Section

140. Why is Rebasing Scary? Is it?

* There are two main ways to

use the git rebase command:

* as an alternative to merging
* as a cleanup tool

141. Comparing Merging & Rebasing

* Rebasing helps to create a cleaner project history.
* Rebase reapplies commits on top of another base branch.
* Merge joins two or more development histories together.
* Key difference between rebase and merge is that while merge preserves history as it happened, rebase rewrites it.

142. Rebase Demo Pt 1: Setup & Merging

* Using the switch and merge commands to merge one branch onto another and displaying the log to show the commit history is clustered and un-organized.

143. Rebasing Demo Pt 2: Actually Rebasing

* "git rebase <branch-name>" command will rebase your current working branch to the entered branch.
* In this type it will create a clean linear structure.
* It will avoid any merge commits done.
* Commit hash changes and rewrites the history.

144. The Golden Rule: When NOT to Rebase

* Never rebase commits that have been shared with others. If you have already pushed commits up to Github...DO NOT rebase them unless you are positive no one on the team is using those commits.

145. Handling Conflicts & Rebasing

* Follow the on screen instruction.
* If any conflicts occur, try manually fixing the error and instead of commiting, conitnue rebasing using "git rebase --continue" command.

Section 16: Cleaning Up History With Interactive Rebase

146. What Really Matters In This Section

147. Introducing Interactive Rebase

Running git rebase with the -i option will enter the interactive mode, which allows us to edit commits, add files, drop commits, etc. Note that we need to specify how far back we want to rewrite commits.

148. Rewording Commits With Interactive Rebase

* To reword the commits use "git rebase -i <commit-head>" command.
* It will open a interactive window in text editor.
* Commonly used commands in interactive rebase

|  |  |
| --- | --- |
| pick | use the commit |
| reword | use the commit, but edit the commit message |
| edit | use commit, but stop for amending |
| fixup | use commit contents but meld it into previous  commit and discard the commit message |
| drop | remove commit |

149. Fixing Up & Squashing Commits With Interactive Rebase

* To fixup and squash a commit onto previous commit use interactive rebase command and add fixup instead of pick.
* If we want to sqush multiple commits, we can use multiple FIXUP .

150. Dropping Commits With Interactive Rebase

* To remove a commit with its data use "drop" in interactive rebase.

Section 17: Git Tags: Marking Important Moments In History

151. What Really Matters In This Section

152. The Idea Behind Git Tags

* Tags are ref's that point to specific points in Git history.
* There are two types of Git tags we can use: lightweight and annotated tags
* lightweight tags are just a name/label that points to a particular commit.
* annotated tags store extra meta data including the

author's name and email, the date, and a tagging message

153. A Side Note On Semantic Versioning

* Semantic versioning : The semantic versioning spec outlines a standardized versioning system for software releases. It provides a consistent way for developers to give meaning to their software releases.
* Example : 2.4.1 - 2 major release,4 minor release,1 patch release
* Semantic versioning has set of rules.

154. Viewing & Searching Tags

* "git tag" command will print a list of all the tags in the current repo.
* "git tag -l "<pattern>" " command will print all the tags having that particular pattern.

155. Comparing Tags With Git Diff

* Use git diff command to compare between two tags.

156. Creating Lightweight Tags

* "git tag <tagname>" command will add a tagname to current head.

157. Creating Annotated Tags

* "git tag -a <tagname>" command will add a tagname to current head.
* We can use -m option to add inline comment.
* "git show <tag>" command will show the meta data of the tag.

158. Tagging Previous Commits

* "git tag <tagname> <commit-hash>" command will add tag name to a particular commit.

159. Replacing Tags With Force

* "git tag <tagname> <commit-hash> -f" command will add tag name to a particular commit with force. Is useful to rename tags.

160. Deleting Tags

* "git tag -d <tagname>" command will delete a tag.

161. IMPORTANT: Pushing Tags

* "git push --tags" command will push all the tags at once to a remote repo.
* "git push <tagname>" command will push particular tags to a remote repo.

Section 18: Git Behind The Scenes - Hashing & Objects

162. What Really Matters In This Section

163. Working With The Local Config File

* The config file is for configuration. We've seen how to configure global settings like our

name and email across all Git repos, but we can also configure things on a per-repo basis.

* We can change colors in config file

164. Inside Git: The Refs Directory

* Inside refs directory there multiple files (heads,remotes,tags)
* Heads store commit hash of the head in each branch.
* Tags are just pointers which points to a particular commit hash and is stored in refs.
* Remotes store details of the user.

165. Inside Git: The HEAD file

* HEAD s just a text file that keeps track of where HEAD points.
* If it contains refs/heads/master, this means that HEAD is pointing to the master branch.

166. Inside Git: The Objects Directory

* The objects directory contains all the repo files. This is where Git stores the backups of

files, the commits in a repo, and more.

* 4 types of Git objects : commit, tree, blob, annotated tag

167. A Crash Course On Hashing Functions

* Cryptographic hash functions:
  + One-way function which is infeasible to invert
  + Small change in input yields large change in the output
  + Deterministic - same input yields same output
  + Unlikely to find 2 outputs with same value
* Uses SHA-1 hashing function to encrypt.