Lab: Rebasing

An important aspect of Git is how clean you keep your repository and history. A clean repository is easier to work with and understand what's happened.

Lab Overview

This scenario will cover how you can re-write your Git history using Rebase to restructure your commits to ensure they're understandable before you push your changes.

Recommendation

You should only rebase commits that have not been shared with other people via push. Rebasing commits causes their commit-ids to change which can result in losing future commits.

Pre-reqs:

- Google Chrome (Recommended)

Lab Environment

There is no requirement for any setup.

Run the following command in **terminal**: `cd ~/Desktop/gitlab-intro/lab5/ && mv git .git`

Step 1 - Amending Commit Messages

Re-writing the repositories history is done using git **rebase** -interactive. By putting rebase into interactive mode you have more control over the changes you want to make. After launching into interactive mode you are given six commands to perform on each commit in the repository. By using the editor which opens, by default Vim, you define which actions you want to perform on each commit.

In this lab, we want to change the commit. To put it into this state we need to change the word "pick" next to the commit to match the action you want to perform based on the list shown in the Vim window, in this case "reword".

In this lab, we want to change the commit message.

Start

To begin we need to enter Interactive Rebase mode using `git rebase --interactive --root`

Select Interactive Mode

To begin with Vim can be a little confusing, to edit text you need to first type `i` which will put you into "insert mode".

We want to edit the "comit" typo in the first commit message "Initial comit of the list". For the commit change the word "pick" to match the command we want to perform on the commit, in this case "reword".

To save and exit press `esc` key then `:wq`. This will open another Vim editor window.

Changing Message

Again using Vim, edit the commit message to change "comit" to "commit". After saving and exiting Vim you will see the output of Git changing the commit. Use `git log --oneline` to see the updated commit message.

Protip

The --root argument allows you to rebase all commits in the repository, including the first commit.

A faster alternative to change the last commit message is using `git commit --amend` and make the change using Vim.

Step 2 - Squash Commits

A series of 8 different commits has been made in your local environment. At the time these commits made sense however now they need to be just a single commit. Using Rebase we need to squash the commits together.

Using `git rebase --interactive HEAD~8` , we have the range of commits from HEAD to the last 0. To squash we need a base commit which everything will be squashed into. As such, leave the first commit as "pick" but change the rest to "squash".

After you save you will have the chance to edit. By default, the Git commit message will be a combination of the previously squashed commit messages.

Task

When we enter Interactive Rebase we can specify that we want to modify the previous 8 commits using `git rebase --interactive HEAD~8`

In the previous stage we used reword. Here we want to use squash. We want to squash 8 commits into one, if we labelled all the commits as squash then we'd get the error "Cannot 'squash' without a previous commit" as there isn't a base commit to squash everything into.

To squash the commits we need to leave the first commit as our base, and label the following 7 with squash.

Commit Message

As saving and quitting Vim we are shown a new Vim window which lists a combination of the 8 commit messages in the rebase.

After saving the commit message the history will be modfied. You can see this using `git log --oneline`

Step 3 - Re-order Commits

Re-ordering commits can help build a better picture of the logical order that worked was completed in.

Task

We want to re-order our last two commits. Using HEAD~2 allows us to modify them.

```
git rebase --interactive HEAD~2
```

Using Vim, simply reorder the lines, save & quit, and the commits will match the order.

Step 4 - Split Commit

Just like with squashing commits, sometimes it's useful to split changes out of commits in order to keep them focused and make cherry picking or reverting easier.

Splitting commits is a two staged process. First we need to define which commit we want to split and then we need to define how we want the new commits to look.

Defining Commit To Split

Here we want to split the previous commit. We enter rebase mode using

`git rebase --interactive HEAD~1`

Like with previous rebasing, we need to change the task to edit

We are now in a state of interactively edit the history. Git will record all the changes and the end result will be applied to the repository.

Splitting Commits

After defining we want to edit the commit we are now in a state that allows us to change the history.

As we want to split an existing commit we first we need to remove it using `git reset HEAD^` .

The commit has been removed but the files still exist. We can now perform the commits as we previously desire, as two separate actions.

Execute the commands:

```
git add file3.txt
git commit -m "File 3"
git add file4.txt
git commit -m "File 4"
```

Saving

Once happy with the state of the repository, we tell Git to continue the rebase and update the repository with --continue.

`git rebase --continue`

You can see the output and the two new commits using `git log --oneline`