**GIT E GITHUB**

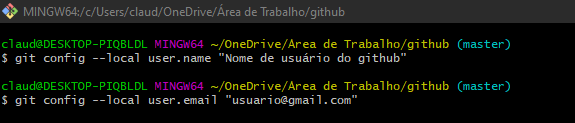
**GITHUB** é a aplicação na Web [www.github.com](http://www.github.com) aonde é possível criar sua conta e visualizar as alterações feitas no GIT.

**Git ou GitBash** é um (CMD)controlador de versionamento, serve para salvar diversas versões de arquivos. O arquivo pode ser acessado pelo usuário aonde vai localizar todas as versões de arquivos salvas por ele, com isso o usuário pode voltar a primeira, antepenúltima, penúltima ou “n” versão utilizada se desejar. Outra opção é que o arquivo pode ser usado em conjunto com outras pessoas, aonde os usuários que alterarem esse mesmo arquivo vão receber alertas de mudanças ao tentar salvar o arquivo no local “servidor” aonde salva todas as versões desses arquivos.

**Comandos básicos com GIT: (Identificar versão “git - -version”)**

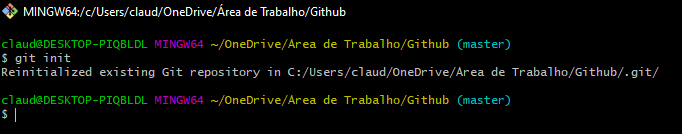
**CONFIGURANDO USUÁRIO E EMAIL “LOCAL” OU “GLOBAL”**

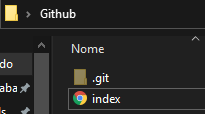
Configurar e atrelar o seu usuário e e-mail ao git instalado localmente, assim ele passará a carregar as modificações de arquivos para a sua conta informada. Para utilizar global basta trocar –local por –global



**INICIANDO O GIT EM UM REPOSITÓRIO**

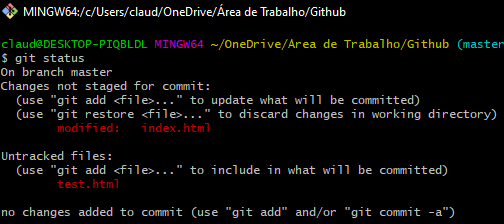
Vai inicializar o seu repositório e vai criar um subdiretório (.git) dentro do seu repositório, o processo ainda não é monitorado, mas o GIT já entende que aquele repositório pertence a ele e que vai ser utilizado.

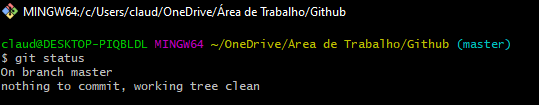




**CONSULTANDO E IDENTIFICANDO STATUS (HISTÓRICOS) DOS REPOSITÓRIOS NO GIT**

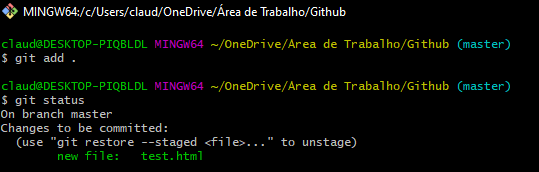
Dentro do seu diretório existem arquivos que nunca foram modificados (não-rastreados Untracked Files **ex:test.html**) e estão disponíveis para serem adicionados ao stage (arquivos que o GIT “vai conhecer”) pela primeira vez. E existem também arquivos que já foram adicionados ao stage alguma vez e já foram modificados (**ex:index.html**). A segunda imagem identifica que não existe nenhum arquivo para ser enviado ao stage, porque provavelmente todos os arquivos já foram adicionados.





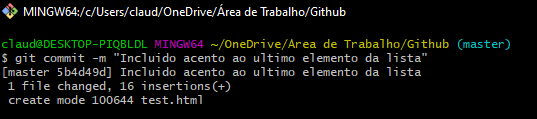
**RASTREANDO ARQUIVOS**

Utilizando o “git add .” você passa a rastrear todos os arquivos dentro do seu repositório, para rastrear um diretório específico que possa conter outros subdiretórios utilizar o “git add ./Nomedodiretório” ou arquivo “git add index.html”. Quando o arquivo passa a ser rastreado ele fica com a mensagem de “New File” no comando git status.



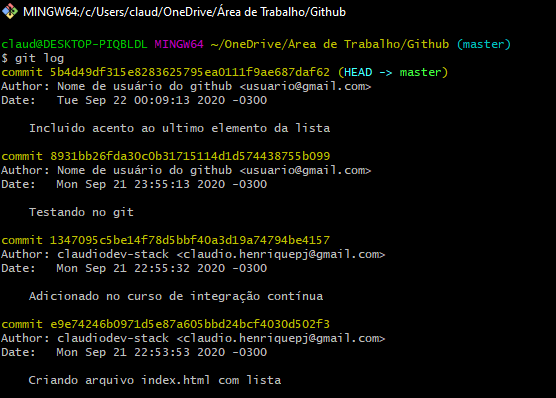
**COMMITANDO ARQUIVOS**

Após o arquivo ser rastreado ele está preparado para ser salvo.   
O “-m” é para enviar uma mensagem no arquivo que você está fazendo, com isso vai informar que o arquivo foi incluído com sucesso.



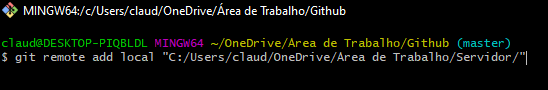
**LOG DE ALTERAÇÕES DE ARQUIVOS**

Visualiza todas as alterações feitas dizendo o Usuário, Data etc. E cada commit é gerado um HASH code e os comentários de cada commit. Existem também comandos como “git log -p” que detalha as mudanças internas do arquivo e “git log –oneline” que detalha as descrições inseridas nos arquivos, “git log –graph” gera um gráfico de branches. Existem também consultas de log avançadas no link [https://devhints.io/git-log /](https://devhints.io/git-log%20/) Para sair do git log basta utiliza a tecla “Q” de quit



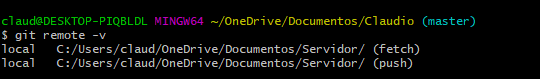
**TRABALHANDO DE FORMA REMOTA COM OS REPOSITÓRIOS**

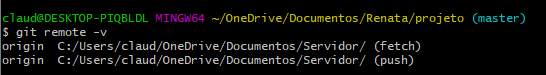
Para adicionar repositórios remotos como por exemplo (O repositório do usuário2 ou servidor) você deve adicionar e setar o caminho desse repositório, pode ser por pasta, url, geralmente url github. PS: É necessário utilizar as “” quando o comando existe algum espaço como por exemplo /Area de Trabalho/ se o caminho não tiver espaços não é obrigatório.



**IDENTIFICANDO REPOSITÓRIOS REMOTOS VINCULADOS**

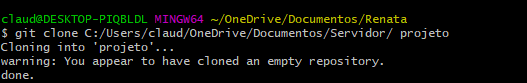
Para identificar qual repositório remoto foi adicionado a sua... digamos “conta” ex: Claudio/Renata basta usar o comando abaixo, veja que está pronto para “buscar” e “empurrar” no repositório Servidor.





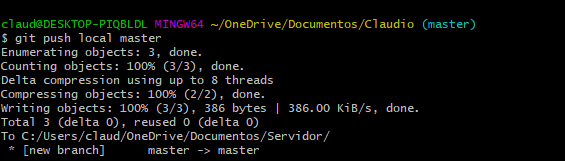
**CLONANDO UM REPOSITÓRIO REMOTO**

Quando temos um repositório já criado por outro usuário e sendo utilizado, basta clonar do caminho correto do repositório remoto dele para o seu diretório, assim todas as mudanças feitas serão clonadas. PS: Se o outro usuário não subiu as mudanças para o servidor por exemplo, talvez ele retorne com os diretórios vazios.



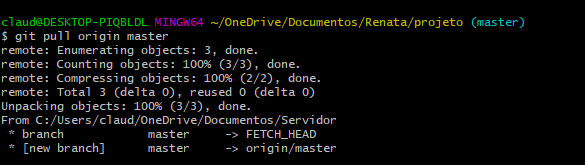
**DANDO PUSH DE UM REPOSITÓRIO**

No exemplo abaixo, estou empurrando ou enviando as mudanças feitas nos arquivos para o repositório remoto “servidor”. Com isso é possível um outro usuário pegar as mudanças desse “servidor”, sem necessitar ser diretamente do meu local remoto.



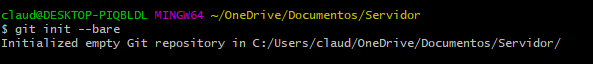
**DANDO PULL DE UM REPOSITÓRIO**

O comando abaixo busca as mudanças ou atualizações feitas nos arquivos e carrega como sendo o arquivo atual. É necessário identificar o remoto que você está dando pull

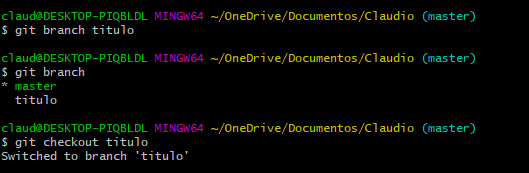


**INICIANDO COM --BARE**

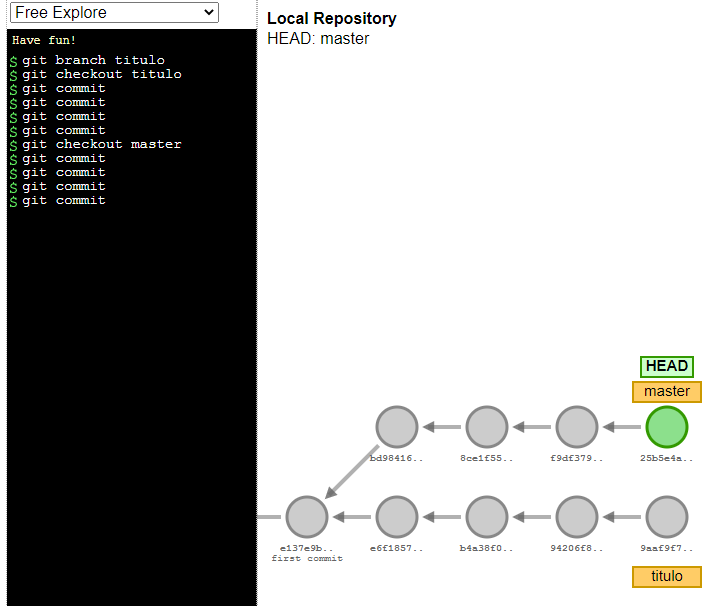
Com o comando git init --bare você está criando um repositório que é *pushable*. Geralmente os repositórios bare são criados no servidor e são considerados repositórios para armazenamento, em contraste aos repositórios que vão nas máquinas dos desenvolvedores que seriam os repositórios de desenvolvimento, criados com o comando git init (sem o --bare).  
Apesar do GIT ser um sistema de controle de versionamento distribuído, é muito comum que exista um repositório central que facilite a troca de informações entre os desenvolvedores, evitando a necessidade que os computadores dos desenvolvedores se comuniquem diretamente entre si.



**BRANCHES**

<http://git-school.github.io/visualizing-git/>  
A ideia das branches (ramos) em resumo, são criar diretrizes diferentes para os usuários não utilizarem as mesmas branches, porque com isso as alterações feitas por ambos não hora de buscar as atualizações não criem conflitos umas com as outras. O link acima explica a utilização de branches de forma visual. Para criar uma nova branch basta utilizar o primeiro comando, o segundo comando para consultar as branches criadas e o terceiro comando para trocar para a branch que quer usar. A diferença entre o “-d” e “-D” é se tiver algum commit o -D força o delete, senão o -d funciona.

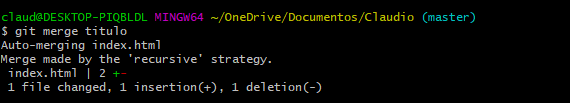




**UNINDO BRANCHES**

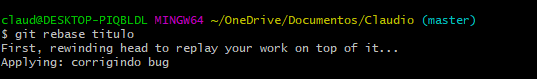
Após realizar as mudanças dos arquivos na branch (título) identifiquei uma outra pequena falha no meu arquivo da branch (master), portanto eu corrigi ele e após isso com as duas branches corretas, eu fiz um merge ou seja a junção dos conteúdos e trocas nas duas branches para que tivesse um conteúdo único e final. Para sair “:” e “x”

<https://www.treinaweb.com.br/blog/git-merge-e-git-rebase-quando-usa-los/>  
Link sobre funciona o Merge e o Rebase



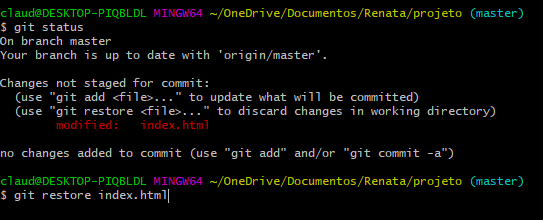
**ATUALIZANDO BRANCHES**

O rebase faz é produzir um histórico linear, mais limpo e mais fácil de ser lido, pois os branches são literalmente fundidos. Pela fusão, também não é gerado aquele commit adicional “estranho” que acontece no merge.

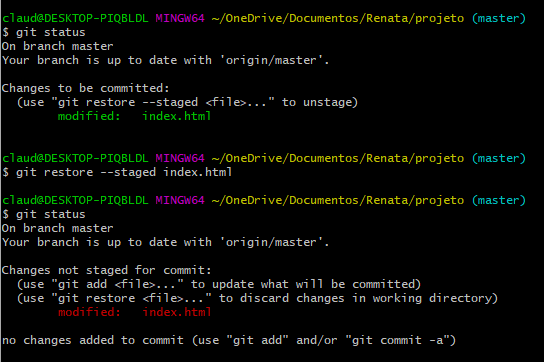


**DESFAZENDO MUDANÇAS DE ESTADOS NO GIT (CTRL Z)**

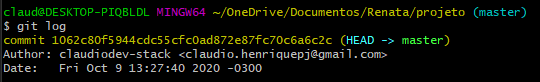
Caso eu tenha feito alguma alteração **SOMENTE** no arquivo e salvado, quando visualizo o status dele existe a opção de restaurar o arquivo para a versão anterior com o comando restore.

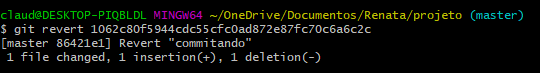


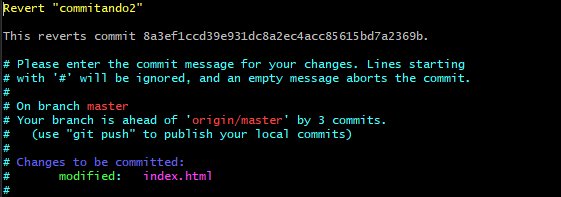
Caso eu já tenha salvado esse arquivo e depois dado git add no arquivo, será necessário utilizar o staged para voltar ao estado anterior do arquivo somente salvo sem ser adicionado.

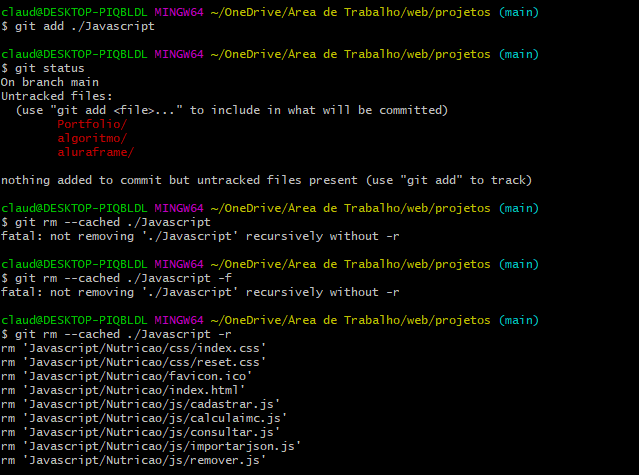


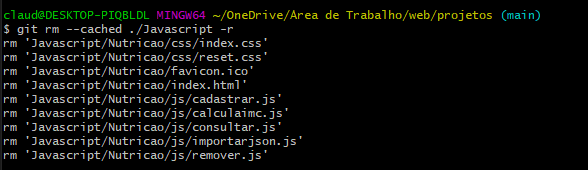
Caso eu já tenha salvo o arquivo, adicionado e dado o commit, para desfazer a operação após o commit será necessário pegar o HASH Code (ou só os 7 primeiros caracteres) no git log e utilizar o revert utilizando o hash code. Para sair da tela “:” “x”





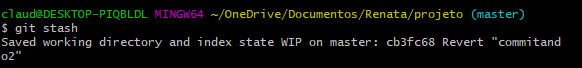


As vezes você pode dar um git add e depois git status para saber se o arquivo foi rastreado e ele não exibir, o comando de rm – cached remove a ação no arquivo e volta a exibir no status novamente 

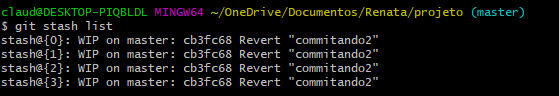


**GUARDANDO DADOS DE ALTERAÇÕES NOS ARQUIVOS PARA DEPOIS REUTILIZAR**

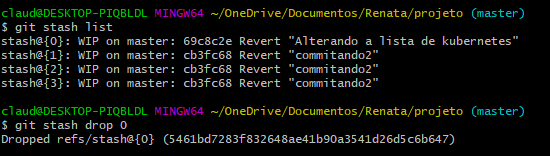
Quando realizarmos alterações no arquivo e caso seja necessário realizar outras alterações naquele arquivo sem ser no mesmo campo que você estava utilizando no momento, o comando abaixo guarda essas informações em background para no futuro se quiser integrar as mudanças feitas no campo atual com as mudanças do anterior.



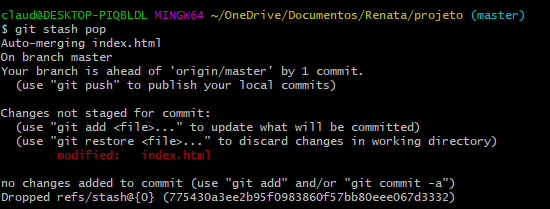
Listando os stash’s criados



O comando abaixo remove as stash não utilizadas

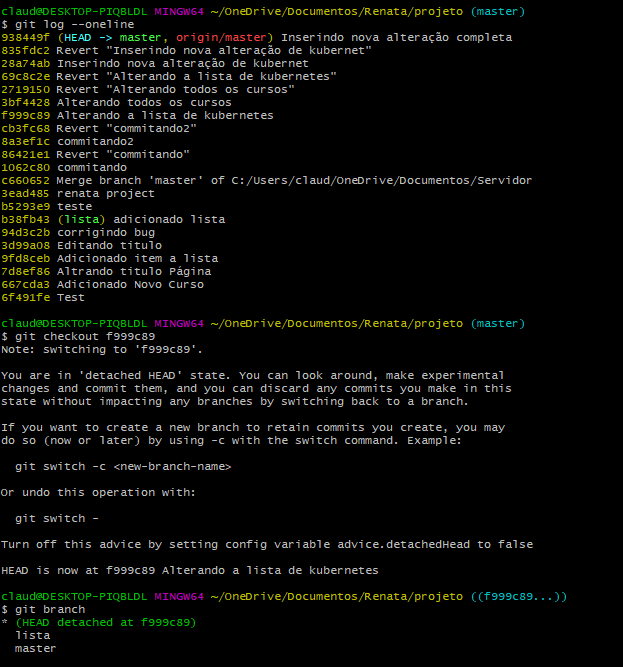


O pop serve para adicionar as mudanças e depois remover a stash criada do git



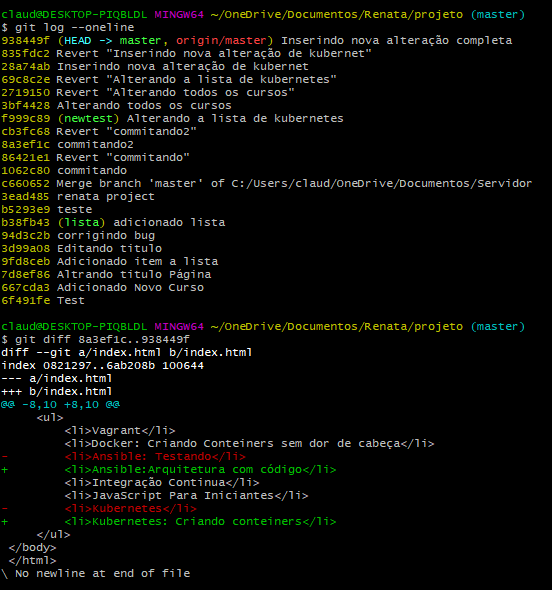
**VOLTANDO NA VERSÃO DO ARQUIVO QUE QUER UTILIZAR**

Se você consultar os log’s de alterações comitadas e pegar os 7 primeiros hash code e dar um checkout, ele vai criar uma branch “fantasma” desanexada da sua branch e sem poder comitar. Com isso vai levar a versão que você deseja utilizar novamente, para utilizar essa nova versão basta criar uma nova branch para utilizar o projeto e poder comitar.



**VERIFICAR MUDANÇAS ENTRE DIFERENTES COMMITS**

Basta colocar os Hash codes que deseja saber no git diff. Exemplo: do hashcode1 até hashcode5, ele vai exibir todas as mudanças feitas no arquivo. O ATÉ no git é colocado com “..” (dois pontos)   
Utilizando somente o git diff vai exibir o que foi modificado no momento que está utilizando o arquivo.



**GERANDO VERSÕES (RELEASES) A PARTIR DOS COMMITS.**

Para criar releases basta utilizar o comando abaixo, e caso eu envie um push para o nome da minha release ela será exibida no campo de releases no github e pronta para download.

