

**M.K Academy**

**Matheus Keshin**

## **Mini curso – GIT e Github**



**SALVADOR  
2021**

## Sumário

INTRODUÇÃO.....	3
Navegação por Ficheiros.....	4
Git por debaixo dos panos.....	6
Connectando-se ao github usando TOKEN de acesso pessoal.....	7
PRIMEIROS COMANDOS COM O GIT.....	8
CICLOS DE UM REPOSITÓRIO.....	10
EMPURRANDO O NOSSO DIRETORIO PARA O GITHUB.....	10
RESOLVENDO CONFLITOS.....	12
TRABALHANDO COM BRANCHES.....	14
RENOMEANDO E DELETANDO UMA BRANCH.....	15
USANDO O STASH.....	15
RECUPERANDO UMA STASH.....	15
LIMPANDO STASH.....	16
COMANDO PARA VER O HISTÓRICO DE ALTERAÇÕES.....	16
VISUALIZANDO HISTÓRICOS DE UM REPOSITÓRIO.....	16
VARIAÇÕES DO GIT LOG.....	17
REVERTENDO COMMITS.....	17
PRATICANDO ESTRUTURAÇÃO DE COMMITS.....	19
VERSIONAMENTO SEMÂNTICO.....	22
GIT E GIT HUB FOCADO EM PULL REQUEST.....	23
FAZENDO O FORK DE UM REPO.....	23
PERMISSÕES PARA MANUSEAR REPOSITÓRIOS NO GITHUB.....	25
ORGANIZAÇÕES E TIMES NO GITHUB.....	26
CRIANDO MODELO DE PULL REQUEST (FORM).....	26
CRIANDO TEMPLATE PARA ISSUES.....	27
ALIASES (APELIDOS).....	31
INFORMAÇÕES EXTRA.....	32
LISTA DE COMANDOS UTILIZADOS AO LONGO DO LIVRO.....	32

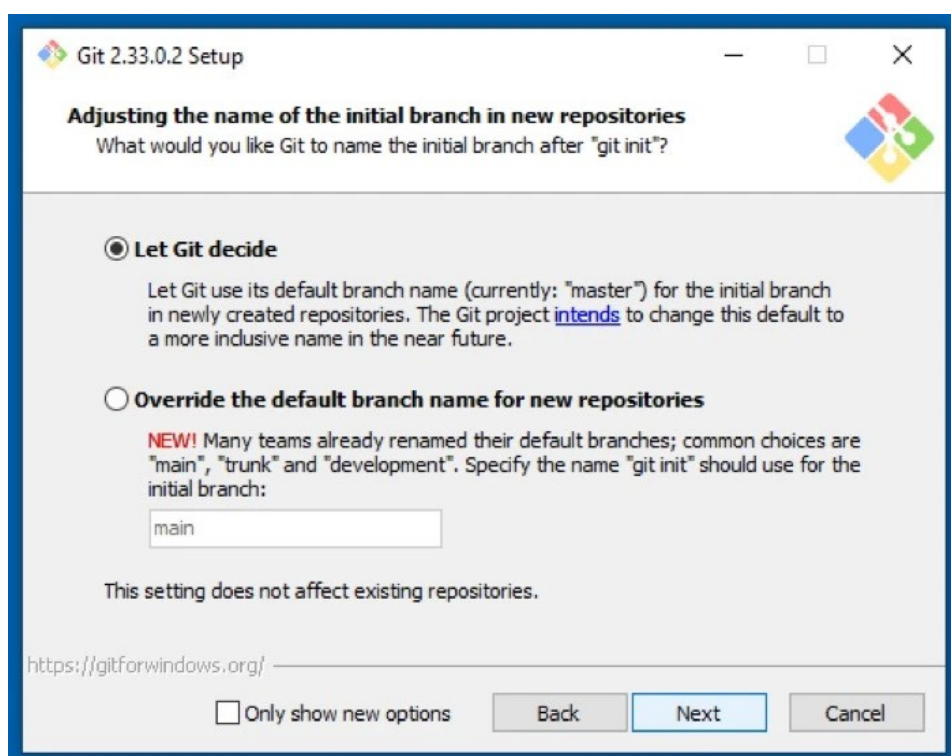
## INTRODUÇÃO

Git é a mesma coisa de Github? Respondendo a essa pergunta, não! O git é totalmente diferente do github mas ambos são muito importantes para o ramo da tecnologia e mais precisamente para a colaboração em projetos. O Git é um sistema de controle e versionamento de código distribuído, ou seja, é nele que versionamos o nosso código enquanto que o github é uma plataforma de hospedagem de códigos-fonte, é lá que guardaremos nosso código e é lá também por onde conseguimos manter um software em um ciclo de mudanças e principalmente softwares livres que se mantêm em constante avanço devido a colaboração de diversos contribuintes.

Então é importante que você já tenha criado uma conta no github e tenha instalado o Git no seu computador, o processo de instalação do git é relativamente simples e não necessita de alterações somente atente-se quando chegar a essa tela a baixo.

**ATENÇÃO:** ESSE LIVRO É UM CONTEÚDO EXTRA PARA ALUNOS DA MK ACADEMY, ASSISTA TAMBÉM AOS VÍDEOS PARA TER UMA MELHOR VISUALIZAÇÃO E COMPREENSÃO! ACESSE O LINK: [encurtador.com.br/pqzCJ](https://encurtador.com.br/pqzCJ)

Nesse momento você pode escolher como deverá ser chamada a sua “branch original”, mais para frente veremos o que é uma branch. Atualmente o git vem mudando várias nomenclaturas a fim de abolir termos pejorativos e fazer mudanças que já deviam terem sido feitas. Hoje as branch principais por padrão quando criadas no github são branches MAIN enquanto que anteriormente eram chamadas de branch master e essa é uma mudança relativamente nova.



Atualmente o git deixa o desenvolvedor no momento da instalação escolher, o que é o caso dessa imagem a cima. Você pode deixar o git decidir ou clicar em override e dar o nome para a branch primaria.

## NAVEGAÇÃO POR DIRETORIOS

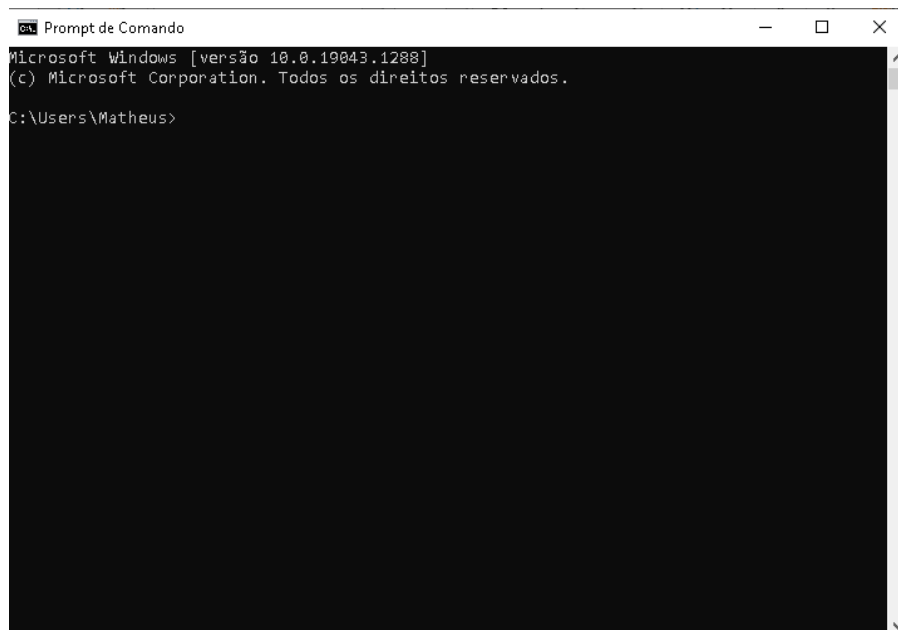
Antes de começar a de fato mexer com o bash do git precisamos entender alguns comandos básicos para navegar por diretórios tanto no linux quanto no windows. O bash é praticamente a mesma coisa do shell que nada mais são do que interpretadores de linhas de comandos sendo que o primeiro é a evolução do segundo.

Começando pelos comandos de navegação por ficheiros do windows usando o CMD.

Com o cmd aberto temos os seguintes comandos:

- **CD / CD..** → CD nomedapasta para entrar em uma pasta, CD.. para voltar a anterior ou somente CD para exibir o seu diretório atual (sua localização).
- **DIR** → para listar pastas e arquivos das pastas.
- **MKDIR nomedapasta** → para criar pastas.
- **DEL OU RMDIR /S /Q nomedapasta** → para deletar itens.
- **CLS** → para limpar a tela do CMD.

Como exercício se você estiver utilizando o Windows use o comando **DIR** para listar todos os arquivos de onde você estiver e tente entrar e sair de alguma pasta usando o comando **CD** para abrir uma pasta e **CD..** para voltar a pasta anterior. Se você quiser ter um pouco mais de praticidade quando for escrever o nome de uma pasta extensa, use a tecla TAB para usar o autocompletar. O autocompletar pelo TAB funciona também no linux.



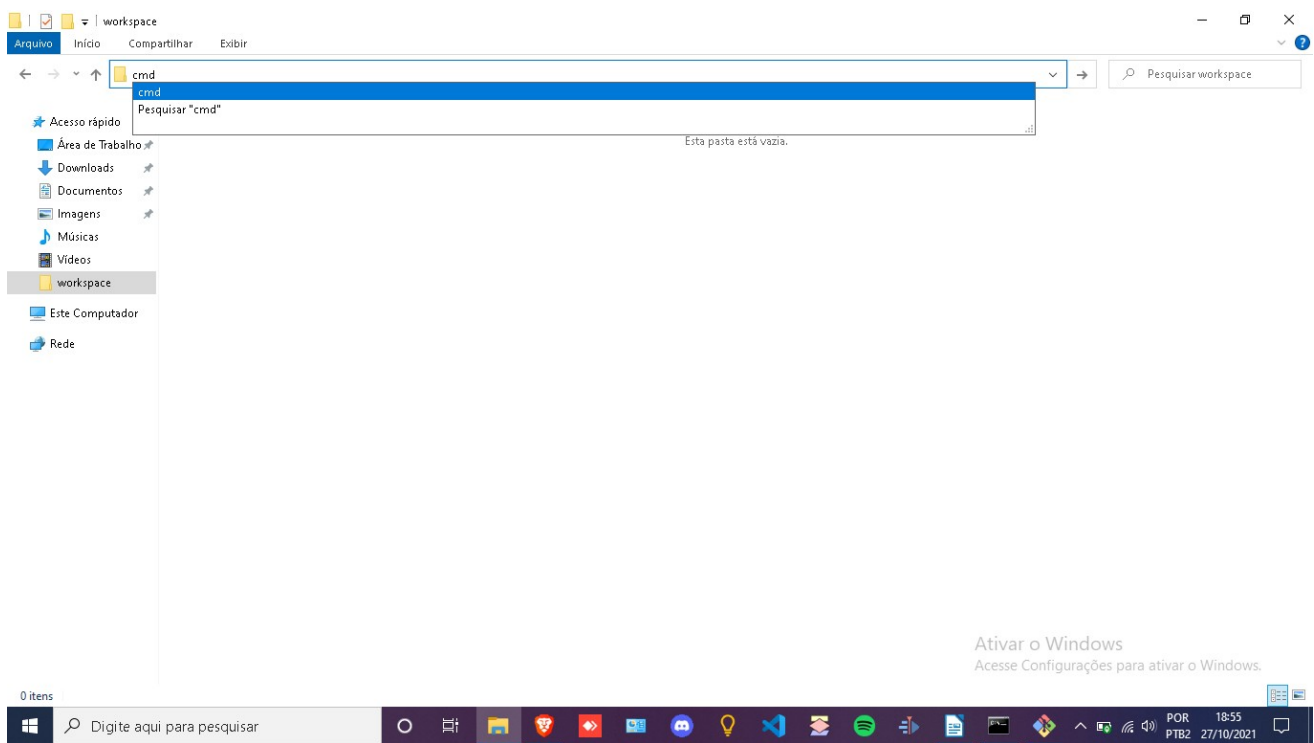
Agora no linux, temos os seguintes comandos:

- **CD / CD..** → CD nomedapasta para entrar em uma pasta ou CD.. para voltar a anterior, ex: “cd /c” entra no disco local C:
- **LS** → listar pastas e arquivos que estão na pasta atual.
- **PWD** → para mostrar sua localização atual de diretório.
- **MKDIR** → MKDIR nomedapasta, cria uma pasta com o nome desejado.
- **RM -RF** → apagar diretórios e arquivos.
- **Clear** ou CTRL+L → Limpa a tela do bash.

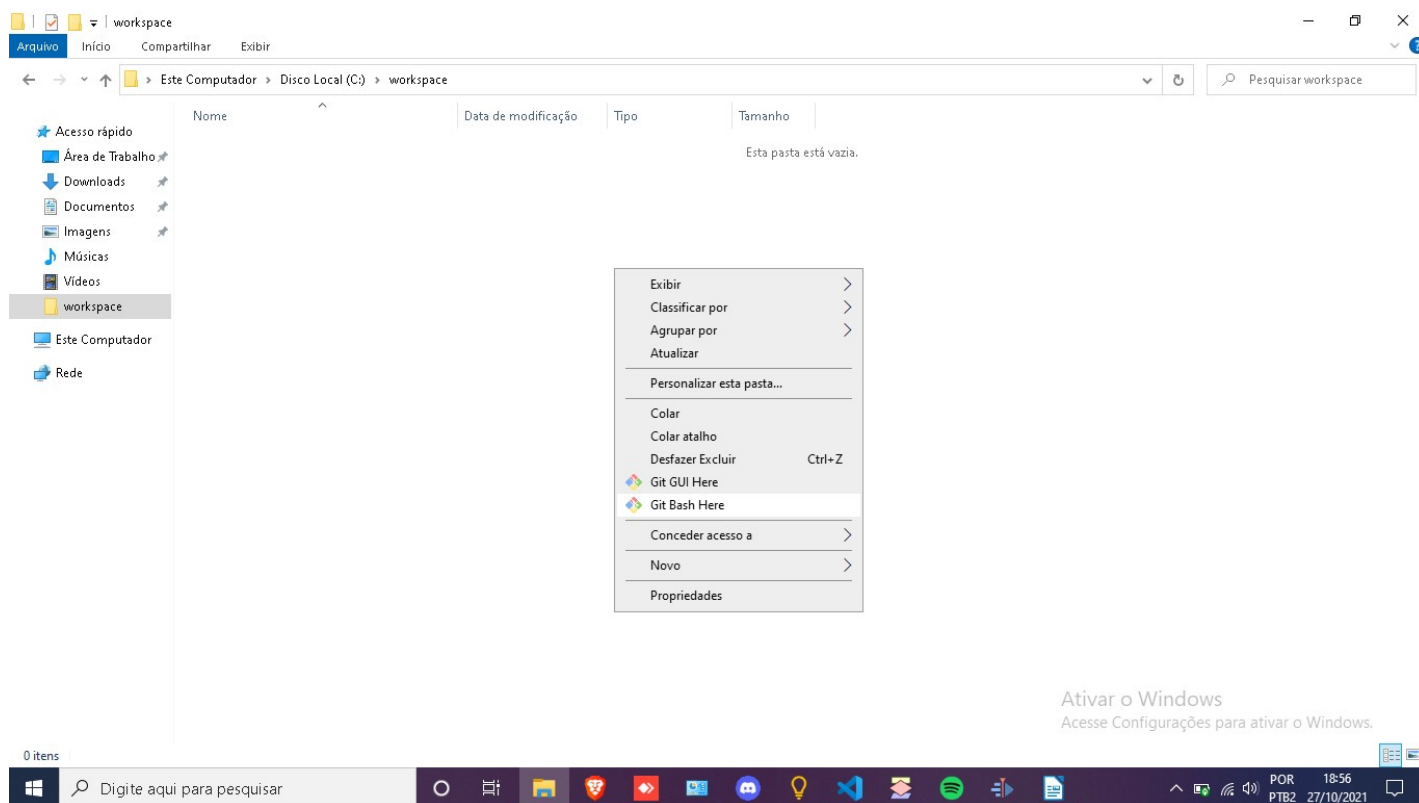
O TAB para usar o autocompletar também funciona no bash do linux. Além desses comandos mais simples temos comandos com complementos chamados de “flags” que geralmente são algumas modificações simples como por exemplo “`git checkout -b novab`” onde -b é uma flag que complementa o comando checkout e serve para criar branches.

Quando por exemplo não queremos ficar dando yes nas instalações de programas do linux por linha de comando adicionamos a flag `-y` que dá yes para tudo que o programa perguntar assim facilitando a instalação, temos também a -f que é a force, esse comando força para que algo aconteça e ignora mensagens de erro. O windows também faz uso das flags mas nós precisamos apenas saber o somente os comandos aqui apresentados para usar em nosso curso e ao longo dele podemos introduzir novos comandos.

Agora vamos usar um comando simples chamado “ECHO” ele printa na linha de comando o que você escrever! Por exemplo abra seu cmd ou bash do windows e digite o seguinte: `echo hello world!` E veja a resposta da linha de comando, isso mesmo, ela repetiu o que você escreveu. Agora vamos tentar inserir essa mensagem em um arquivo e salva-lo em nosso computador! Antes de qualquer coisa inicie o bash ou o cmd em um local onde não tenha muitas pastas ou arquivos. No windows temos uma maneira bem simples que é indo para uma pasta qualquer e clicando em cima da localização do diretório e escrevendo CMD ou se quiser usar o bash do git você pode clicar com o botão direito e escolher git bash here.



OU



E agora com o Bash ou cmd aberto escreva o seguinte comando: `echo oi meu chapa > meuarq.txt`. Perceba que um novo arquivo foi criado e se você abrir ele verá que dentro dele tem a mensagem “oi meu chapa”. Guarde esse comando de criação de arquivos de texto pois iremos utilizá-lo bastante. A partir de agora unificaremos as coisas e usaremos somente o GIT BASH no windows e no linux e não mais o CMD! Atente-se para os comandos com o fundo escuro e letras brancas! Isso quer dizer que esse comando deve ser usado dentro do git bash.

Com o git bash aberto escreva novamente o comando `echo` que cria arquivos e para ler o que tem dentro desse arquivo de texto usamos o comando `CAT nomedoarquivo`, nesse caso `cat meuarq.txt` e então você visualizará a mensagem dentro do arquivo.

## GIT POR DEBAIXO DOS PANOS

O git trabalha com o SHA1 que é nada mais do que um padrão de criptografia, ou seja, seus arquivos são criptografados e com essa encriptação dos arquivos tem-se uma chave de 40 dígitos e é uma chave única, se você alterar um ponto uma vírgula ou mesmo um espaço em um dos arquivos contidos você fez uma modificação que gerará uma nova chave SHA1 com 40 dígitos e caso essa pequena mudança seja desfeita no momento de geração da nova chave irá ser exibida a chave anterior. Essa chave é uma maneira curta de representar todo o documento.

## CONECTANDO-SE AO GITHUB USANDO CHAVE SSH E TOKENS

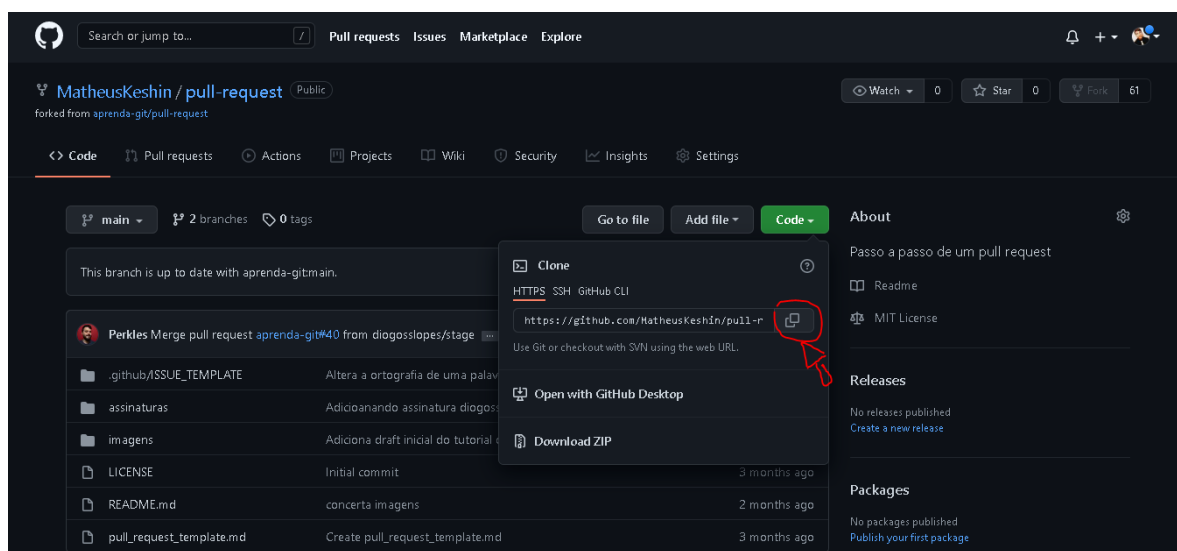
Entendido o conteúdo anterior a este, agora faremos a conexão do nosso git ao nosso github usando conexões SSH e TOKENS para que com essa conexão estabelecida possamos enviar e puxar códigos do nosso repositório do github.

E vamos ao passo a passo, começando pela conexão SSH que é uma conexão segura e encriptada:

- 1- Entre no github e faça login, em seguida clique em settings e procure por SSH e crie uma nova SSH.
  - 2- Digite o comando no Git bash, seguindo as letras maiúsculas e minúsculas como no código!  
`ssh-keygen -t ed25519 -C "seuemail@gmail.com"`
  - 3- Após gerar a chave deve-se navegar até onde a chave está salva e abrir o arquivo .pub
  - 4- Visualize o conteúdo do arquivo .pub usando o comando CAT  
`cat id_ed25519.pub`
  - 5- Cole a chave que está dentro desse arquivo no github
  - 6- Inicie o agent no bash com o comando, esse agent gerencia sua chave ssh  
`eval $(ssh-agent -s)`
  - 7- Após iniciar o agent devemos passar a chave privada para ele, dentro da pasta .ssh (chave privada não tem o .pub)
  - 8- Ele pedirá a mesma senha criada no momento da criação da chave ssh
  - 9- Após tudo certo o bash dirá que a identidade foi adicionada.
- E pronto! Sua conexão ssh com o git e github já está funcionando.

## CONECTANDO-SE AO GITHUB USANDO TOKEN DE ACESSO PESSOAL

- 1- Dentro do github procure por: Settings / Developer settings / Personal access token / generate new token
- 2- Marque a opção Repo, escolha o tempo para o token expirar, dê um nome para o token e clique em generate.
- 3- Salve o token gerado pois ele não aparece uma segunda vez, use o bloco de notas ou algum programa que você possa colar o token ou você perderá esse token e terá que gerar outro!
- 4- Tente clonar qualquer repositório seja seu ou de outra pessoa usando o https, como na imagem abaixo clique em code dentro de um repositório após isso selecione https e clique em copiar.



5- use o seguinte comando para clonar um repositório:

```
git clone COLEAQUIOLINKCOPIADO
```

6- O github abrirá uma interface de login para você fazer login e irá ter a opção de acesso por token pessoal, agora é só colar aquele token gerado ali e pronto! E lembre-se de que se você colocou um timer até expirar seu token você deve abrir outro quando isso acontecer ou optar pela conexão ssh no tutorial anterior a esse.

## PRIMEIROS COMANDOS COM O GIT

E agora sim! Vamos “meter mão” ou seja começar a trabalhar mais com o git e versionamento do nosso código! E para iniciar tudo devemos definir duas informações muito importantes que são o e-mail e o username (nome de usuário) e tanto o e-mail quanto o nome de usuário para facilitar o processo tem de ser iguais aos dados usados no github mas isso não é obrigatório que esses dados sejam iguais.

Para dar Set nessas configurações usaremos os seguintes comandos:

```
Git config --global user.email “seuemailaqu@gmail.com”
```

 → define seu e-mail como o responsável por um commit que é uma etapa muito importante do git.

```
Git config --global user.name “seunomededeusuarioaqu”
```

 → define seu nome de usuário.

E para desfazer essas alterações caso você precise, você usará os comandos:

```
Git config --global --list
```

 → para listar todas as configurações globais feitas por você assim mostrando as configurações que você fez de e-mail e username ou outras.

```
Git config --global --unset user.email
```

 → desfaz a configuração de e-mail.

```
Git config --global --unset user.name
```

 → desfaz a configuração de user name.

Agora que essas informações estão setadas vamos definir uma pasta a ser usada como *Workspace* que é uma pasta onde todos os seus trabalhos a partir de agora serão gerados e salvos. Você pode criar a pasta no linux ou no windows usando a linha de comandos ou fazer isso usando o modo convencional e abrir dentro dessa pasta o git bash.

Vamos começar agora o nosso repositório local, um repositório no nosso computador! E para iniciar vamos usar o comando:

```
Git init
```

 → inicia um repositório.

A partir de agora você iniciou seu repositório! Se você usar o comando LS para listar pastas e arquivos dentro desse diretório você verá que não tem “nada” mas se você usar o mesmo comando com a flag -a esse comando mostrará pastas ocultas, `ls -a`, pastas ocultas começam com um ponto na frente do nome da pasta como no caso .git que irá ser mostrada.

Navegue até a pasta usando `cd .git/` e dê outro `ls -a` para observar o que existe dentro dessa pasta e após isso sem alterar nada use o comando `cd ..` para retroceder.



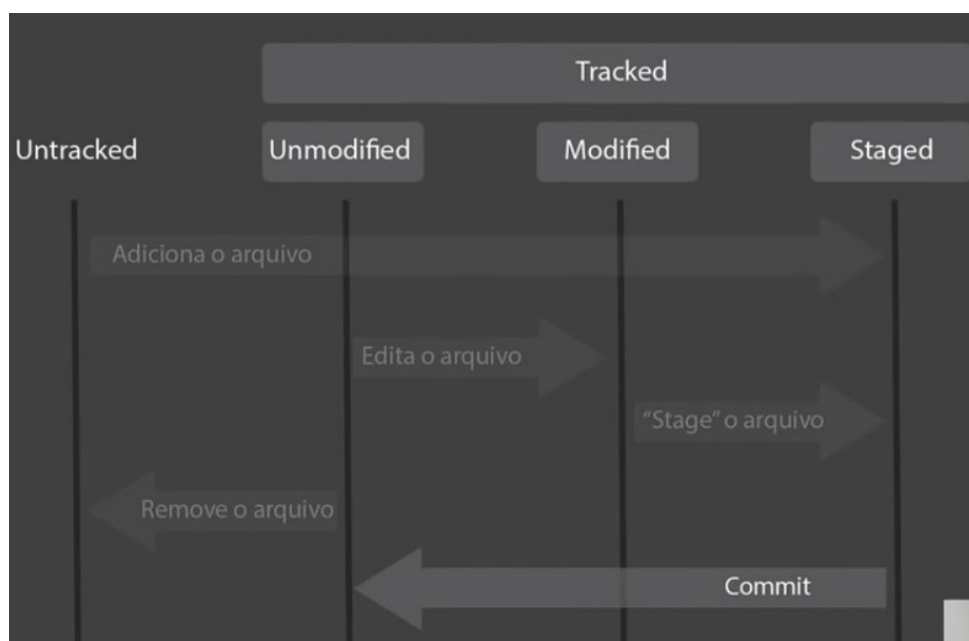
Agora simularemos como se tivéssemos criado um código dentro dessa pasta gerando arquivos dentro dela usando o comando `echo` isso é um test > test1.txt. Com esse novo arquivo gerado precisamos adicioná-lo ao nosso repositório para que ele saiba que esse arquivo está lá e o reconheça oficialmente como parte do repositório por isso usaremos o comando `git add *` para adicionar todos os documentos criados ou editados nessa pasta e o próximo processo é o Commit que é um processo muito importante pois é nele que é gerado um SNAP SHOOT que nada mais é que uma espécie de foto tirada do seu repositório salvando o estado atual no qual ele se encontra.

Então o processo natural para gerar os commits são 3:

- 1- **Git init** → toda vez que criamos uma pasta na qual ainda não demos o `git init`.
- 2- **Git add \*** → usamos esse comando toda vez que criamos um arquivo novo ou editamos um arquivo antigo e precisamos readiciona-lo, sendo que o asterisco representa a palavra all, todos os arquivos contidos ali que precisam ser adicionados ao repositório. Caso queira adicionar um arquivo específico você pode usando `GIT ADD` nomedoarquivo
- 3- **Git Commit -m "Mensagem do que esse commit faz"** → esse comando cria um commit que é uma espécie de foto tirada no momento em que o diretório se encontra e essa foto é salva para ser usada para ser recordada no futuro ou fazer parte de uma lista de historico de alterações ao longo do programa criado, todo commit tem uma identificação chamada Hash q é um tipo de chave para recobrar um commit anterior ou verificar se algo mudou no arquivo.

OBS: também temos um comando chamado `Git Status` → esse comando mostra o estado desse repositório no momento e o nos dá dicas do que fazer, por exemplo agora crie outro arquivo usando o comando no git bash `echo isso é outro test > test2.txt` após isso dê `git status` e leia a mensagem em inglês, caso queira pode também traduzi-la. Ao traduzir verá que o git diz que tem um arquivo untracked ou seja tem um arquivo que não está rastreado, não está de fato presente nos dados do repositório e para fazer ele ser reconhecido podemos usar o `git add *` para que ele seja de fato parte do repositório, em seguida de outro `git status` e o git bash irá nos mostrar a mensagem que diz que temos mudanças a serem commitadas, por fim faremos o commit do arquivo usando o comando **Git Commit -m "adiciona um segundo arquivo"** e caso você faça uma exclusão direta na pasta ou mova um arquivo para outro diretório dentro da pasta o git entenderá que a foto tirada no commit anterior não está de acordo com o momento de agora no qual algo foi excluído, alterado ou movido para outra pasta e então será necessário novamente usar os comandos para adicionar e commitar os arquivos em um novo commit.

## CICLOS DE UM REPOSITÓRIO

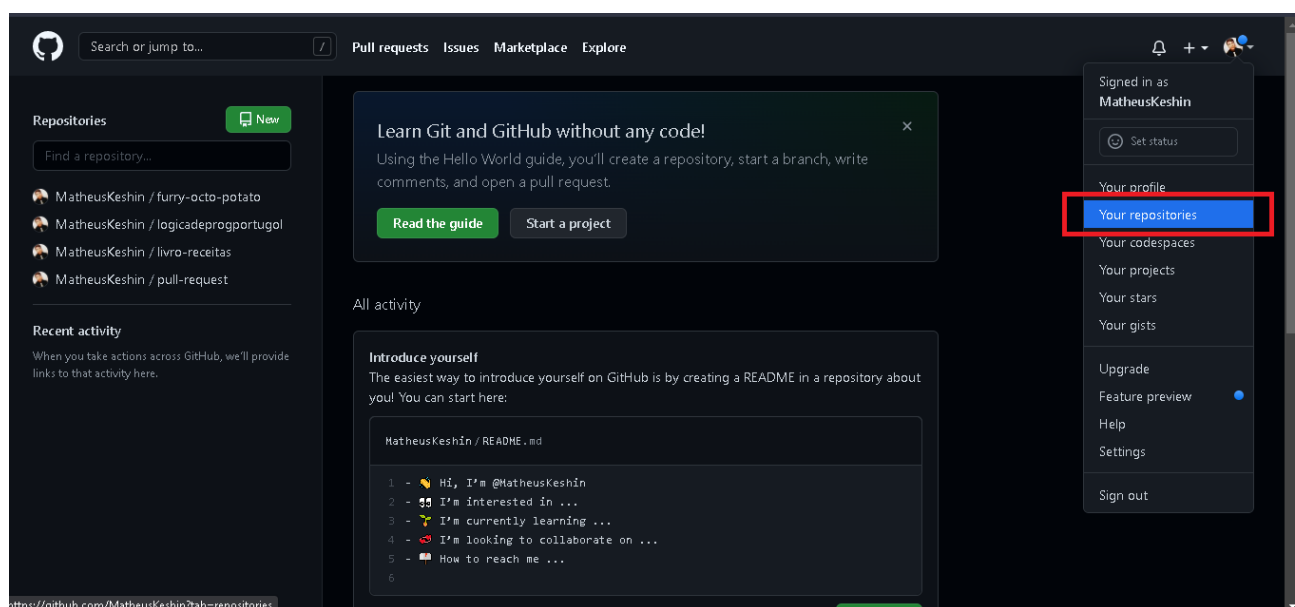


A imagem a cima nos mostra o ciclo de um repositório local, após o **git init** um novo repositório é criado e quando criamos um arquivo dentro dele o arquivo encontra-se UNTRACKED para isso usamos o **git add \*** que nos leva de UNTRACKED para STAGED que é basicamente uma área os arquivos estão sendo preparados para serem commitados, ou seja, uma foto do diretório como ele está agora será salva, após executar o comando **git commit -m "mensagem"** o arquivo vira um arquivo UNMODIFIED e caso ele seja alterado ele vai para o estado de MODIFIED onde terá que ser executado novamente git add e commit para ele voltar a UNMODIFIED, e não se esqueça que entre os processos você pode usar o **git status** para verificar como está o estado do seu repositório local e se é necessário o uso de algum comando para que tudo seja commitado e esteja pronto para ir ao github que é o nosso repositório online.

## EMPURRANDO O NOSSO DIRETORIO PARA O GITHUB

Agora vamos ao passo a passo para hospedar nosso repositório local no github:

1- Abra o github e clique em repositório no seu perfil!

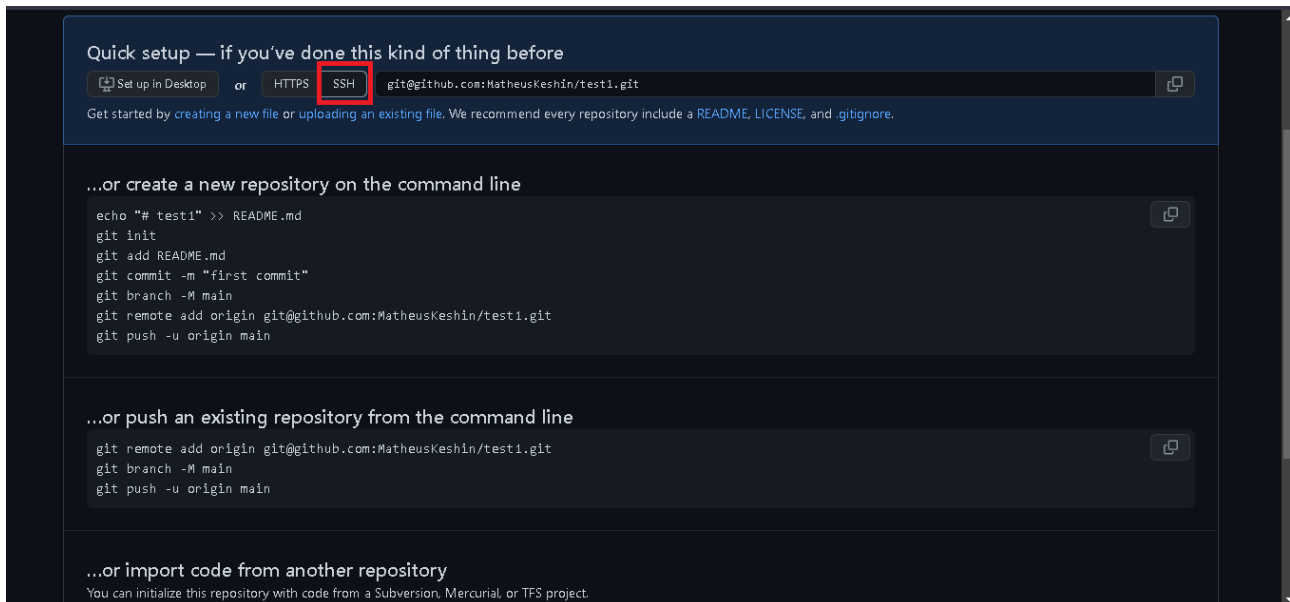


2- você ira ver um botão escrito **NEW** onde irá clicar para criar um novo repositório.

3- Escolha um nome para esse repositório e clique em **CREATE REPOSITORY**

4- Com o repositório criado agora vamos copiar o link dele e usar no nosso git bash, mas antes de copiar é importante saber qual modo de conexão você utilizou caso seja o SSH você clicará em SSH e copiar e caso seja TOKEN você irá escolher HTTPS. Recomendo que use o SSH pois é esse que utilizaremos a partir de agora, caso ainda não tenha configurado é só voltar nas páginas anteriores e seguir as instruções.

5- Selecione SSH e copie o link informado para utilizar no git bash e fazer a conexão entre seu GIT (repositorio local) eu GITHUB (repositorio online).



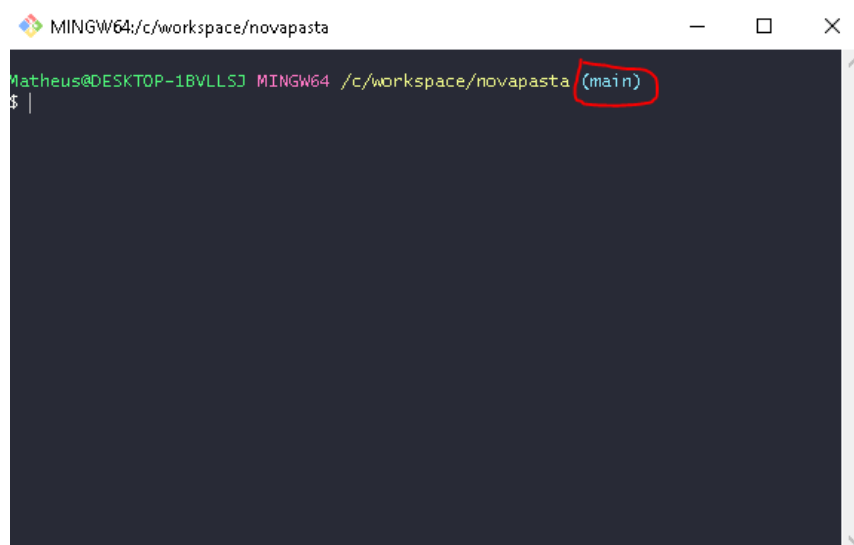
6- Abra o seu repositório na pasta raiz dele e use os seguintes comandos:

6.1 – **Git remote add origin coleaquireulinkssh** → esse comando vincula seu repositório local ao repositório online.

6.2 – **Git remote -v** → esse comando vai mostrar para onde seu repositório local está apontando

6.3 – **Git Status** → verifica se o repositório está com os arquivos prontos para envio para o github

6.4 – **Git push origin main** → empurra seu código para o github, verifique como o seu gitbash chama sua branch original se é main ou master como na imagem abaixo. Para isso você observará no próprio git bash e caso seja master substitua o main por master nesse comando. Se o git bash pedir uma senha vai ser a senha que você definiu lá no momento de ativação do SSH no seu git bash.



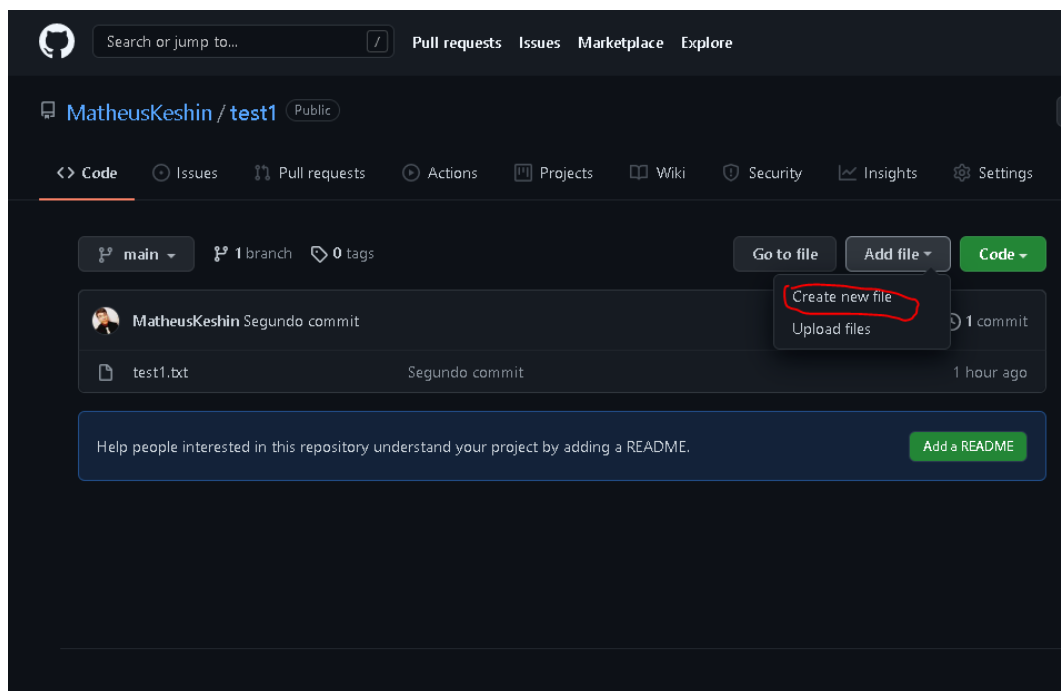
7 – Agora você já tem seu repositório criado no github! E pode verificar isso indo em repositórios no seu perfil.

## RESOLVENDO CONFLITOS

Agora imagine que você no seu repositório local (repositório na sua máquina) cria um novo arquivo e não usa o comando para adicionar esse novo arquivo (`git add *`) e nem usa o comando para commitar e mesmo assim quer enviar esse código para o github ele dirá que tudo está sincronizado pois ele ainda não reconhece os arquivos recentemente adicionados, para isso é importante que antes de qualquer envio para o github deve-se utilizar o comando `git status` para verificar se há necessidade de usar os comandos de adicionar arquivos e o de commit.

O segundo conflito é quando temos em nosso repositório online (github) um repositório que está mais atualizado do que o nosso e tentamos mesmo assim fazer um envio de um repositório local “desatualizado” para simular isso faremos o seguinte:

1- vá até seu repositório no github e lá crie um novo arquivo, como na imagem a baixo!



2- Nomeie seu novo arquivo, escreva algo dentro do arquivo e clique em **COMMIT NEW FILE**, em seguida vamos criar um arquivo no nosso repositório local e dar `git add + commit` e fazer o push(empurrar) para o repositório do github.

```
MINGW64:/c:/workspace/novapasta
Matheus@DESKTOP-1BVLLSJ MINGW64 /c:/workspace/novapasta (main)
$ echo aaaaaaaaaaaaa > novarq53.txt

Matheus@DESKTOP-1BVLLSJ MINGW64 /c:/workspace/novapasta (main)
$ git push origin main
Enter passphrase for key '/c:/Users/Matheus/.ssh/id_ed25519':
To github.com:MatheusKeshin/test1.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'github.com:MatheusKeshin/test1.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.

Matheus@DESKTOP-1BVLLSJ MINGW64 /c:/workspace/novapasta (main)
$
```

Perceba que o nosso push foi rejeitado pois como eu tinha dito antes, em nosso repositório online tem arquivos novos que foram comitados pelo github e que não existem no nosso repositório local, para resolver esse conflito usaremos o `git pull origin main` esse comando vai puxar tudo do nosso

reposto rio do github para nosso repositório local (nosso pc) e a partir daí temos um repositório atualizado porém não teremos o arquivo que foi antes criado, então teremos que cria-lo novamente e após isso usar os comandos add + commit e por fim o `git push origin main` para fazer envio do nosso código para o github sem problemas. Além disso também podemos usar o comando `git clone linkparaclona` e aí não perdemos os códigos que editamos anteriormente e tentamos fazer a mescla desses dois e por fim, enviar para o git sem mensagens de erro.

Então antes de tentar enviar qualquer arquivo para um repositório utilize `git status` para verificar se tá tudo bem e sem alterações para serem feitas, veja se seu repositório local está atualizado e após estar tudo em seu devido lugar, é só fazer o push.

Bônus: outro comando para criar arquivos é o `Touch nomedoarquivo.extensao`

## TRABALHANDO COM BRANCHES

Agora entenderemos o que são as branches e como elas funcionam assim como os comandos para usar para várias coisas com relação a branches. Traduzindo a palavra branch para o português temos as possíveis traduções: ramificação, bifurcar, ramo, galho, derivar e etc. Ou seja, uma branch do nosso git é nada mais nada menos do que uma linha de trabalho onde de podemos trabalhar e a essa linha pode derivar outras sendo a primeira e mais importante a linha main e podemos nomear outras linhas de trabalho que derivam dessa main e trabalhar em novas coisas dentro dessa linha paralela sem afetar a linha main, uma branch derivada de uma main vai carregar todos os arquivos da main até aquele momento criados, ou seja, uma branch paralela vai herdar arquivos da branch main e adicionar novas atualizações desse código e mais para frente após tudo estiver certo podemos fazer a mescla da branch main e branch paralela para que a linha paralela funda-se a linha main e tornem-se uma nova branch main.

Quando por exemplo temos um programa V1.0 e colocamos novas funcionalidades (features) ou corrigimos um bug em uma versão nova V1.1.0 ou V1.0.1 podemos fazer isso sem comprometer o programa na versão V1.0 que é a versão PRIMARIA versão MAIN/MASTER do programa então para isso criamos uma bifurcação ou uma nova branch assim tendo uma linha de trabalho MAIN e uma segunda linha de trabalho para testes, correções e features podendo termos mais de uma branch porém a branch principal será sempre a main/master então assim que terminamos de consertar um bug em uma nova versão que estamos trabalhando em uma branch paralela temos que fazer o MERGE que é uma MESCLA de duas branches. Nesse processo o git pega o que tem de novo e junta com o que tem da versão antiga (V1.0) e cria uma nova versão com algo mais recente.

Agora vamos criar uma branch em nosso repositório local usando o git bash, com o git bash aberto no diretório de trabalho onde já temos nossa pasta na qual estamos trabalhando desde o início do livro usaremos os seguintes comandos:

- `Git checkout -b branch_dahora` → com esse comando criamos uma branch nova chamada “branch\_dahora” mas o comando git checkout é usado originalmente para mover-se entre branches e commits o que faz ele conseguir criar uma branch é a flag(complemento) -b. Caso queira outro nome no lugar de “branch\_dahora” é só criar uma branch com o nome desejado no lugar.
- `Git branch` → esse comando mostra as branches criadas nesse repositório e em \*verde temos a branch na qual nos encontramos.

- **Git checkout main** → com esse comando estamos nos movendo para a branch main e caso sua branch seja master é só trocar o main por master ou se desejar mover-se pra uma terceira branch ou qualquer outra é só colocar o nome dela no comando no lugar de main ex: git checkout terceira\_branch

Quando criamos uma novabranh ela carrega os arquivos da branch main e a partir dai você complementa com mais arquivos para trabalhar em paralelo sem comprometer a branch main que pode estar em uso por alguém. Feito as alterações necessárias na branch em paralelo a branch main você então precisará fundir a branch na qual você estava trabalhando com a branch main e para isso você vai fazer o seguinte:

1- Mova-se até a branch main ou master com o comando **git checkout main**

2- escreva o comando **git merge nomedasuabranh**. → esse comando mesclará as duas branches e se estiver tudo certo a sua branch vai se integrar a branch main e a main vai ter uma nova versão.

## RENOMEANDO E DELETANDO UMA BRANCH

Renomeando:

1- escolha a branch e mova-se até ela com checkout

2- Digite o comando **Git branch -m novo\_nome**.

OU

1- Estando na branch main você pode alterar o nome de outra branch com o comando **git branch -m nome\_antigo novo\_nome**.

Deletando:

1- use o comando **Git Branch -d nome\_da\_branch** → esse comando deleta uma branch especifica, no lugar onde tá escrito nome\_da\_branch substitua pelo nome da sua branch que você deseja deletar.

## USANDO O STASH

Imagine o comando stash como sendo uma caixinha onde você guardas alterações inacabadas para serem continuadas em um outro momento sem ter que adicionar arquivos inacabados ao repositório ou comitar algo incompleto.

Você está em uma sexta trabalhando em um código colossal e seu expediente chegou ao fim, você está numa branch paralela a branch main trabalhando em novas features porém ainda não concluiu o seu trabalho e quer deixar para continuar na segunda feira, você então usa o comando stash para salvar o seu trabalho até o momento. O comando stash é uma espécie de estoque temporário, após salvar as alterações de trabalho no stash e retornar no próximo dia de trabalho você vai usar um outro comando para carregar as alterações salvas no stash e voltar a trabalhar nela.

Então dentro da pasta de trabalho (o seu repositorio local) abra o git bash faça:

1- Mova-se a branch na qual você trabalhará com o comando checkout (sem o -b)

2- Após adicionar ou modificar arquivos existentes deve-se usar o **git add \*** para adicionar os arquivos ao repo.

3- Usar o comando **Git stash save "mensagem contexto"**, a mensagem contexto trata-se de uma mensagem para te lembrar o que você estava fazendo nesse código e o que deve ser feito a seguir.

Pronto! Agora você salvou todas as alterações em um vetor stash que salva suas alterações em andamento que ainda não foram concluídas para serem recuperadas mais tarde para continuar o trabalho.

## RECUPERANDO UMA STASH

1- Dentro da branch onde o stash foi criado, use o comando `git stash list` para ver a lista de stashes e escolha a stash para “estourar”, pop.

2- Use o comando `Git stash pop numerodastash`, onde tem escrito número da stash substitua pelo número id da stash assim estourando e recuperando os dados dentro dela. Após isso você volta a codar e fazer suas alterações dentro de um arquivo, ao finalizar todas as alterações use `git add *` depois `git commit -m “mensagemcontexto”` e após isso é só fazer o push e enviar para seu repositório online(github).

## LIMPANDO STASH

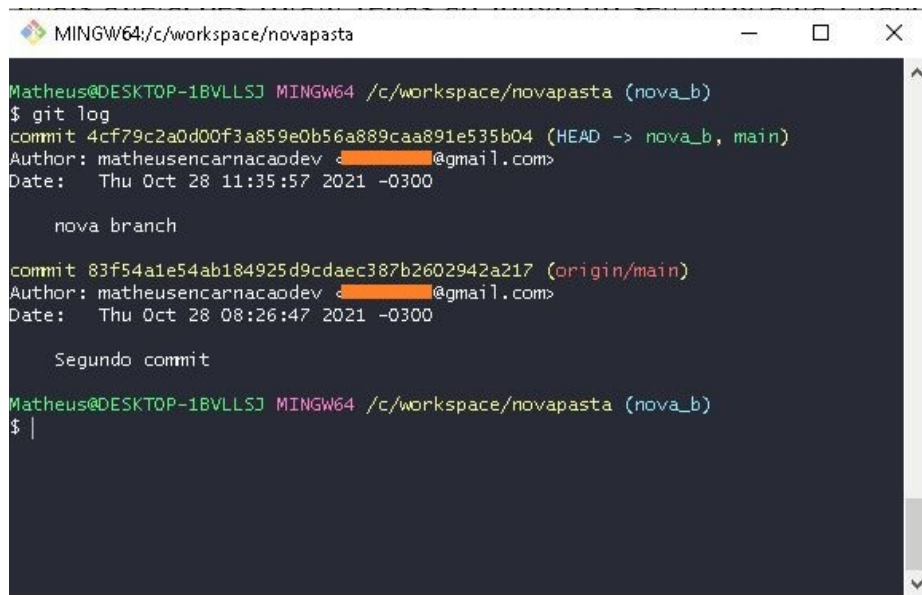
Use o comando `git stash clear`, esse comando limpa todos os stashes dentro de uma branch.

## COMANDO PARA VER O HISTÓRICO DE ALTERAÇÕES

Agora imagine que você quer saber quais alterações foram feitas ao longo do seu programa criado, cada commit representa um ponto chave onde algo foi adicionado ou modificado e todo commit tem alguns dados que são CARIMBADOS nele ou seja esses dados fazem parte do commit, são eles o Hash que é a chave gerada para aquele commit, o e-mail linkado a esse commit, o usuário linkado a esse commit, a mensagem contexto do commit e etc. Lembra quando usamos o comando global para definir e-mail e usuário logo no início do livro ? Pois é eles são importantes para os commit e fazem parte da assinatura de um commit.

## VISUALIZANDO HISTÓRICOS DE UM REPOSITÓRIO

1- Com o bash aberto dentro do nosso repositório local escreva `git log`, esse comando te mostra detalhadamente os dados de um repositório e mostra também o HEAD que é uma espécie de bússola que aponta sempre para o commit mais recente( o último commit válido). Como na imagem a baixo podemos ver vários dados e vemos que temos 2 commit, Obs.: o e-mail está censurado por ser um e-mail pessoal.

A screenshot of a terminal window titled 'MINGW64:/c:/workspace/novapasta'. The prompt is 'Matheus@DESKTOP-1BVLLSJ MINGW64 /c:/workspace/novapasta (nova\_b)'. The user enters '\$ git log'. The output shows two commits. The first is 'commit 4cf79c2a0d00f3a859e0b56a889caa891e535b04 (HEAD -> nova\_b, main)' by 'matheusencarnacao@dev' on 'Thu Oct 28 11:35:57 2021 -0300'. Below it, it says 'nova branch'. The second is 'commit 83f54a1e54ab184925d9cdaec387b2602942a217 (origin/main)' by the same author on 'Thu Oct 28 08:26:47 2021 -0300'. Below that, it says 'Segundo commit'. The prompt returns to '\$ |'.

## VARIAÇÕES DO GIT LOG

O git log tem algumas variações a fim de melhorar nossa visualização deixando a saída de dados mais detalhada, minimalista ou com mais detalhes gráficos.

- **Git log --oneline** → um resumo do git log mais minimalista.
- **Git log --graph** → Output graph um pouco mais ilustrado graficamente.
- **Gitk** → Abre uma ferramenta mais detalhada graficamente, atenção essa ferramenta funciona SOMENTE git instalado em WINDOWS, para usar ferramentas semelhantes ao gitk no linux você pode ir ao site oficial do git e instalar por exemplo o gitg que é semelhante ao gitk ou outro software complementar gratuito do git. Acesse → <https://git-scm.com/download/gui/linux>, na coluna do lado esquerdo em downloads selecione a opção gui clients e em seguida linux e, por fim, selecione a ferramenta escolhida e pesquise mais sobre como utiliza-la no linux e como instalar.

## REVERTENDO COMMITS

Git revert VS Git reset( --soft, --mixed, --hard)

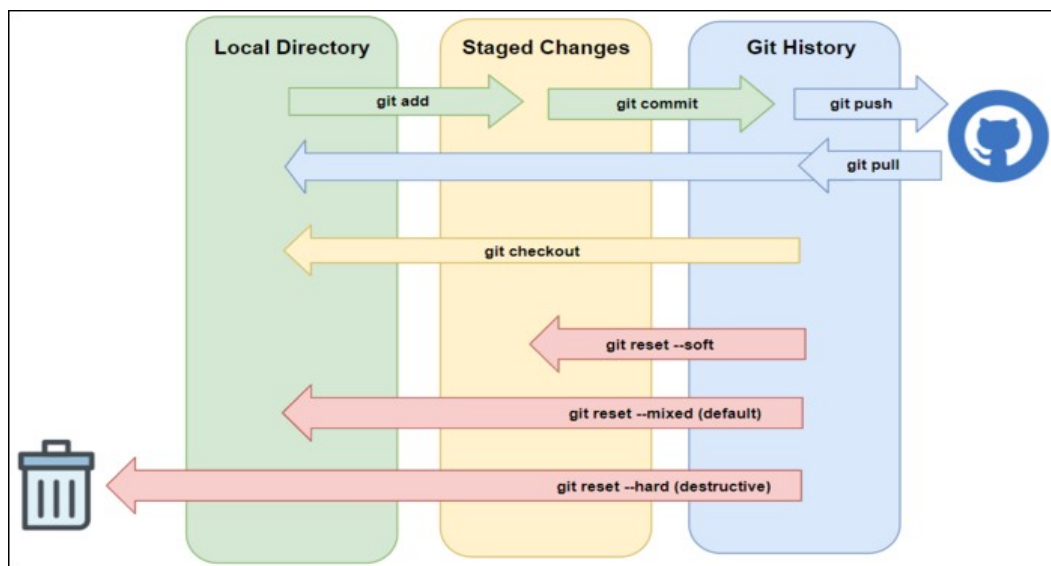
Imagine que você precisa reverter algo em um commit e para reverter algo tem que tomar muito cuidado para não apagar arquivos necessários e causar uma confusão no seu repositório, de maneira geral o git reset é usado para quando você precisar reverter algo em seu repositório local (no seu repositório dentro do seu pc) antes de enviar para o repositório do github(repositório online) o git reset tem 3 variações que são git reset --soft, git reset --mixed e git reset --hard sendo que se você usar git reset puro sem nem um sufixo ele escolherá por padrão o git reset --mixed. É importante tomar cuidado com o git reset --hard o mais perigoso pois ele destrói códigos. Explicando de maneira mais detalhada as variações do git reset:

**Git reset --soft head~1** → a variação soft não altera nada no diretório local e nem desfaz commits, essa variação apenas retira os arquivos da staging area que é a área na qual ficam os arquivos prontos para serem commitados HEAD~1 indica a localização de um commit e nesse caso 1 é o commit recente então no commit recente ele vai tirar os arquivos desse commit da staging area, se head fosse head~2 ele tiraria os arquivos dos dois últimos commits criados da staging area e assim por diante.



**Git reset --mixed head~1** → já aqui o commit atual é deletado e os arquivos criados no commit que foi deletado são mantidos como untracked necessitando fazer um `git add *` e `git commit` para ficar pronto para o push até o github. Como mencionado `head~1` é um apontador que aponta para o commit mais recente valido e se eu aplicar com o head sendo 1 o comando se aplica ao commit recente se for 2 vai ser o aplicado no penultimo commit e assim por diante.

**Git reset --hard head~1** → E esse é sem duvida o mais perigoso pois ele ao contrario dos outros dois deleta tudo, o historico de que o commit existiu, os arquivos criados e alterados. Esse é o comando que você usa com sabedoria ! E eu já expliquei por 2 vezes o head né :v



Já o **Git revert** você vai usar para reverter itens que já foram enviados para o repositório online (github), vamos supor que você tem um repositório online com 3 commits e no seu repositório local você também tem 3 commits e então você decide alterar algo e deletar um commit ficando com 2 commits, você então tenta empurrar esse novo repositório para o github mas ele trará a mensagem dizendo que tem algo errado e que tem um commit faltando assim te impedindo de fazer o push de seus arquivos para o github.

Para resolver isso usamos o `git revert` que como ele diz ele revert um commit porém ele não o deleta! Ele apenas desfaz alterações feitas nesse commit e cria um novo commit sem essas características do commit desfeito sendo que apesar do commit 3 existir na linha temporal do histórico de commits ele não vai ser deletado e um novo commit, o quarto commit vai desfazer o que o commit 3 faria. Nesse caso o github vai entender que nada foi alterado porém que existe um novo commit de número 4 e que o repositório online está desatualizado e em seguida ele aceita a ação de envio para o github e pronto agora todos os seus arquivos do repositório local e online estão salvos e alterados como você preferiu.

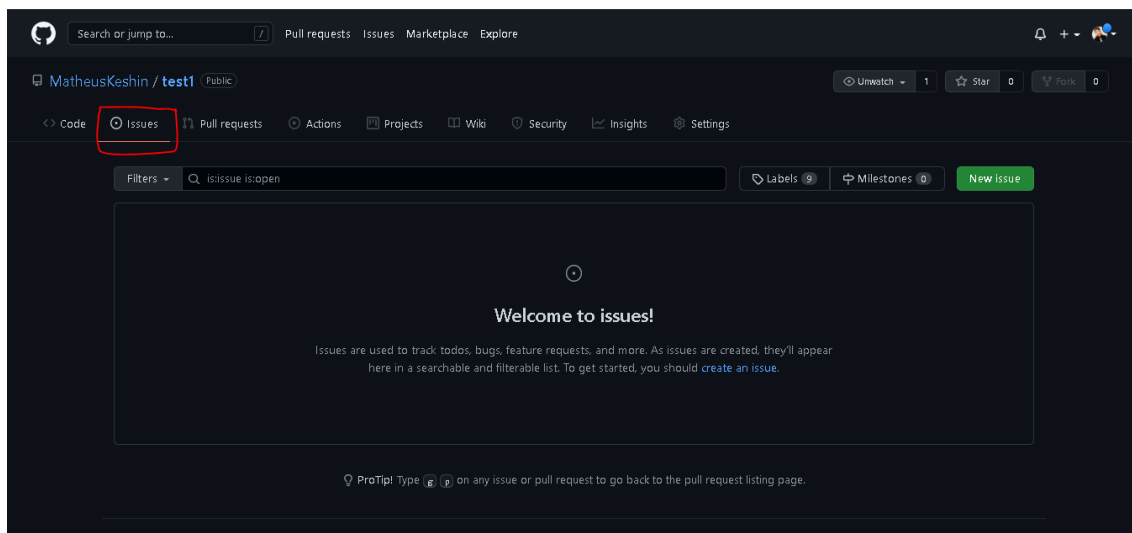
## Estruturando Commits

Os commits são momentos chaves importantes nos quais tudo é empacotado e salvo através de um hash. Existem algumas regrinhas e coisas simples para manter os commits limpos, explicativos e com mensagens bem estruturadas.

Commits atômicos: Isso quer dizer que os commits devem ser únicos ou o menos possível separado, e o que quer dizer isso? Bom, imagine que você faz 3 commits para alterar uma parte visual do programa/site, 5 commits para mudar uma lógica de programação, 10 commits para uma nova feature, tudo isso em um intervalo de tempo curto e esses commits estão todos embaralhados na linha temporal do log. para deixar as coisas mais limpas e “atômicas” você tinha de ter somente 3 commits, ou seja, definir um dia para mexer na parte visual, outro para a lógica de programação e outro para criar features features, ou seja, no final você terá apenas 3 commits a mais na linha temporal e não 18 commits embaralhados entre visual, lógica e novas features.

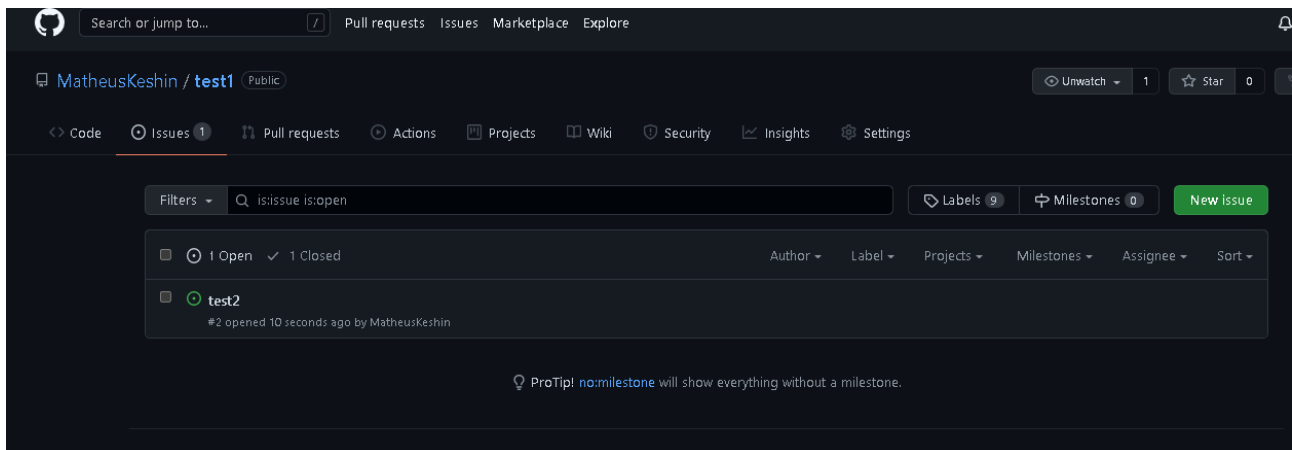
Mensagens de um commits: As mensagens de um commit devem ser imperativas isto é, quando descrita a atividade que aquele commit implementa ou faz não deve ser escrito por exemplo: “adicionado parte visual” a mensagem correta seria “adiciona parte visual” como se fosse realmente feita uma pergunta a cada commit tipo: o que esse commit faz ?

## PRATICANDO ESTRUTURAÇÃO DE COMMITS

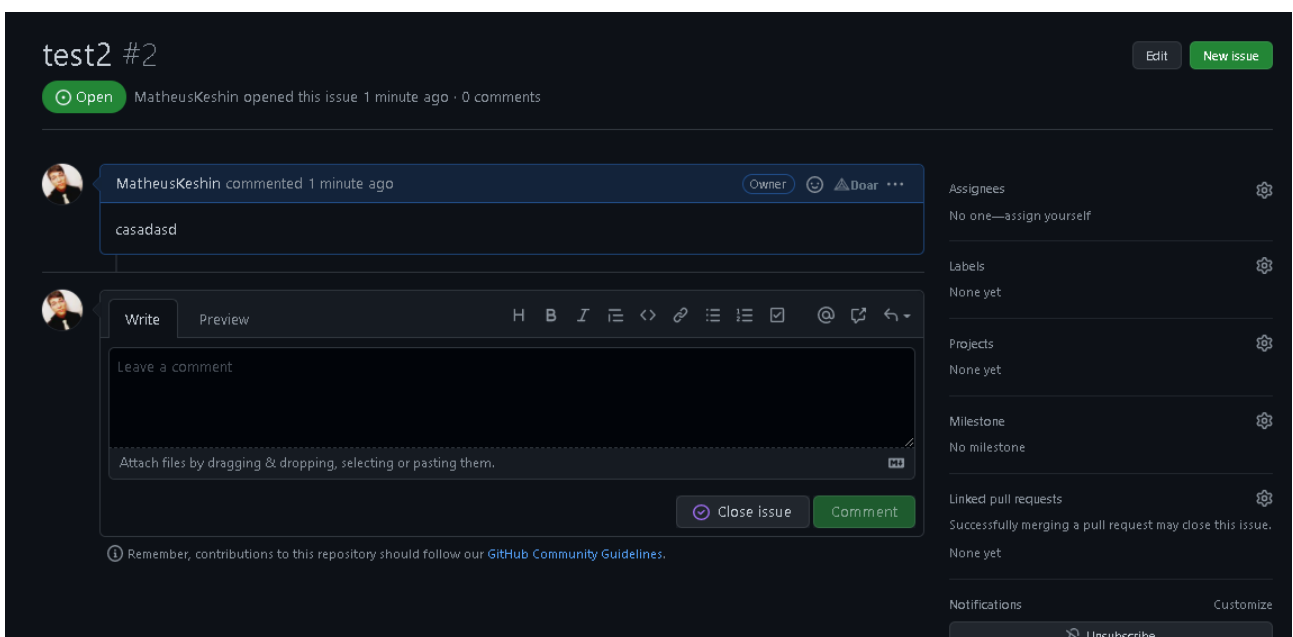


Agora vamos tratar das ISSUES você sabe o que isso significa? As issues são uma aba dentro de um repositório com sugestões de melhorias, apresentação de bugs para os criadores, sugestões de novas features. Resumindo essa aba dentro de um repo do github é um fórum de discussões para melhorias e correções de bugs. Para abrir uma nova issue é só clicar no botão verde ao lado escrito new issue e você pode ver issues abertas e fechadas, pode reabrir fechadas ou fechar abertas.

Para fechar issues você pode fazer isso em uma mensagem de commit, isso mesmo! Quando você comita algo no corpo de mensagem você pode fechar um commit que está aberto.



Perceba que temos duas issues porém apenas uma delas está aberta! Vamos clicar na test2 e ver.

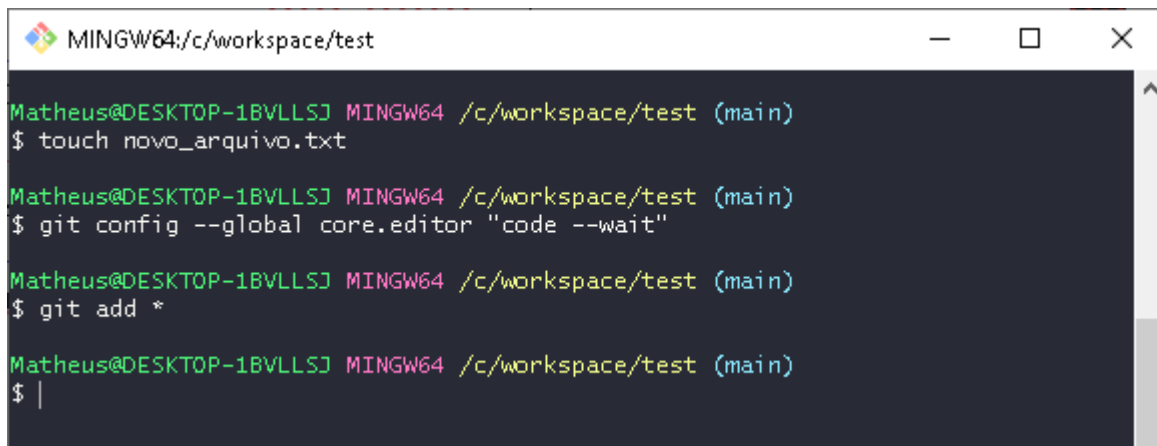


Perceba que toda issue tem um número de identificação que vem ao lado do título da issue na primeira linha, test2 #2 ← esse #2 indica a identificação dessa issue e vamos precisar dessa identificação. Ehhh... mais uma coisa eu percebi agora kkkk, percebi que escrevi casadas na imagem a cima mas não foi intencional tá kkkk eu as respeito muito e espero que elas mantenham seu casamento kksaksaksak.

Voltando ao contexto do nosso livro, Antes de estruturarmos uma mensagem de commit para o padrão para facilitar nossa vida vamos usar como editor de texto padrão do git bash o visual code studio que é um programa gratuito e muito utilizado para codar. Disponível em vários sistemas operacionais como linux, mac e windows e agora temos também disponível para Raspbian que é o sistema operacional do Rasper berry. A instalação é tão simples quanto o git.

Com o git bash aberto digite o comando **Git config --global core.editor "code --wait"** esse comando definirá como padrão o editor de texto do visual code studio e para reverter isso é só usar o **Git config --global --unset core.editor** em seguida **Git config --global core.editor "vim"** e se seu editor de texto for outro é só colocar o nome dele com --wait em seguida como no exemplo do primeiro comando com terminação "code --wait". Agora vamos fazer um novo commit na nossa pasta local com o git bash aberto nela, crie um novo arquivo ou edite um arquivo existente como na imagem a

baixo, nesse caso eu criei um arquivo e em seguida definir o visual code como editor de texto e após isso vem o git add para adicionar os arquivos ao repositório. OBS: no momento da instalação você pode escolher seu editor de texto e se já tiver o visual code studio instalado você pode selecionar ele.



```
MINGW64:/c:/workspace/test

Matheus@DESKTOP-1BVLLSJ MINGW64 /c:/workspace/test (main)
$ touch novo_arquivo.txt

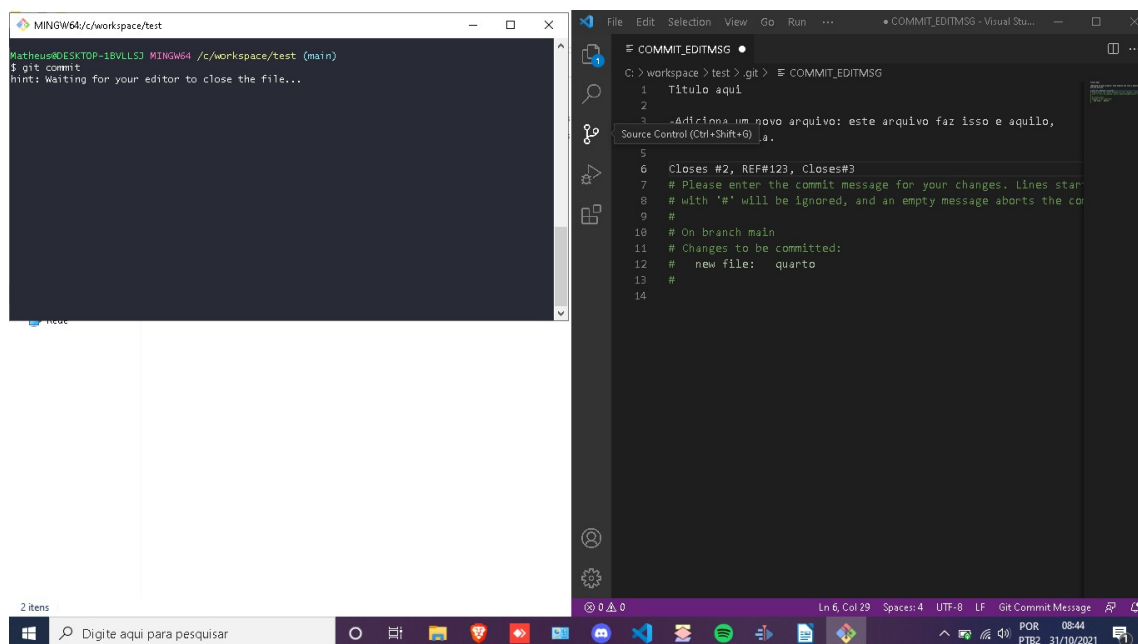
Matheus@DESKTOP-1BVLLSJ MINGW64 /c:/workspace/test (main)
$ git config --global core.editor "code --wait"

Matheus@DESKTOP-1BVLLSJ MINGW64 /c:/workspace/test (main)
$ git add *

Matheus@DESKTOP-1BVLLSJ MINGW64 /c:/workspace/test (main)
$ |
```

Agora vamos fazer um novo commit e dessa vez estruturar a mensagem de commit nos padrões devidos padrões, porém antes de querer padronizar tudo desse jeito caso você trabalhe em uma empresa use como base os padrões de mensagem de commit da empresa pois nem sempre o meu padrão é o mesmo padrão da empresa onde eu trabalho e isso pode causar conflitos mas supondo que todas empresas usam o mesmo padrão para mensagens de commit que eu vamos lá!

Apos isso é só usar o comando `git commit` sem a flag `-m` e sem “mensagem”, como na imagem abaixo e o git bash abrirá o editor visual code para você editar sua mensagem de texto nele.



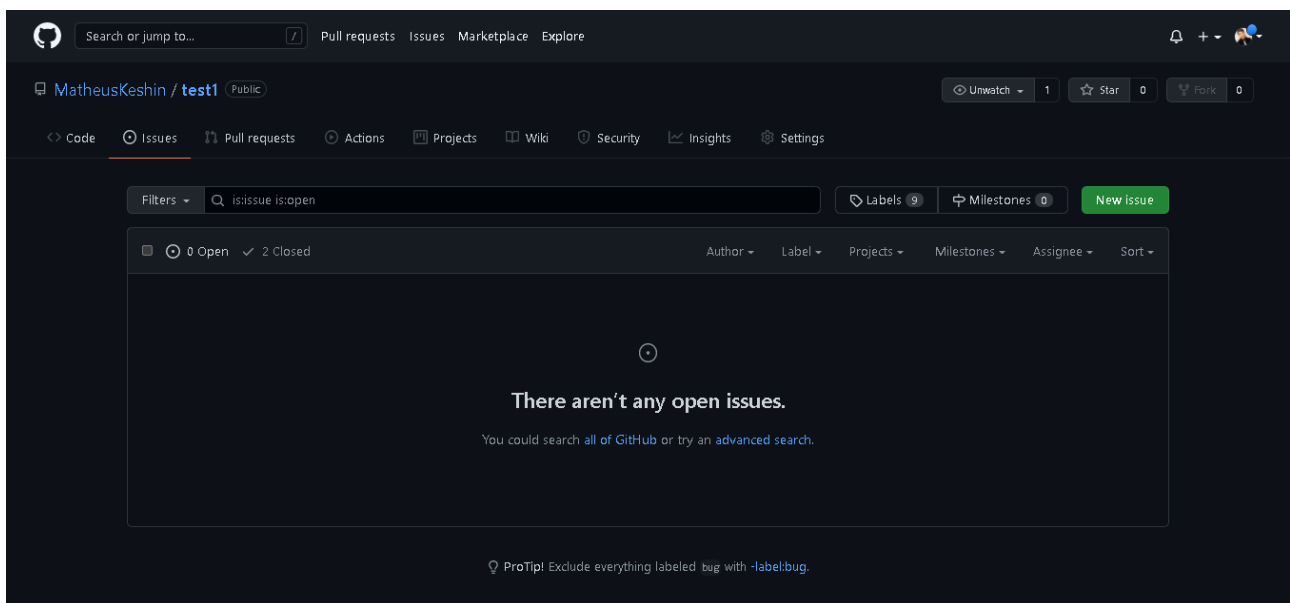
Como podemos ver na imagem a cima temos o visual code já aberto com um modelo bem simples onde tem-se TITULO, Descrição do commit detalhadamente e no rodapé temos algumas marcações responsáveis por relacionar arquivos que tenham como referência 123 que é o REF#123 e temos também o Closes #2 e Closes #3 que fecha issues do github indicando que elas foram resolvidas nesse commit atual e o próximo passo é salvar com `ctrl+s` após isso fechar o visual code

```
MINGW64:/c/workspace/test

Matheus@DESKTOP-1BVLLSJ MINGW64 /c/workspace/test (main)
$ git commit
[main 9171845] Titulo aqui
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 quarto

Matheus@DESKTOP-1BVLLSJ MINGW64 /c/workspace/test (main)
$ |
```

Commit feito agora vamos empurrar para nosso repositório no github com o comando `git push` e após isso é só abrir o github e ver se a issue ainda tá aberta. Caso você queira manter suas mensagens de commit pequenas use o `git commit -m "mensagemcontexto, Closes #1"` deve fechar também uma issue aberta.



Veja que antes tinha 1 issue aberta e agora já fechamos ela.

## VERSIONAMENTO SEMÂNTICO

Sabe aqueles programas que tem a versão V1.0 ou V 3.2.7 convencionalmente usamos a seguinte nomenclatura, no segundo exemplo onde temos a versão 3.2.7, onde temos 7 → trata-se de um patch que são correções de bug e pequenas alterações e a medida que as correções e alterações pequenas vão sendo feitas esse número vai crescendo até que a segunda casa que é o 2 vira número 3, 2 → trata-se de alterações feitas que não há quebra de compatibilidade entre versões e esse número assim como os patch vão crescendo de acordo com as alterações feitas até que acrescenta +1 a primeira casa que é o 3 e onde temos 3 → trata-se de um programa que esse número mudará a cada quebra de compatibilidade com sua versão anterior ou versões extremamente diferentes de sua versão antiga ou com um número considerável de alterações vira uma nova versão então o a v3.2.7 vira uma versão 4 e não mais 3.

Então isso um patch conta desde 0 até 9 e quando chega em 9 a casa 2 (0.x.0 onde tem-se o x) vira 1, quando uma alteração menor vai de 0 a 9(x.0.0) acrescenta em 1 a versão major que é o primeiro dos três números.



E é isso, esse foi o modelo de versionamento semântico mas lembre-se de que existem outros tipos de versionamento e as empresas escolhem quais tipo adotar e geralmente tem softwares complementares que ajudam a versionar o código automaticamente.

## GIT E GIT HUB FOCADO EM PULL REQUEST

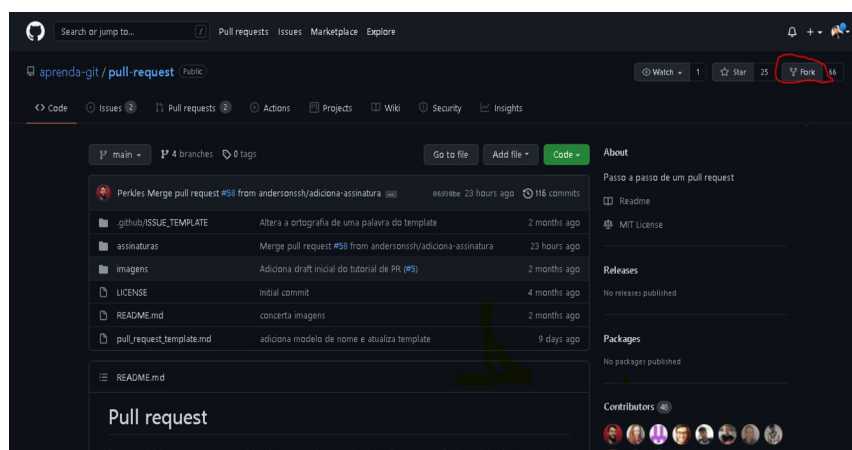
Imagine que você achou um repositório de um software livre no github e que você quer contribuir para aquele software mas não tem permissão para fazer comandos de push ou pull, você apenas pode clonar pelo git bash porém você não quer só clonar você quer ajudar o criador do repositório a melhorar o software. Para isso faremos pull requests que são pedidos de pull, ou seja, fazemos o fork do repositório que queremos transformamos esse repositório em “nosso” mas sendo ele um repositório que aponta para o repositório de seu criador original, então após fazermos as alterações que queremos para contribuir com o software.

O criador do repo original vai então avaliar se sua colaboração é valida e se for vai ser mesclado ao repositório dele, se precisar de alterações ele irá apontá-las e caso não seja valida ele negará o pull request. Tudo isso é feito no próprio github.

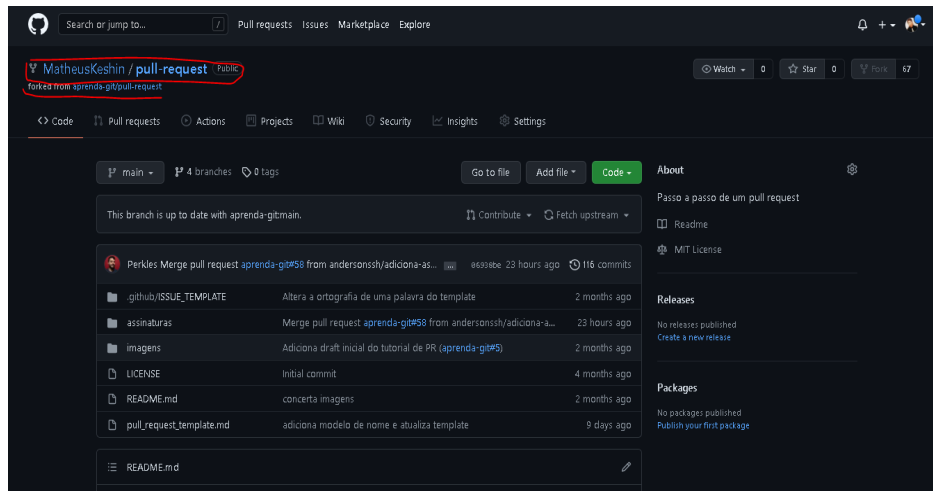
## FAZENDO O FORK DE UM REPO

Dentro de um repositório que você deseja contribuir faça o seguinte na pasta raiz:

1- Clique em FORK como na imagem a baixo circulado em vermelho.



2- Observe na imagem a baixo, agora que você fez o fork de um repositório esse repositório passa a ser “seu” ou seja você tem um clone de um repositório que não pertence originalmente a você. Perceba que abaixo do item circula indicando que esse repositório te pertence tem escrito: “forked from aprenda-git/pull-request”, isso indica que esse repositório clonado se originou do repositório aprenda-git/pull-request.



3- agora que temos o clone desse repo no nosso perfil do github vamos fazer o clone dele para nosso workspace! Clique em **code**, escolha ssh ou https e copie a url em seguida abra o git dentro do seu workspace e digite o comando **git clone coleurlaquishouhttps** e pronto você agora tem um repositório que foi forkado para seu perfil do github e clonado para sua área de trabalho, agora você já pode trabalhar nele.

4- para começar a mexer e adicionar novas implementações a este repositório antes de tudo crie uma branch paralela onde todo o seu trabalho será feito isoladamente da branch main, assim deixando mais fácil para o desenvolvedor dono do repositório original para verificar e fazer testes nos arquivos editados e criados por você.

5- Após todas as alterações você precisa usar o comando dentro do git **git push origin nomedabranhparalela** com isso você fará o push para o seu repositório da branch na qual você trabalhou em seguida aparecerá uma mensagem com um link para abrir um pull request que pede permissão de mescla do seu código para o código origem no caso, pergunta para o dono original do código se o meu código pode ser integrado ao dele, se sim assim que recebermos uma resposta positiva vamos ser notificados no e-mail e no próprio github, caso o dono do código peça para alterarmos algo também seremos notificados assim como também seremos notificados caso a resposta seja negativa. Observe a imagem a baixo.

```
MINGW64:/c:/workspace/pull-request/assinaturas

Matheus@DESKTOP-1BVLLSJ MINGW64 /c:/workspace/pull-request/assinaturas (main)
$ git checkout -b novabranhmk
Switched to a new branch 'novabranhmk'

Matheus@DESKTOP-1BVLLSJ MINGW64 /c:/workspace/pull-request/assinaturas (novabranhmk)
$ git push origin main novabranhmk
Enter passphrase for key '/c:/Users/Matheus/.ssh/id_ed25519':
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'novabranhmk' on GitHub by visiting:
remote:   https://github.com/MatheusKeshin/pull-request/pull/new/novabranhmk
remote:
To github.com:MatheusKeshin/pull-request.git
 * [new branch]      novabranhmk -> novabranhmk

Matheus@DESKTOP-1BVLLSJ MINGW64 /c:/workspace/pull-request/assinaturas (novabranhmk)
$ git push origin main novabranhmk AC

Matheus@DESKTOP-1BVLLSJ MINGW64 /c:/workspace/pull-request/assinaturas (novabranhmk)
$
```

Ainda sobre pull request, eu posso revisar, pedir alterações, aceitar mudanças e negar também como mencionado a cima dentro do github, clicando dentro de um repositório em:

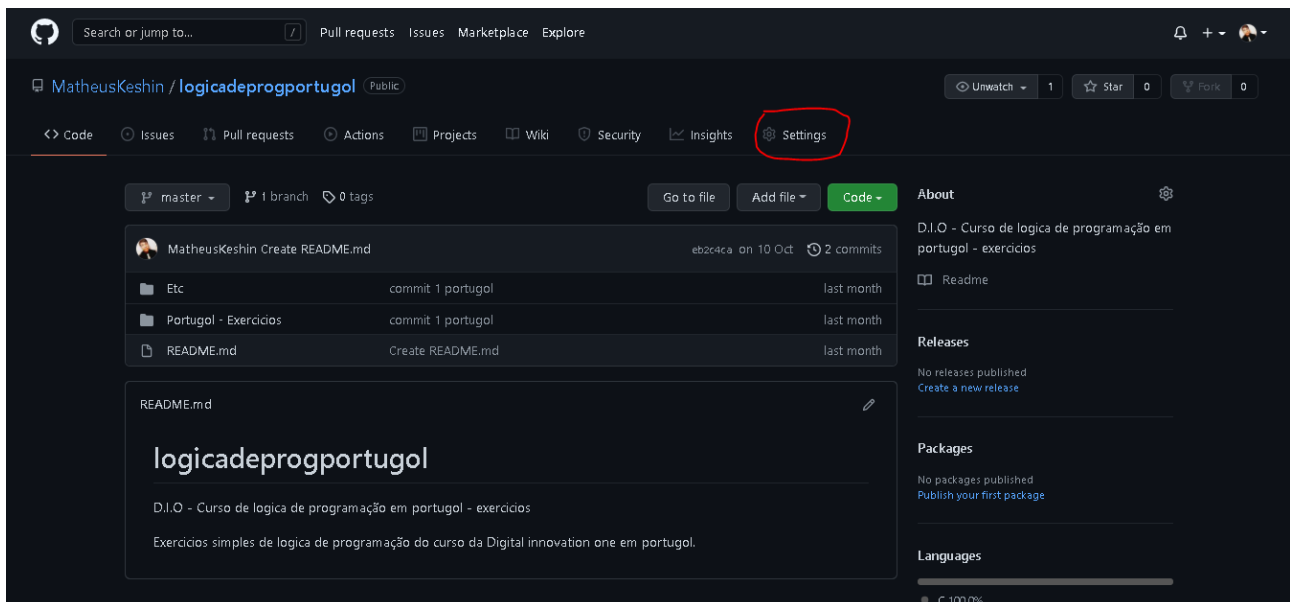
- 1- pull requests – open
- 2- commit, adiciona assinatura nomedapessoa
- 3- clica na linha, escreve o que deve mudar
- 4- start a review

## PERMISSÕES PARA MANUSEAR REPOSITÓRIOS NO GITHUB

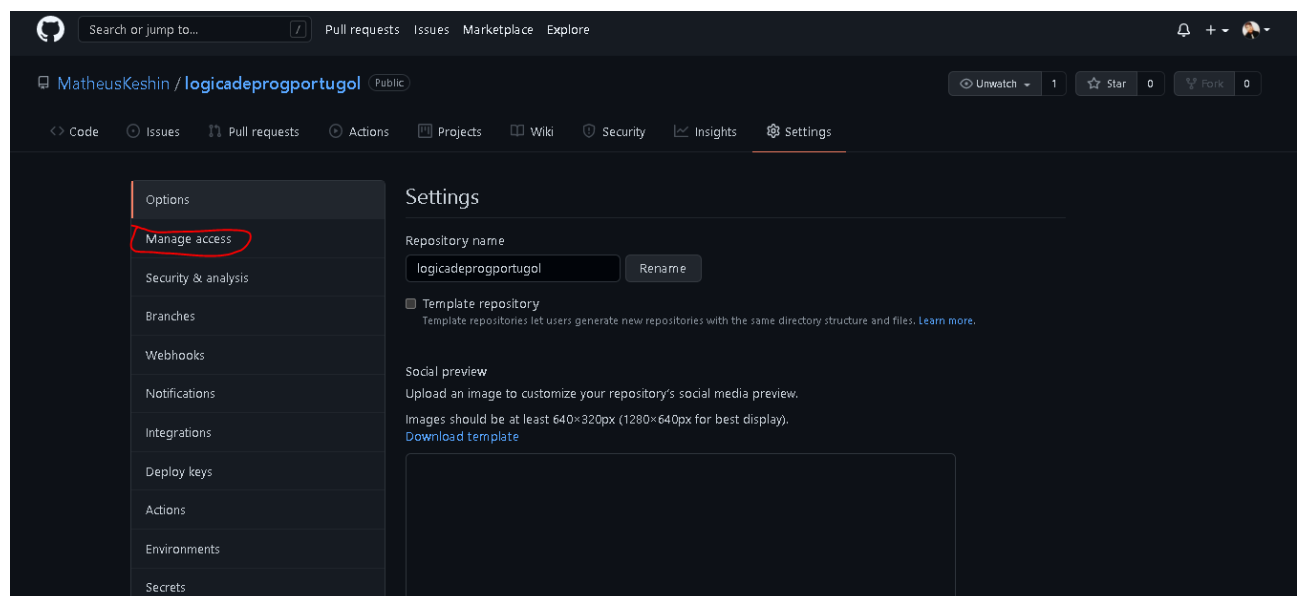
Você pode também dar permissões para usuarios especificos assim não são necessárias requisições, pull request.

Agora como posso dar permissões para que outras pessoas manuseiem meu repositorio no github ?

- 1- Entre no seu repositorio e click em settings

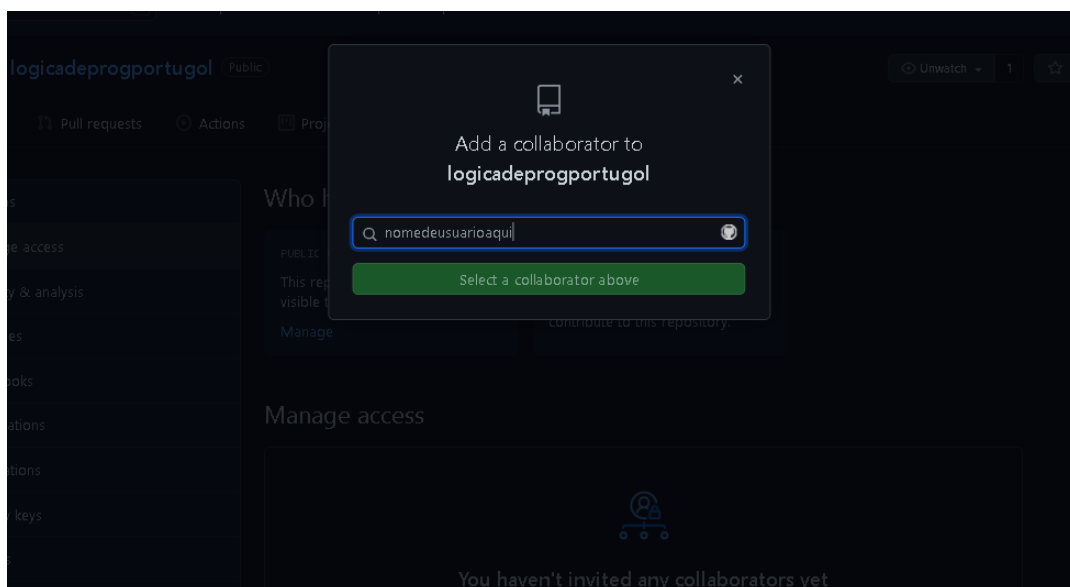
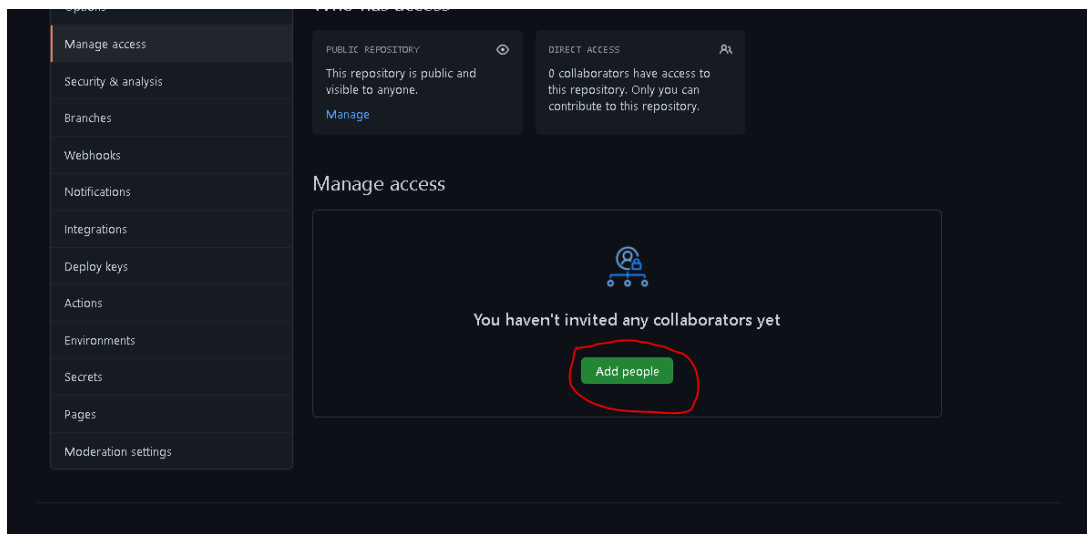


- 2- Click em Manage Access





3- Após confirmar sua senha, adicione os usuários pelo nome de usuário do github.

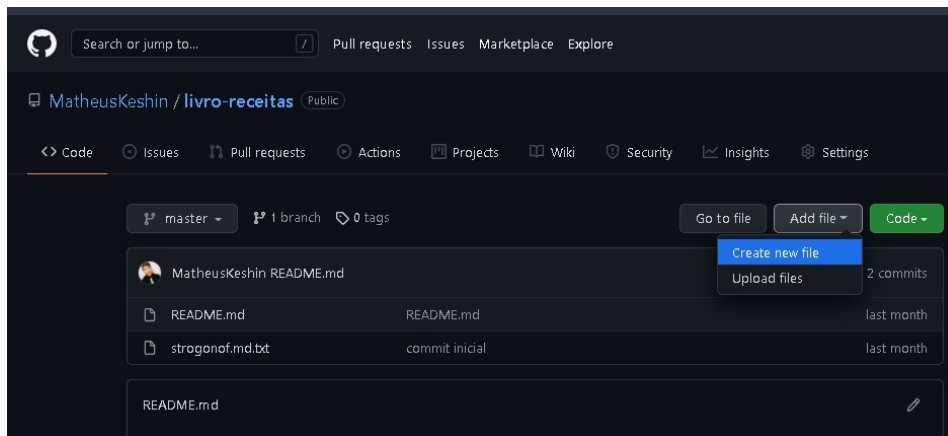


## ORGANIZAÇÕES E TIMES NO GITHUB

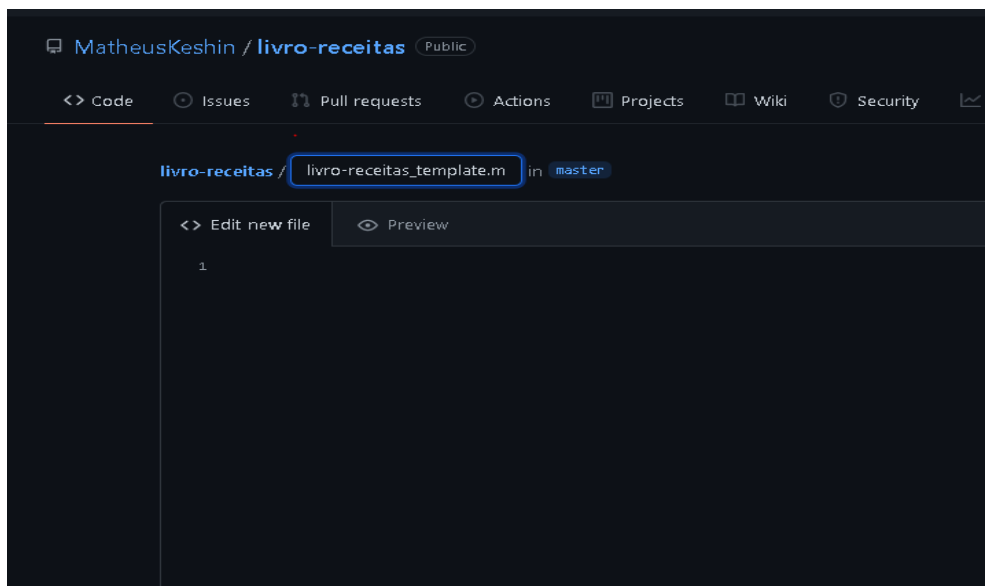
Essa ferramenta é criada no github assim criando uma profile organization que pode ser organizada por espaços e pode ter criação e limitações para times. Não irei retratá-la a fundo pois não tenho acesso a essa ferramenta, mas basicamente são comunidades criadas com repositórios separados por tipos, ou seja, imagine uma empresa de software que criou a comunidade x e dentro dessa comunidade temos repositórios para os developers, Q.A e pessoas responsáveis por fixar os bugs ou seja dentro dessa “comunidade” de uma empresa temos 3 repositórios diferentes que podemos limitá-los a apenas trabalhar no seu repo ou poder visualizar outros repos e etc.

## CRIANDO MODELO DE PULL REQUEST (FORM)

1- Dentro da pasta raiz do projeto no github clique no botão add file → create a new file



2- Crie um arquivo com nome idêntico ao repositório! Depois do nome de usuario vem o nome do repo nesse caso é: livro-receitas e nosso arquivo criado vai ser livro-receitas\_template.md

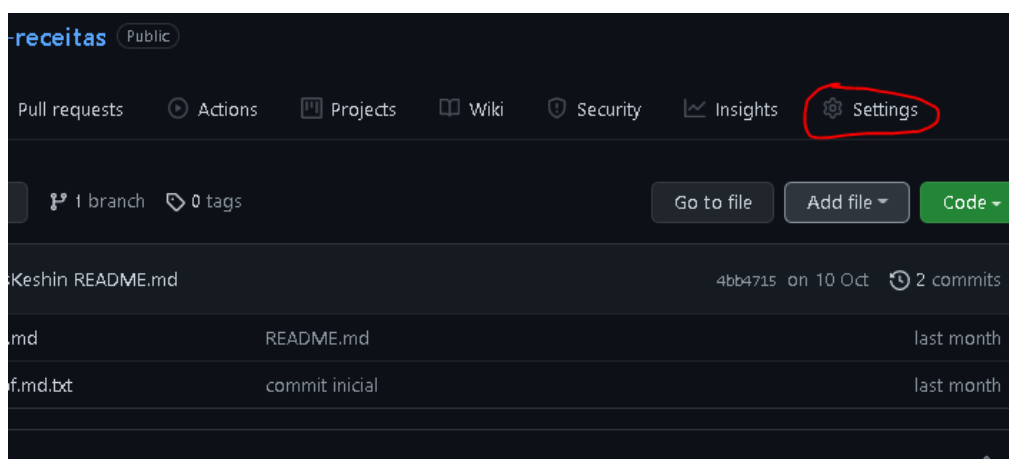


3- Faça a escrita de instruções, informações e requisições nesse template e a partir dai toda vez que alguém fazer o pull request vai visualizar essas informações

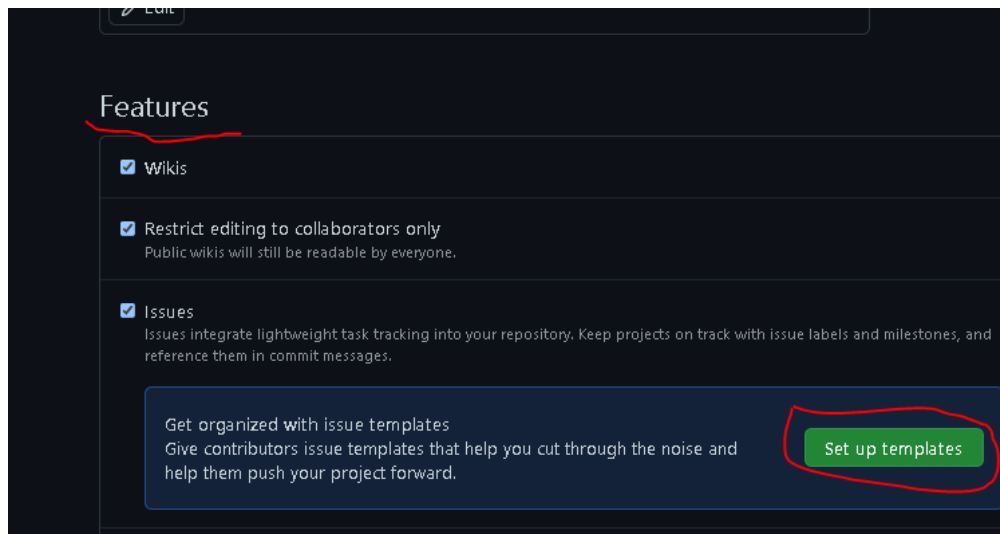
OBS: Utilize do markdown para “enfeitar” o template esse template quem cria é o dono do repositório original.

## CRIANDO TEMPLATE PARA ISSUES

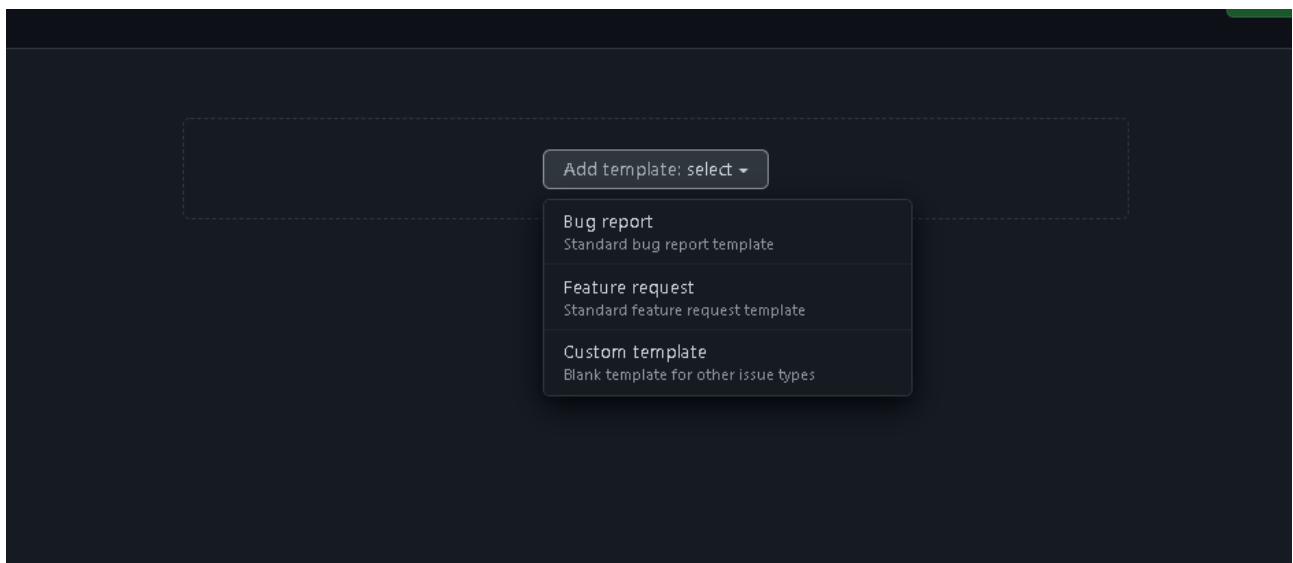
1- O proprietario original do repo irá na pasta raiz do repo, e clicar em settings

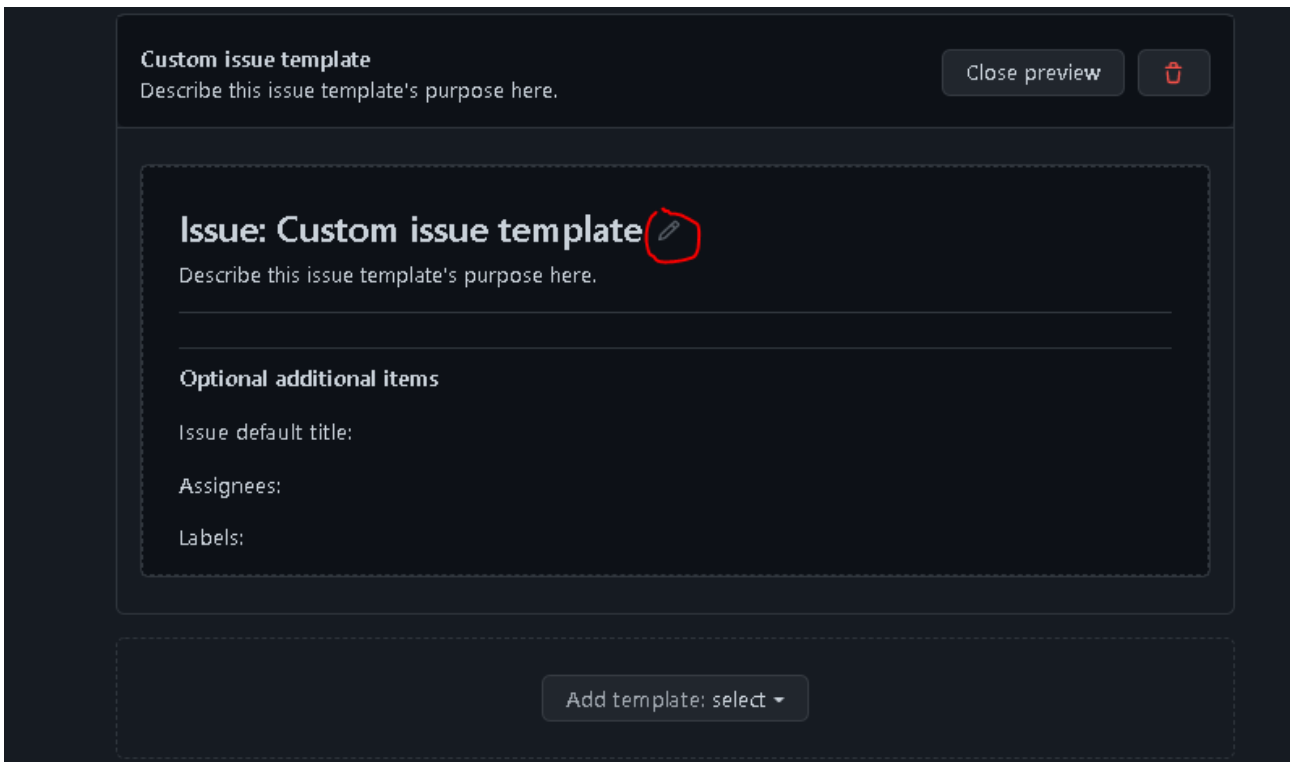
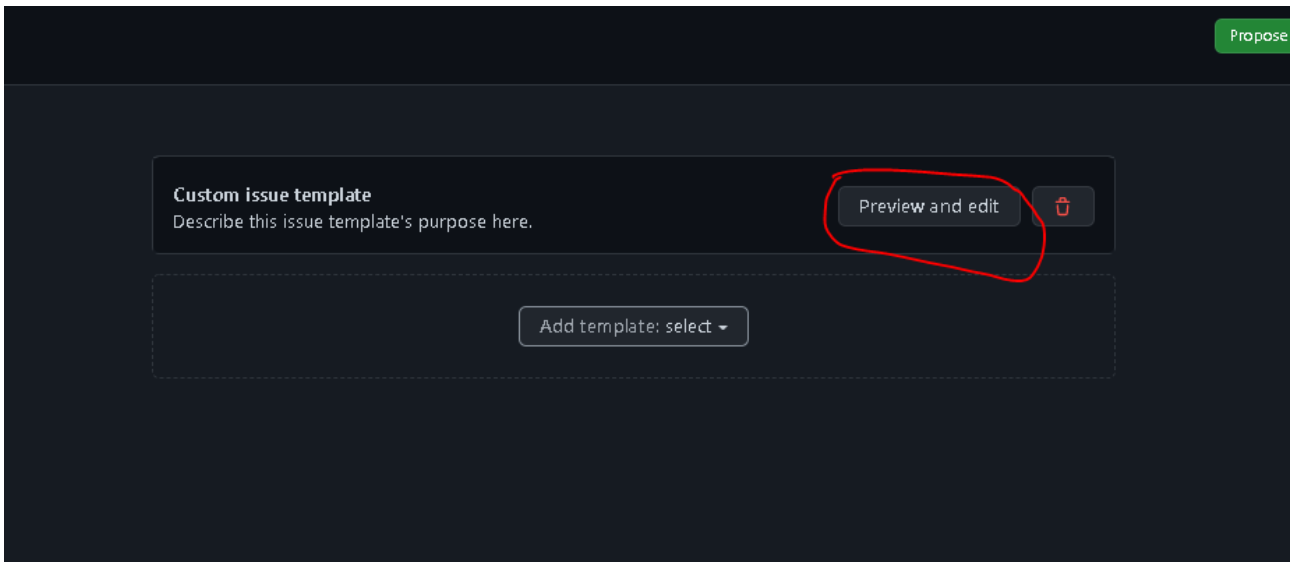


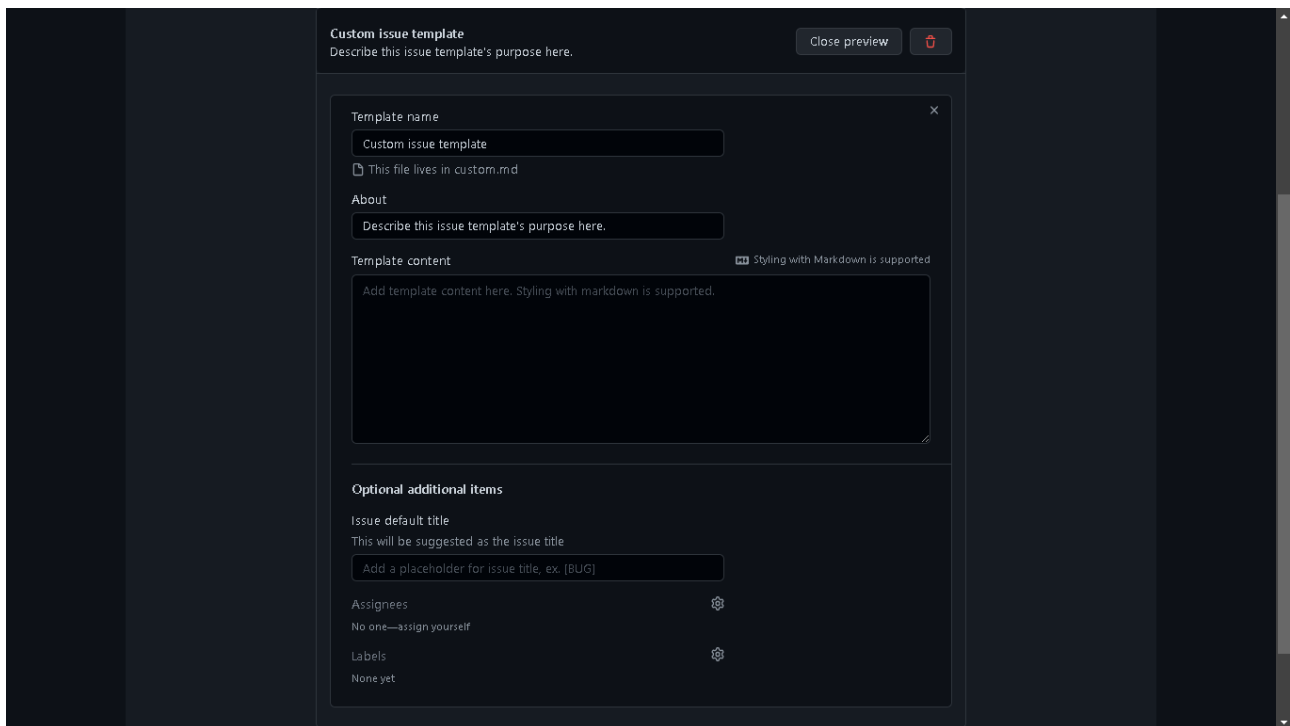
2- Rolar a página até achar Features - [ Issues – clicando set up templates



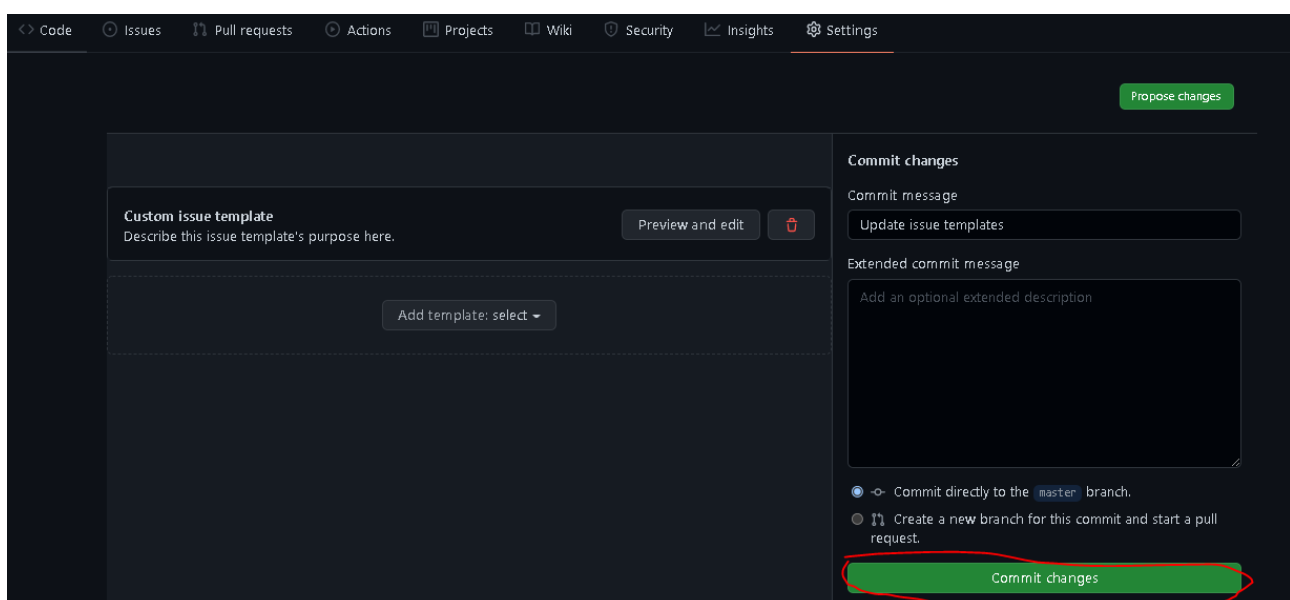
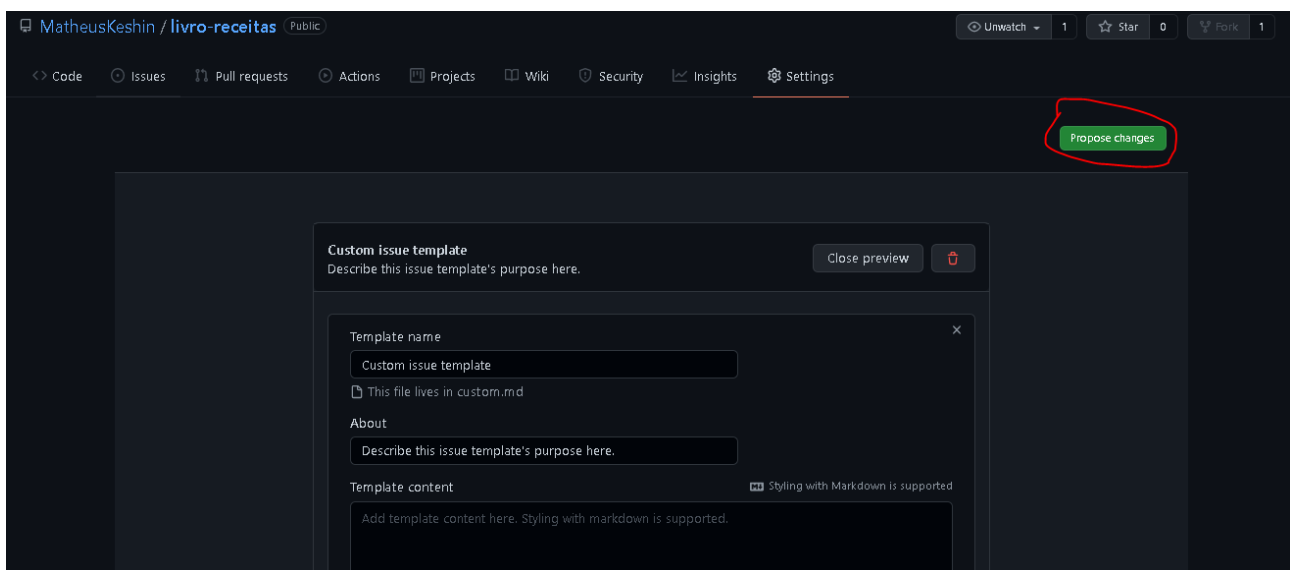
3- ADD Template, após isso escolha um tipo de issue e edite-a clicando em PREVIEW AND EDIT e clica no lapis para editar o template de issue.







4- clica em propose changes e clica em commit changes após preencher os campos devidamente



## ALIASES (APELIDOS)

Os apelidos são formas de se ganhar mais agilidade pois com eles, por exemplo, você pode trocar branch por apenas a letra B. Então o `git branch` irá tornar-se apenas `git b`

Para definir apelidos de comandos utilizamos o seguinte código dentro do git bash

```
git config --global alias.b branch
```

agora com esse comando funcionando não iremos mais usar o `git branch` e sim o `git b` então resumidamente, `git config --global alias.nomedonovocomando comandoantigo`.

E para desfazer os apelidos criados utilizamos:

```
git config --global --list → para visualizar os comandos globais criados
```

`git config --global --unset alias.nomedoapelido` → para desfazer os apelidos, no nosso caso definimos branch como “b” aí ficaria `git config --global --unset alias.b` e agora nós voltamos a usar `git branch`.

## INFORMAÇÕES EXTRA

Git reset vs git revert: O revert é projetado para desfazer de forma segura um commit público, eu reset é projetado para desfazer alterações locais.

## LISTA DE COMANDOS UTILIZADOS AO LONGO DO LIVRO

### Navegação por diretórios e utilitários.

**Windows**  
`cd e cd..`

**Linux**  
`cd e cd..`

`cd nomediretorio` entra e `cd..` volta ao anterior.

`dir`

`ls`

lista pastas e arquivos de um diretório

`ls -a`

lista itens ocultos

`mkdir`

`mkdir`

cria diretórios

`del` ou `rmdir nomearq /s /q`

`rm -rf nomedirouarq`

apaga arquivos e diretórios, a flag `-rf` indica que `r` é algo recursivo (deleta todas as pastas dentro de pastas) e `f` força a ação de deletar o arquivo para o sistema não perguntar nada.

`cls`

`clear` ou `ctrl+l`

limpa mensagens da linha de comando

`cd /`

`cd /`

entra no diretório raiz do sistema

`echo mensagem`

`echo mensagem`

escreve na linha de comando a mensagem pedida

`echo mensagem > nomearq.extensão` → serve para criar um arquivo com um nome escolhido com uma extensão também escolhida e a mensagem vai estar dentro desse arquivo ao abrir, por exemplo: `echo hello my brother > hello.txt`

`cat arquivo.txt` → abre um arquivo na linha de comando e mostra o que tem escrito nele.

`cd` `pwd` mostra onde me encontro na linha de comando.

`mv arquivo.txt ./outrapasta` → move o arquivo.txt para uma pasta chamada ./outrapasta (funciona no git bash)

`touch nomearquivo.extensão` → cria um arquivo novo.

Obs.: use o TAB para autocompletar comandos e nomes de diretórios.

### Chaves SSH

`ssh-keygen -t ed25519 -C "email@gmail.com"` → comando para gerar chave ssh

`eval #(SSH-AGENT -S)` → ativa o agente gerenciador da chave

### Primeiros comandos com o git (gitbash)

`git config --global --list` → lista configurações globais do git setadas.

`git config --global user.email "email@email.com"` → vinculado um e-mail aos commits

`git config --global --unset user.email` → desvincula um e-mail adicionado ao git

`git config --global user.name "namegithub"` → vincula um nome de usuário aos commits

`git config --global --unset user.name` → desvincula um nome de usuário adicionado ao git.

`git config --global core.editor "code --wait"` → configura o editor de texto padrão do git sendo o visual code studio, caso queira voltar ao padrão que é o vim use o código abaixo depois use esse mesmo código e substitua o "code --wait" por "vim".

`git config --global --unset core.editor` → desvincula um editor de texto padrão.

`git init` → inicializa um repositório git dentro de uma pasta

`git add *` → adiciona todos os arquivos novos ou modificados a área de espera para serem commitados.

`git commit -m "mensagem contexto"` → efetiva um commit e cria uma "foto" do momento onde todos os arquivos até o momento existem.

`git status` → mostra o estado do repositório, se falta adicionar algum arquivo ou commitar algo.

`git remote add origin URL` → adiciona um link ao qual será enviado o repositório local para a nuvem ou repositório online(github).

**git remote -v** → mostra os repositórios vinculados em um repositório.

**git push origin master/main** → após vinculado um repositório local utilizamos o push para fazer o envio para o github, se seu repositório for master coloque master e se for main coloque main.

**git clone URL** → clona um repositório por ssh ou https.

**git pull origin master** → puxa um repositório mais atualizado do github para seu repositório local assim atualizando o repositório que antes já existia mas estava desatualizado.

### **Trabalhando com branches**

**git branch** → mostra as branches existentes dentro de um repositório.

**git checkout nova-branch** → comando para se movimentar entre as branches, nesse caso estou saindo de uma branch qualquer e entrando na branch nova-branch.

**git checkout -b br4nch** → a diferença desse comando para o anterior é a flag “-b” que cria uma nova branch chamada “br4nch” e após cria-la entra nessa nova branch.

**git merge branchx** → faz a mescla de duas branches onde branchx é o nome da branch que será fundida a branch atual no qual nos encontramos.

**git branch -m nomeantigo novonome** → renomeia uma branch

**git branch -d nomebranch** → deleta uma branch

### **STASH**

**git stash save “mensagem contexto”** → cria uma stash, só podemos criar uma stash após usar o git add \*.

**git stash list** → mostra todas as stashes criadas na branch onde estamos.

**git stash pop number** → recupera os dados salvos em uma stash onde number é igual ao id da stash.

**git stash clear** → limpa as stashes.

### **Visualizando históricos de um repositório.**

**git log** → mostra com detalhes um histórico de commits e outros itens sobre o repositório local.

**git log --oneline** → mostra um resumo do histórico do repositório local.

**git log --graph** → saída de dados um pouco mais ilustrada graficamente.

**gitk** → dados são abertos em uma nova janela onde são muito bem ilustrados graficamente, Obs.: esse comando funciona apenas para o software instalado em windows.

### **Reset vs Revert**



`git reset hashdocommit` ou `git reset head~1` → reseta para um commit com um hash específico ou usando `head~1` reseta um commit para trás e esse comando por não especificar o tipo de reset ele usa como default o mixed.

`git reset --soft head~1` → esse comando não exclui nem modifica um documento porém move os itens já commitados para o estado de stage area ou seja esperando para serem commitados, é como se eu desse `git init`, `git add *` apenas e ficaria faltando o `git commit` para finalizar.

`git reset --mixed head~1` → esse comando revert os comits para a area unstaged/untracked, antes do `git add *`. é como se eu tivesse dado `git init` criado arquivos e não usasse o comando `git add`.

`git reset --hard head~1` → esse comando deve ser usado com sabedoria pois ele é destrutivo e pode apagar arquivos importantes de um commit, nesse caso deleta o ultimo commit feito como se ele não tivesse existido.

Obs: o `head~1` pode ser `head~2,3,4` e etc isso indica em ordem decrescente qual a qual commit será resetado ou revertido algo mas caso queira voltar a um commit específico use o código hash desse commit no lugar do `head~1`.

`git revert head~1` ou `git revert hashdocommit` → não precisa de flags e pode ser usado com `head~1` ou com código hash de um commit anterior, diferente do reset esse commando é utilizado quando queremos reverter algo sem deletar o commit pois esse commit pode já estar em uso no github e deleta-lo significa que não poderemos mais fazer envios de arquivos pois nossos arquivos vão estar desatualizados, então para desfazer algo que já foi colocado no github sem afetar o momento de envio usamos o revert.

### **Pull requests**

`git push origin nomedabbranch` → com esse comando você faz o push da branch na qual você estava trabalhando para fazer o pull request.

### **Aliases(apelidos)**

`git config --global alias.x` comando → com esse comando podemos apelidar um comando específico como x então quando for usar podemos usar `git x` ao invés de `git comando`.

**ATENÇÃO:** ESSE LIVRO É UM CONTEÚDO EXTRA PARA ALUNOS DA MK ACADEMY, ASSISTA TAMBÉM AOS VÍDEOS PARA TER UMA MELHOR VISUALIZAÇÃO E COMPREENSÃO! ACESSE O LINK: [encurtador.com.br/pqzCJ](https://encurtador.com.br/pqzCJ)