



## Aprendendo a trabalhar com git e github em equipe.

Nesse tutorial, você verá comandos básicos do terminal, de como iniciar o git em seu projeto e comandos essenciais para um trabalho em equipe.

### 0.1 -> Comandos básicos

Nessa sessão veremos alguns comandos que facilitaram o uso do terminal, para identificar a pasta que esta, ver os conteúdos delas entre outros...

**Mostra a pasta que está no momento.**

```
$ pwd
```

**Cria um arquivo para o editor de texto a partir do terminal.**

```
$ touch exemplo.txt
```

**Mostra o que tem dentro da pasta.**

```
$ ls
```

**Para saber quem criou a pasta e os arquivos, quando criou, e a quantidade de arquivos dentro dele.**

```
$ ls -l
```

**Para mostrar se tem pasta oculta dentro dessa pasta que está.**

```
$ ls -a
```

Para mostrar o que tem dentro do arquivo, porém vai mostrar dentro do terminal, ele não vai abrir o vscode, exemplo de uso para abrir algum código:  
\$ cat doc/lib/main.dart.

```
$ cat nome_arquivo
```

---

## 0.2 -> Git help

Nessa sessão veremos que o git nos mostra vários comandos que podem ser utilizados

**Mostra todos os comandos que podem ser usados.**

```
$ git help
```

**Mostra os comandos mais avançados.**

```
$ git help -a
```

---

## 0.3 -> Dizendo ao Git quem é você

Esses comandos fazem seu nome e e-mail aparecerem nos commits feitos por você

**Abra o terminal e digite:**

```
$ git config --global user.name "SeuNome"  
$ git config --global user.email "SeuEmail"
```

---

## 0.4 -> Iniciando o git em um projeto

Esse comando vai iniciar o git em seu projeto criando um repositório para ele

**Entre na pasta do projeto pelo terminal ou abra o projeto no editor de texto (e abra um terminal nesse editor) e digite:**

```
$ git init
```

---

## **0.5 -> Adicionando arquivos para a área de staging para serem monitorados**

Começar a monitorar arquivos e/ou adicionar arquivos à área de staging “preparar para o commit”<sup>4</sup>

**Quando utilizamos dessa forma, ele vai incluir tudo que estiver no projeto (menos o que estiver na pasta gitignore), então sempre que usar git add “.” tenha cuidado pois você pode incluir arquivos sigilosos e sensíveis (use quando tiver certeza do que estiver fazendo e adicione pastas com conteúdo sensíveis e sigilosos no gitignore).**

```
$ git add .
```

**Podemos utilizar o git add incluindo arquivo por arquivo e pastas também, e a forma mais segura, porém, um pouco mais trabalhosa ou pode ser mais ágil quando tiver poucos arquivos.**

```
$ git add minha-pasta
```

```
$ git add arquivo.dart
```

---

## **0.6 -> Salvando alterações inclusas no git add, comitando o código, informando a os dev's parceiro ou a você mesmo, o que realizou nesse commit**

Os commits contam a história do projeto, então sempre seja específico ao fazer um 'commit' novo

**Depois de dar um git add, salve a alteração digitando um git commit:**

```
$ git commit -m "Mensagem explicando o que foi feito"
```

---

## 0.7 -> Checando o status do repositório

Exibe a branch atual, arquivos monitorados e não monitorados, e arquivos na área de staging e fora dela.

**No terminal digite:**

```
$ git status
```

**Logo em seguida será exibido informações como essas:**

```
$ git status
Exemplo de resposta:

On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    seu_arquivo

nothing added to commit but untracked files present (use "git add" to track)
```

---

## 0.8 -> Visualizando as modificações feitas em arquivos

Ao commitar um código, podemos usar um git diff para ver o que foi feito de diferente do último commit

**Para ver a diferença entre o ultimo commit e o commit atual.**

```
$ git diff
# outra opção
$ git diff seu_arquivo
```

**Permite visualizar as alterações e informações realizada em um commit específico.**

```
$ git show hashCommit
```

**Com o git diff staged podemos visualizar arquivos alterados em que enviamos para a staging com o comando git add (imagine que colocamos git add porém nos arrependemos de não ter olhado as alterações e com esse comando eu consigo verificar os arquivos da staging que não foram comitados ainda.**

```
$ git diff -staged
```

---

## 0.9 -> Visualizar histórico do projeto, e acessar commits

Nessa sessão podemos ver como acessamos o histórico do nosso projeto, ou seja, nós vamos acessar os commits feitos desde o primeiro até o ultimo.

**Para visualizar todos os commits feitos no projeto.**

```
# Listar todos os commits feitos
$ git log
```

**Para mostra os commits de forma resumida em apenas uma linha, aparecendo os 7 primeiros caracteres da hash e a mensagem do commit.**

```
# Lista os commits de forma reduzida, melhor para uma rapida visualização
$ git log --oneline
```

Para ver apenas o ultimo commit feito, podemos alterar o `-1` para o número que queremos, exemplo `-5`, isso iria mostrar os últimos 5 commits feito.

```
# Para ver o ultimo commit feito
$ git log -1
```

---

## 10.0 -> Utilizando .gitignore, para poder ignorar arquivos (por exemplo pastas de ambientes virtuais, arquivos com informações confidenciais, etc.)

Crie um arquivo chamado .gitignore na raiz do projeto (é o mesmo lugar onde você digitou git init) e dentro desse arquivo escreva o nome dos arquivos ou pastas que devem ser ignorados.

### Recomendação: Utilize templates prontos!

Se estiver usando o VSCode, baixe esta extensão: '*gitignore templates*'. Após instalar pressione **CTRL + P**, coloque o sinal '>' e digite '**Generate gitignore**', com isso ele vai dizer para você colocar a linguagem que está utilizando.

Caso contrário, utilize este site:

<https://www.toptal.com/developers/gitignore>, você precisa apenas fornecer a linguagem que está utilizando.

---

## 11.0 -> Armazenando as credenciais do github no terminal para evitar ficar colocando login e senha de uma maneira super segura!

Nessa sessão veremos como deixar salva nossas credencias do github em nosso terminal, para que ao fazer um git push o git não fique solicitando login e senha pelo terminal toda hora!

**Não utilize esse comando para gravar senhas**, pois ele cria um arquivo chamado '*.git-credentials*' e nesse arquivo tem sua senha do github sem segurança nenhuma.

Obs.: no mac temos a opção do 'osxkeychain' isso vai criptografar a sua senha

e estará seguro.

No Windows também tem a opção de 'osxkeychain'

```
$ git config --global credential.helper store
```

### Melhores Opções 'cache' e 'cache timeout':

Armazenar as configurações de login se senha temporariamente na memória, é mais seguro (por padrão é 15 minutos)

```
$ git config --global credential.helper cache
```

Para prolongar o tempo padrão de 15 minutos para o tempo que quisermos (exemplo: 3600 => segundos)

```
$ git config --global credential.helper 'cache --timeout=TEMPO_EM_SEGUNDOS'
```

---

## 12.0 -> Criando um repositório remoto no github

Dado que o repositório no GitHub ainda não existe, o primeiro passo é criá-lo

Acesse sua conta do GitHub e escolha a opção "New Repository".

Nomeie seu repositório, adicione uma descrição e selecione se ele será público (qualquer pessoa do mundo pode visualizar seu conteúdo) ou privado (apenas quem você autorizar poderá visualizar o conteúdo).

Não marque a opção "Initialize this repository with a README".

Clique em Create repository.

Siga as instruções abaixo para você poder enviar o seu projeto para esse repositório criado no github, essas instruções você coloca no terminal que este seu projeto.

```
$ git remote add origin https://github.com/SEU_NOME_DE_USUARIO/NOME_DO_REPOSITORIO.git  
$ git push -u origin master
```

---

## 13.0 -> Clonando um repositório do github

Nessa sessão você vai aprender a clonar um repositório do github

No terminal de seu pc, vá até a pasta que deseja e execute o comando exemplo abaixo:

Isso vai fazer com que o projeto que está no repositório remoto no github seja clonado para o seu pc

```
$ git clone https://github.com/USUARIO/NomeRepositorio.git
```

Temos a opção também de colocar um nome para a pasta em que o projeto clone ficará

```
$ git clone https://github.com/USUARIO/NomeRepositorio.git nome_pasta
```

Nomenclatura 'origin': É o nome(apelido) de um dos nossos repositórios remotos e quando eu aciono esse 'origin' ele aponta pra o repositório remoto: origin <https://github.com/CadisRaziel/git-aula-4.git> (fetch), ou ao invés de escrever esse endereço todo sempre que eu precisar fazer alguma operação com o repositório remoto, é mais fácil apelida-lo, por isso o 'origin'.

Obs.: quando a gente clona, o remote 'origin' já vem de brinde.

---

## 14.0 -> Listando repositórios remotos que tem no projeto

Nessa sessão você vai ver quais repositórios remotos estão incluídos em seu projeto

Ao executar o git remote ele vai listar o repositório que você está no github

```
$ git remote -v
```

Exemplo de como é apresentado um git remote

```
→ django-rest-framework git:(master) git remote -v
origin https://github.com/fabioruicci/django-rest-framework.git (fetch)
origin https://github.com/fabioruicci/django-rest-framework.git (push)
```



Exemplo de como é apresentado um git remote com um projeto fork por exemplo e como adiciona-lo (veremos sobre fork no final deste tutorial e fará mais sentido esse comando e esse apelido upstream)

```
→ django-rest-framework git:(master) git remote add upstream https://github.com/encode/django-rest-framework.git
→ django-rest-framework git:(master) git remote -v
origin  https://github.com/fabioruicci/django-rest-framework.git (fetch)
origin  https://github.com/fabioruicci/django-rest-framework.git (push)
upstream https://github.com/encode/django-rest-framework.git (fetch)
upstream https://github.com/encode/django-rest-framework.git (push)
```

---

## 15.0 -> Enviando alterações locais para o repositório remoto no github(PUSH)

Nessa sessão você vai ver como enviar seu projeto 'LOCAL' para o repositório REMOTO no github. Não se esqueça de já ter criado esse repositório e ter feito as instruções acima '11.0 -> Criando um repositório remoto no github

remoto → o que está no github

local → o que está apenas na sua maquina

push → enviar do local para o remoto

pull → enviar do remoto para o local

Git push comando utilizado para enviar as alterações do repositório/branch LOCAL para o repositório REMOTO que já esteja configurado no projeto (resumindo: fiz alterações no meu repositório LOCAL 'no meu pc' essas alterações só estão salvas no meu pc, com o git push nós conseguimos mandar essas alterações para o repositório remoto no github).

Lembrando que ele vai subir para o repositório remoto a branch que você está!!

Obs. lembre-se de já ter criado esse repositório no github ou clonado algum repositório!!!

```
$ git push
```

Apelidando o push, podemos apelidar o git push de acordo com o repositório que temos, com isso não precisamos passar a url completa desse repositório exemplo: Git origin nosso projeto ou git upstream projeto privado que não temos acesso de alteração por isso nesse caso precisamos fazer um 'fork' e nomear como upstream (exemplo de fork estará no fim desse tutorial)

```
$ git push origin
```

```
$ git push upstream
```

Esse comando envia o push da branch selecionada para o repositório

```
$ git push REMOTE_NAME BRANCH_NAME
```

Para ser mais específico ainda podemos colocar a branch selecionada que será enviada para a branch selecionada que irá recebê-la!!

```
$ git push REMOTE_NAME BRANCH_NAME:BRANCH_REMOTE
```

---

## 16.0 -> Baixando alterações remota para o repositório local em nosso pc(PULL)

Nessa sessão você vai ver como pegar as alterações do repositório remoto e baixarmos em nosso projeto no nosso pc local

remoto → o que está no github

local → o que está apenas na sua máquina

push → enviar do local para o remoto

pull → enviar do remoto para o local

O git pull é uma junção de dois comandos o 'fetch' e o 'merge':

Com esse comando eu baixo as mudanças encontradas no repositório remoto que não estão no repositório local. (posso passar também 'git fetch origin'), porém em seguida preciso fazer o merge abaixo

```
$ git fetch
```

Depois do fetch precisamos dar um merge com o 'origin/main'

```
$ git merge origin/main
```

Com isso esse git pull abaixo é o mesmo que fazer os dois comandos acima 'fetch' e 'merge'!!

Utilizado para baixar as alterações do repositório remoto para o repositório local (ele também faz um 'fast-forward')

```
$ git pull origin/main
```

### Erros ao tentar fazer um git push

Não é possível fazer um git push se os REPOSITÓRIOS não estão sincronizados. É necessário fazer um git pull primeiro, resolvendo eventuais conflitos.

Pequena história para servir como exemplo de um erro no git push

Imagine que tem você e o fulano trabalhando no mesmo projeto... de repente você começou a trabalhar na branch X e ele na branch Y (seguimos com branches diferentes, com commits diferentes no projeto). Só que aí eu fui lá e peguei minhas alterações locais na branch X e dei um git push, agora no repositório remoto as minhas alterações estão lá, somente as minhas!

Se nesse momento o fulano der um git push com o branch Y, ele vai tomar um erro, pois ele precisa baixar minhas alterações com o git pull aí sim depois ele consegue dá um push. (não se preocupe que quando você der um git pull você não vai perder seus dados que já criou, o que pode acontecer é ter conflito aí você o resolve) 'git pull faz um merge. lembre-se'

Baixa as alterações do remote que eu to deixando explicito e faz o merge já automático

```
$ git pull REMOTE_NAME
```

Baixa as alterações da branch que tá no remote que eu deixei explícito!

```
$ git pull REMOTE_NAME BRANCH_NAME
```

Baixa as alterações da branch remota, do origin(remote) e com isso faz o merge com a branch local automaticamente.

```
$ git pull REMOTE_NAME BRANCH_NAME:REMOTE_BRANCH
```

---

## 17.0 -> Criando, recuperando e deletando Branch

Nessa sessão você vai aprender a como criar branches e deletar branches

Listar branches

```
$ git branch
```

Criando branch

```
$ git branch NOME_DA_BRANCH
```

Cria a branch e já mudar para ela

```
$ git checkout -b NOME_DA_BRANCH
```

Trocar de branch

```
$ git checkout NOME_DA_BRANCH
```

Caso tenha um arquivo excluído ou você quer criar uma branch a partir de um commit, eu consigo resgatar ele pela hash do commit '12b24d' (só um exemplo de hash), com isso eu vou clonar essa branch a partir do hash e ter o conteúdo dela!!! (Para pegar a hash é só dar um git log no projeto)

```
$ git checkout -b NOME_DA_BRANCH HASH_DO_COMMIT
```

Renomear uma branch

```
$ git branch -m NOVO_NOME
```

Esse comando envia o push da branch selecionada

```
$ git push REMOTE_NAME BRANCH_NAME
```

Para ser mais específico ainda podemos colocar a branch selecionada que será enviada para a branch selecionada que irá receber no remoto

```
$ git push REMOTE_NAME BRANCH_NAME:BRANCH_REMOTE
```

### Deletando branch remota

Deleta a branch remota, lembre-se de deletar a local também

```
$ git push -d origin NOME_BRANCH
```

### Deletando branch local

Deletar uma branch totalmente (não importa se está com commit ativo ou na área de staging)

```
$ git branch -D NOME_DA_BRANCH
```

Deletar uma branch com segurança (caso ela esteja na área de commit ou de staging)

```
$ git branch -d NOME_DA_BRANCH
```

Lista todas as branches locais, quantas as branches do repositório remoto

```
$ git branch -a
```

Imagine que tenha 2 branches criada no repositório remoto do git, porém você quer criar uma nova branch local para enviar para o repositório remoto do github... Ao criar uma nova branch local ele dará um erro informando que ela não faz parte do repositório remoto do github, e para corrigir isso fazemos o seguinte comando abaixo ✓

Comando abreviado do -U (esse comando linka, conecta nosso repositório local e a branch local com o repositório remoto e a branch remota

```
$ git push --set-upstream origin nome_branch_local
```

---

## 18.0 -> Revertendo GIT ADD – GIT COMMIT e restaurando arquivo do último commit

Nessa sessão você vai aprender a como tirar o arquivo da área de staging, como remover totalmente o arquivo e a restaura o arquivo da forma que estava no ultimo commit

Para reverter um git add ou seja: esse comando 'cached' dizemos para o git deixar de monitorar o arquivo, porém não é para exclui-lo.

```
$ git rm --cached nome_arquivo
```

Logo em seguida que fizer o comando acima precisamos dar um novo git commit para salvar as alterações

Reverte o git add, porém exclui o arquivo.

```
$ git rm -f nome_arquivo
```

O staged vai tirar o arquivo de ser monitorado da área de staging(git add) e o restore vai voltar as alterações anteriores, ou seja, eu fiz um 'void function arquivo\_app\_py (string value)' e dei um git add app.py, beleza agora ele tá na área de staging, porém deu conflito com um merge ou sei lá eu preciso voltar o

arquivo da forma que estava antes com o código que estava antes dessa function que eu criei, eu dou um git restore --staged app.py (isso vai tirar da área de staging) e pra eu voltar o arquivo com os códigos anteriores a função eu coloco git restore app.py e pronto, ele vai voltar com o código do último commit dele !!!

UTILIZE ELES NA MESMA ORDEM ABAIXO!!!

```
$ git restore --staged nome_arquivo
```

```
$ git restore nome_arquivo
```

Os comandos abaixo fazem a mesma coisa dos comandos acima!!! (git restore staged e git restore)

UTILIZE ELES NA MESMA ORDEM ABAIXO!!!

```
$ git reset HEAD nome_arquivo
```

```
$ git checkout nome_arquivo
```

---

## 19.0 -> Revertendo commits e push

Nessa sessão você vai aprender a como descomitar um commit e tira-lo da área de staging sem que remova as alterações dele ou temos a opção de remover as alterações também, vamos aprender também a como voltar os commits especificadamente, como reverter um git push

Reverte o commit e faz a gente volta exatamente para etapa anterior ao commit, ou seja, as alterações continuam feitas e continuam na área do staging (**só vai ser descomitada**)

```
$ git reset --soft
```

**CASO EU NAO PASSE NADA DEPOIS DO 'RESET' O GIT COLOCA ESSA OPÇÃO PADRAO 'MIXED'**, ela reverte o commit e também remove as alterações da área de staging ou seja volta o commit e volta o git add add (dois comandos

em uma operação só), posso utilizar ele com o 'git checkout nome\_arquivo' e com isso vai voltar o commit e a alteração do código do último commit.

```
$ git reset --mixed
```

Reverte o commit reverte o git add e as alterações que fizemos no código (CUIDADO AO USAR, TENHA MUITA CERTEZA DO QUE ESTÁ FAZENDO)

```
$ git reset --hard
```

HEAD~1 diz exatamente pro git que eu quero voltar apenas 1 commit, aí vai de você se quer voltar 2 commits é só trocar o 1 por 2 e assim vai...

```
$ git reset --soft HEAD~1
```

Vai reverter o ultimo commit, ou seja, ele exclui o ultimo commit e volta para o anterior (ele vai até apresenta uma janela no terminal informando sobre o commit que será excluído)

PORÉM O COMMIT CONTINUA LA REGISTRADO !!! só que vai tá como 'revert' que é para mostrar que o commit que tínhamos foi anulado!!

```
$ git revert HEAD
```

Faz a mesma função que eu colocar o git revert HEAD, porém é mais especifico por conta que eu posso colocar o hash do commit que eu desejo da um revert

```
$ git revert HASH_DO_COMMIT
```

Caso eu queria reverter o push que eu fiz... exemplo: eu criei dois arquivos o x e o y, e dei um push, porém não gostei deles e não quero eles no github, primeiro eu dou um git reset --hard ou mixed ou soft dependendo da minha necessidade, e volto até o commit que eu quero, porém eu quero que seja



alterado também no github, com isso eu dou o git push -f, ele vai forçar o github a ficar da mesma maneira do meu repositório local, com os resets que eu dei nos commits !!!

```
$ git push -f
```

*\* git reset → é bom utilizar quando está fazendo projeto sozinho, se existir mais pessoas no projeto tome cuidado pois ele reseta o ultimo commit e pode perder dados, com ele conseguimos: tirar um commit da área de staging para continuar trabalhando nele, descartar commits em um branch privado, desfazer alterações não comitadas.*

*Modos do git reset: --soft, --mixed(mixed é valor padrão caso não coloque nada), --hard, --merge, --keep. Pode ser usado durante os conflitos de merge para redefinir arquivos com conflito para estados 'validos'*

*\* git revert → para usar digite 'git revert 'hash\_do\_commit' para ir em um commit especifico ou 'git revert HEAD' para pegar o ultimo commit, para cancelar o git revert use o 'git revert --abort', tome cuidado pois ele reverte todo o ultimo commit e vai deixar da forma do anterior!*

---

## 20.0 -> Aprendendo a fazer merge

Nessa sessão você vai aprender a como mesclar (exemplo, de uma branch com a main, ou branch com branch etc...)

**Passos para realizar um merge:** de um git checkout master, em seguida de um git merge nome\_da\_branch, se aparecer 'fast-forward' é que deu certo e não teve conflito, aí é só dá um git add e um git commit -m "

*Porém se você fizer o merge e der um conflito vai aparecer ele no vscode e temos as seguintes opções:*

**Accept Current Change** → Aceitar a mudança que temos localmente/em nossa branch atual.

**Accept Incoming Change** → Aceitar a mudança que estamos recebendo de outra branch/remoto.

**Accept Both Change** → Une os dois códigos mesmo com o conflito e você arruma na unha (aceita ambas alterações)

**Compare Change** → Vai comparar o código original com o código que tá vindo do merge

Passa o argumento --merge para o comando do git log, e com isso vai produzir o log com a lista de commits com conflitos entre as ramificações branch

```
$ git log --merge
```

Encerra o processo de merge e faz a branch voltar ao estado em que ela estava antes do merge

```
$ git merge --abort
```

Vai gerar um commit de mesclagem (mesmo no caso de mesclagem de avanço rápido 'fast-forward'), isto é, útil para documentar todas as mesclagens do repositório

```
$ git merge --no-ff nome_da_branch
```

---

## 21.0 -> Unindo commits , salvar código em memória sem precisar dar um git add e um git commit e pegar o commit de uma branch e levar para outra branch com todos os dados inclusos nessa branch

Nessa sessão você vai aprender a como unir vários commits em um só, como salvar o código sem ter que commitar ele. e como transportar um commit de uma branch para outra

Modifica o histórico do projeto, cria novos commits, faz o histórico ficar com um caminho só, tanto o merge quanto o rebase nos dá o mesmo resultado..., porém o merge mante os commits originais.

O rebase elimina a bifurcação e deixa só em uma linha história.

```
$ git rebase
```

**Exemplo de uso do REBASE** → *Imagine a seguinte situação, eu criei uma branch para desenvolver uma nova feature(um ajuste visual), comecei a trabalhar nela ai eu fiz um commit, nesse commit eu acredito que esteja tudo certo e que terminei o trabalho, só que ai fui para o processo de pull request, só que ai nessa pull outros devs viram e revisaram o código antes de aceitar e me informaram que não seguia o estilo do projeto e com isso precisaria fazer algumas modificações para ficar no padrão do projeto... com isso eu voltei no código no meu ambiente de trabalho,*

*realizei a modificação do código e fiz um novo commit porém antes dele ser aprovado pediram pra eu voltar e ajustar mais alguns detalhe que também são visual, fui lá corriji o código e fiz um novo commit (já é o 3 commit), aí finalmente é aprovado.*

*Resumo da história: eu criei um commit com um código da feature, porém eu precisei criar mais dois commits da mesma feature para o código se adequar ao projeto, aí o que acontece, na hora de aceitar minha pull request para fazer o merge, esses commits serão adicionados também.*

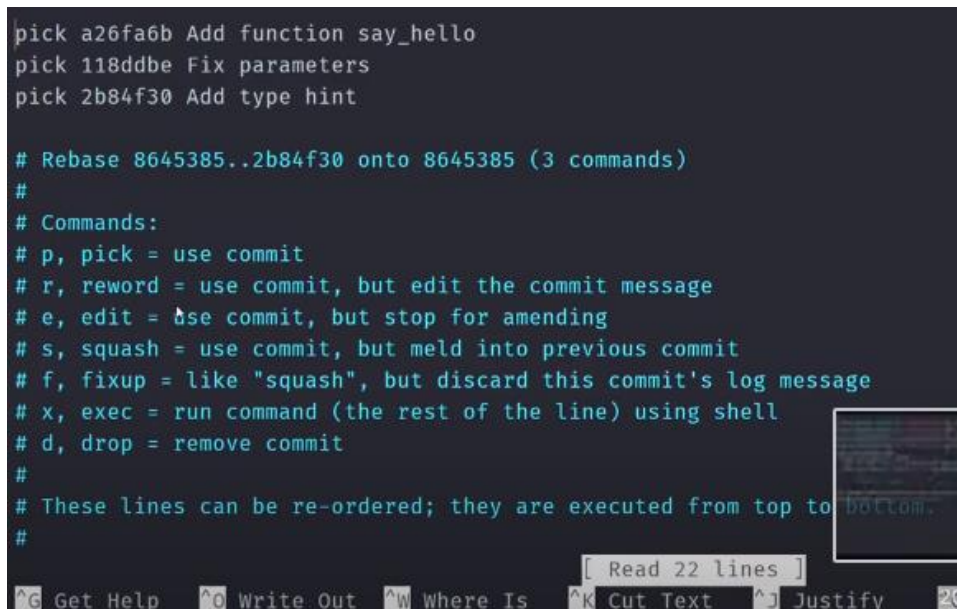
*Olha só que interessante, é muito mais bonito visualmente e fácil de alguém ler quando pegamos o histórico do projeto e temos apenas 1 commit representando a adição dessa fetaure nova ao invés de vários commits irrelevantes para o projeto...em outras palavras eu quero pegar o histórico do projeto lá na branch master e eu não quero ver 3 commits do dev brigando com o código com suas resoluções, o que eu quero é só um commit que indica que nessa data, nesse momento esse feature foi criado...e o git rebase faz exatamente essa função, de unir 3 commits em um só não alterando nada no código, apenas no commit*

Imagine que eu tenha 3 commits, por isso o HEAD~3, o '3' eu to informando que quero unir os 3 commits que tem (posso trocar para 1, 2, 3 isso vai com a quantidade de commits que você quer unir).

```
$ git rebase -i HEAD~3
```

Passo 1: Ao fazer o git rebase -i HEAD~3 vai nos abrir uma tela informando sobre os 3 commits e alguns comandos exemplo:

Os commits vão aparecer assim, com a palavra pick antes deles.



```
pick a26fa6b Add function say_hello
pick 118ddbe Fix parameters
pick 2b84f30 Add type hint

# Rebase 8645385..2b84f30 onto 8645385 (3 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#

[ Read 22 lines ]
^G Get Help  ^O Write Out  ^W Where Is  ^K Cut Text  ^J Justify  20
```

Passo 2: Como vamos realizar um rebase para unificar os 3 commits em apenas 1, não podemos deixar todos como 'pick'.

Por isso eu devo especificar que eu quero 'esmagar' dois commits em um.

Na linguagem do rebase é dar um 'squash' em dois commits em um 'pick'  
Com isso estou dizendo ao git que eu quero que 2 commits sejam esmagados(unidos) dentro de um que será o pick, apenas o commit é alterado (só vai ter 1 commit) porém o código é unido nesse commit.

*Repare que podemos colocar o comando abreviado (p, s)*

**Precisamos saber do pick e do squash**

**p, pick → para escolher o commit que vou usar**

**s, squash → eu quero o que foi feito nesse commit só que ele vai ser esmagado para ser 1 commit só no pick**

```
p a26fa6b Add function say_hello
s 118ddbe Fix parameters
s 2b84f30 Add type hint

# Rebase 8645385..2b84f30 onto 8645385 (3 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to
#
```

[ Read 22 lines ]

^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify 20

Passo 3: Logo em seguida vai aparecer essa tela nos informando que os commits foram unificados em apenas um commit

```
# This is a combination of 3 commits.
# This is the 1st commit message:

Add function say_hello

# This is the commit message #2:

Fix parameters

# This is the commit message #3:

Add type hint

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Wed May 6 21:47:02 2020 -0300
[ Read 28 lines ]
```

Passo 4: Depois desse processo acima, ele vai nos dar uma mensagem de Sucessfully.

ai ao executar o git log --oneline vai aparecer só o commit do pick(só que em uma nova hash ok)

no final é só dar um 'git push -f'

```
(venv) projeto-aula-07 git:(feature-say-hello-sem-rebase) git rebase -i HEAD~3
[detached HEAD 1568292] Add function say_hello
Date: Wed May 6 21:47:02 2020 -0300
1 file changed, 5 insertions(+)
Successfully rebased and updated refs/heads/feature-say-hello-sem-rebase.
```

Git stash, resumindo ele não faz o commit porém guarda o que você fez.

Imagine que você tá em uma branch trabalhando em algo, porém você tem que sair urgente ou alguém pede pra você mostrar algo importante, só que o código ainda tá lá na branch que está trabalhando, ao invés de dar um git add e um git commit o que criaria um commit desnecessário, eu uso o git stash, ele vai salvar o oque estávamos fazendo sem precisar dar um commit

Quando eu utilizar o git stash ele vai pegar todo o código que não foi commitado e vai arquivar ele.

Quando eu uso git stash ele aparece uma mensagem semelhante a essa:  
Saved working directory and index state WIP on feature: 7b878e5 branch feature.

Depois disso eu posso mudar de branch numa boa ou ir fazer outras coisas...

Agora para voltar o código que eu fiz o stash eu faço o seguinte: git stash apply

```
$ git stash
```

```
(venv) projeto-aula-07 git:(exemplo-stash) git stash apply
On branch exemplo-stash
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   app.py

no changes added to commit (use "git add" and/or "git commit -a")
```

Com isso todo o código que foi arquivado no stash(não commitado) voltará a tela!!!

Quando utilizar vai aparecer a seguinte mensagem:

```
(venv) projeto-aula-07 git:(exemplo-stash) git stash
Saved working directory and index state WIP on exemplo-stash: 8645385 Initial commit
```

Mostra a lista de stashes com um índice com todos os stash que fizemos.

```
$ git stash list
```

**Exemplo:**

```
(venv) projeto-aula-07 git:(exemplo-stash) ? git stash list
stash@{0}: WIP on exemplo-stash: 8645385 Initial commit
```

stash@{0}: WIP on feature: 7b878e5 branch feature

stash@{1}: WIP on feature: 7b878e5 branch feature

stash@{2}: WIP on feature: 7b878e5 branch feature

stash@{3}: WIP on feature: 7b878e5 branch feature

stash@{4}: WIP on feature: 7b878e5 branch feature

stash@{5}: WIP on feature: 7b878e5 branch feature

**Podemos pegar essa lista e um índice específico e dar um: git stash apply stash@{3}**

com isso eu posso escolher o stash e ele vai apresentar o código específico daquele stash

Obs.: o stash é para arquivos que ainda não foram salvos!!!

Cherry pick: Permite pegar o commit de outra branch, e trazer para a branch atual.

só que não é tirar o commit de um lugar e colocar em outro... ao pegar o commit de uma branch e colocar em outra, não estamos tirando de um e colocando em outro, nos apenas estamos pegando o commit da branch x e colocando o commit na branch y com as mesmas alterações que estão na branch x, com isso ao colocar ele na branch y vai ser gerado um novo commit. E o commit da branch x vai continuar lá...

```
$ git cherry-pick HASH_DO_COMMIT
```

*porque usar o cherry pick?*

*Imagina que a gente deu um commit em uma branch errada, era pra ser na branch X fizemos na branch Y, podemos mudar para a branch X a certa e em seguida fazemos um git cherry-pick nessa branch X, e ao garantir que pegamos o commit tudo certinho (podemos ver no git log --oneline), na branch X, podemos ir na branch Y e dar um git revert...*

---

## 22.0 -> Aprendendo a fazer Forking Flow

tem uma opção também que se chama 'Forking flow'

**imagina que temos a seguinte situação:**

existe um repositório, porém não temos permissão para altera-lo, mas temos permissão para clona-lo exemplos de repositórios assim são os 'open source', eles são públicos todo mundo pode acessar e clonar, porém nem todo mundo tem permissão de alterar o que tem lá dentro.

Ai caso queremos contribuir com o projeto fazemos o seguinte:

criamos um fork .

O fork é fazer um repositório na minha conta do github que é um clone do repositório que eu quero colaborar.

Depois de fazer o fork, aí sim você clona o repositório para sua máquina local.

Depois de fazer isso no terminal digite 'git remote -v' para você ver que o fork está na sua conta pegando o projeto que deseja.

em seguida é necessário ir no projeto original e pegar a url de clone dele e fazer isso:

git remote add upstream '<https://projeto.github.com/projetoquevamosajudar>', e depois disso ao dar 'git remote -v' teremos o 'origin' e o 'upstream'.

**Mais porque fazer isso?**

Porque agora dessa forma eu consigo trazer pro meu repositório local todas as alterações que são feitas na main oficial.

É só fazer isso: 'git pull upstream main' ao invés de usar 'origin' usaremos 'upstream', depois disso eu posso criar uma branch, para corrigir algo ou adicionar algo novo, git checkout -b arrumando\_bug e nessa branch trabalhamos no que queremos.

No final lembre-se que não podemos enviar as alterações para o repositório oficial, devemos mandar para o meu repositório remoto que eu fiz o fork.

Aqui abaixo ele vai mandar para o meu repositório que eu fiz o fork que está no meu perfil do github, por isso o origin

git push -u origin arrumando\_bug.

E depois disso eu posso abrir uma pull request a partir do meu repositório.

É só ir no repositório oficial, abrir um pull request manual e coloco minha branch - o meu repositório que fiz o fork e mando para o repositório oficial.

Aí é ficar de olho nas discussões pra ver se vai ser aprovado ou rejeitado.

*origin -> repositório da minha conta que está o fork*

*upstream -> repositório oficial que eu apelidei como upstream*