**GIT e GITHUB**

**COMANDOS BÁSICOS – RESUMO** (Obs: não utilize os < >):

**<git init>** iniciar o repositório

**<git add nomeDoArquivo.txt>** para adicionar certo arquivo novo ou modificado à área de staged;

**<git add .>** para adicionar todos os arquivos novos e modificados para a área de staged;

**<git commit –m “mensagem exemplo”>** para commitar o projeto (salvar uma cópia);

**<git log>** para ver os commits que foram realizados. Após, caso tenha vários commits e os dois pontos “**:**” apareça, aperte <**q**> para sair ou <**/palavraExemplo**> para pesquisar um commit com essa palavra.

**<git config user.name “Seu nome aqui”>** para configurar seu nome no git. Ficará a mostra nos commits;

**<git config user.email “seuemail@aqui.com”>** para configurar seu e-mail no git;

<**git checkout nomeDoArquivo.txt**> para retornar o arquivo modified e unstaged ao que era no último commit. É uma espécie de “ctrl + z”. Caso o nome do arquivo contenha espaço, coloque-o entre aspas duplas “”. <**git checkout .**> para retornar todos os arquivos em modified para como eram no último commit. Para retornar os modified e staged ao que eram após o commit, digite <**git checkout HEAD -- nomeArquivo.txt**> (ou utilize o ponto . para incluir todos arquivos).

<**git reset HEAD --hard**> Faz todos os arquivos modified retornarem ao que eram no começo do commit atual. Basicamente é um retorno ao que foi recém commitado. Acho que nem há diferença desse pro último comando acima.

<**git reset HEAD^ --hard**> Descarta o commit atual e retorna para um commit anterior. As modificações commitadas são descartadas.

<**git reset HEAD~1**> Descarta o commit atual, porém deixa os arquivos comitados como unstaged. É bom utilizar em situações em que cria-se um commit mas sem querer esqueceu-se de incluir algumas outras modificações. Ou pode, no lugar, utilizar o git commit --amend.

<**git checkout 00000000**> para navegar até certo commit. Os zeros equivalem ao número hash do commit alvo;

<**git switch -**> para voltar ao commit anterior (após dar o <git checkout> até certo commit);

<**git switch –c novo-nome-do-ramo**> Para criar novo ramo após um checkout de commit. Nome sem espaços!

<**git branch**> para consultar os branches do projeto

<**git branch nomeDoNovoBranch**> Para criar um novo branch

<**git checkout nomeDoNovoBranch**> para mudar para a branch alvo

<**git branch –d nomeDoBranch**> para deletar uma branch

<**git merge nomeDaBranch**> da o merge. Se você estiver no ramo MASTER e der esse comando, juntará a branch nomeDaBranch à MASTER.

<**git merge --abort**> abortar o merge que deu conflito

<**git remote update**> seguido de <**git status –uno**> para verificar se o seu repositório local está atrás, a par ou na frente do repositório remoto.

**COMANDOS IMPORTANTES, MAS NÃO TÃO BÁSICOS**:

<**git commit --amend –m “Sua nova mensagem aqui”**> Para mudar a mensagem do commit atual. Ele commita também! É possível modificar arquivos, adiciona-los à staged area e commitar sobre o commit atual através desse comando. Se não modificar arquivos e apenas executar o comando, ele apenas mudará a mensagem do commit atual.

<**git restore nomeDoArquivo.txt**> restaura o arquivo deletado.

**<git diff>** utilizar quando há arquivos modified. Dá pra ver a diferença para os arquivos antigos (ver as modificações). Não mostra diferenças de certo arquivo se adiciona-lo para a área de staging.

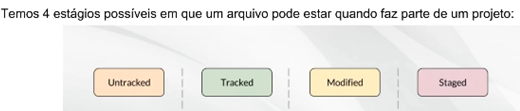
**<git diff --staged>** para comparar os arquivos em staged com o commit anterior.

**Git** é um software de controle de versão.

**Github** é uma plataforma para programadores postarem seus softwares e devidas versões.



**ESTÁGIOS DO GIT E COMO ELES FUNCIONAM:**

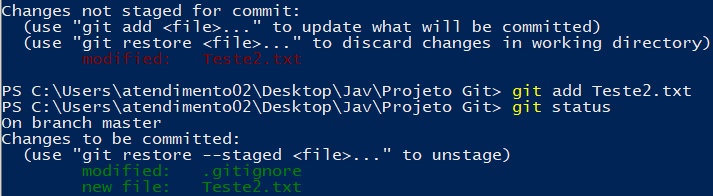


Um arquivo pode “navegar” diversas vezes por estes estágios por várias vezes durante um processo de desenvolvimento.

Ele pode estar em mais de um estágio ao mesmo tempo.

Para iniciar o versionamento de um projeto no git, basta escrever o comando <**git init**> (sem os <>) no terminal dentro da pasta do projeto. Os arquivos do projeto iniciam seu estado como *untracked* (não rastreado). Quando um arquivo novo no projeto é criado, ele passa a ser *untracked* também. Para rastrear um arquivo, basta digitar o código <**git add nomeDoArquivo**>. Se quiser rastrear todos os arquivos da pasta, basta digitar <**git add .**>com esse ponto final. Após utilizar o comando add, o(s) arquivo(s) vão para o estado *staged* (e *tracked*, se for a primeira vez deles) e podem ser commitados.

Se certo arquivo for modificado, ele ficará como *modified* (e continuará como *tracked*, pois ainda está sendo rastreado) e sairá da *staging area*. Deve-se deixa-lo como *staged* novamente para poder comita-lo. Assim, ele ficará como *modified* e *staged*. Se o arquivo não for adicionado à área de “stage”,ele não poderá sofrer commit.



Veja o exemplo acima. No primeiro git status, Teste2.txt está *tracked* e *modified*, mas *unstaged*. Depois de dar um add, ele fica *tracked, modified* e *staged*. Prontinho pro commit!

Na *staging area*, os arquivos que ainda não foram commitados ficam como *new file*. Os arquivos que já foram commitados anteriormente e sofreram modificação ficam como *modified*.

Ao commitar, o cria-se uma “cópia” do projeto e é possível acessa-lo futuramente desse ponto de partida. O comando é o <**git commit –m “Sua mensagem aqui. Diga que alterações foram feitas”**>.

**Aula 16. Fazendo o uso do .gitignore**:

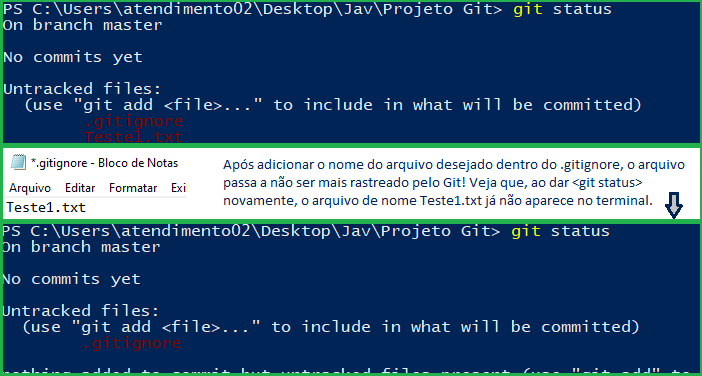
O arquivo .gitignore é um arquivo de texto onde escreve-se todo o arquivo do projeto dos quais o usuário deseja que o Git não rastreie e ignore.

O ponto no início indica um diretório/arquivo oculto.

Cria-se um arquivo de texto com esse nome **.gitignore** e lá adiciona-se o que deseja-se que o git ignore.

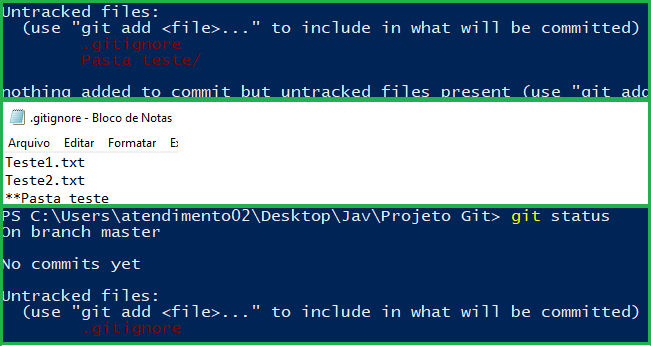
Como fazer isso no Windows: <https://stackoverflow.com/a/12298593>

No Ubuntu, acredito, é só renomear o arquivo. Lá é permitido o nome começar com ponto.

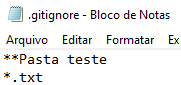


Para adicionar mais arquivos ao .gitignore, basta dar enter e empilhar os nomes dos arquivos!

Pode-se adicionar pastas à lista do .gitignore para serem ignoradas. Para isso, deve-se adicionar dois asteriscos à frente do nome da pasta no arquivo .gitignore, ficando assim: **\*\*pasta exemplo**



Se quiser ignorar todos os arquivos que tenham determinada extensão (.jpeg, .mp3, .mov, etc) então basta escrever, dentro do .gitignore, um asterisco seguido da extensão. **Exemplo: \*.mp3, \*.txt**



Obs: pastas sem conteúdo já são ignoradas automaticamente pelo Git.

**RESUMINDO AULA SOBRE GIT IGNORE**:

- Crie um arquivo de texto chamado .gitignore

- Escreva nele todos os arquivos que você deseja que o Git ignore.

- Para ignorar um arquivo simples, adicione o nome dele (com a extensão) nesse documento de texto.

- Para o git ignorar pastas, escreva \*\* na frente do nome da pasta.

- Para o git ignorar todos os arquivos de determinada extensão, escreva o nome da extensão com um asterisco na frente.

**Aula 17. Operações básicas**:

Quando se inicia um repositório, deve-se configurar o username e e-mail. Comandos:

<**git config user.name “Seu nome aqui”**>

<**git config user.email “seuemail@aqui.com”**>



Faça essa configuração a cada repositório novo criado. Se quiser deixar automático, coloque o comando global.

<**git config --global user.name “seu nome aqui”**> **e** <**git config --global user.email “seuemail@aqui.com”**>

Não execute o comando global quando estiver em uma máquina pública (biblioteca, universidade...).

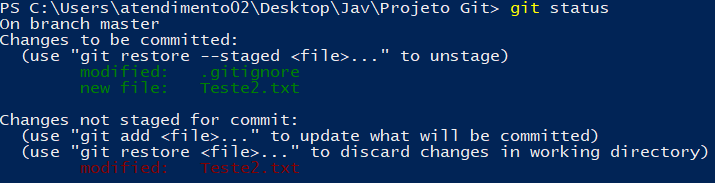
Para adicionar arquivos, escreva <**git add arquivoexemplo.txt**>

Para adicionar todos os arquivos existentes no projeto, escreva <**git add .**>

**Aula 18. Realizando commits**: Um commit vai “salvar” os arquivos rastreados. É possível navegar entre commits, pois um commit é uma cópia do projeto salvo. Para commitar, utilize o comando:

<**git commit –m “Mensagem exemplo. Após criar o capítulo 5. Editei capítulo 4. Bla bla bla”**>

Veja o que ocorreu aqui:



O arquivo Teste2.txt tinha sido adicionado para a área de staged, mas não tinha sido comitado. Então ele foi modificado. Se der um commit agora, o Teste2.txt que não foi modificado será commitado ao invés do que foi modificado. Para a modificação ser comitada, deve-se adiciona-la primeiro à área de staged <**git add Teste2.txt**> e então commitar!

**Aula 20. Consultando o histórico do projeto**:

Digite <**git log**> no terminal. Se houver muitos commits, dois pontos “**:**” aparecerão. Tecle <**q**> para sair (quit) ou digite <**/algumaPalavra**> para pesquisar por um commit que contenha essa palavra.

Se quiser que o git log mostre todos os commits, deve-se mudar a configuração. Para isso, digite <**git config core.pager cat**>

**Aula 21. Conhecendo outros recursos de log do git**:

<**git log -2**> Mostra os últimos 2 commits. Pode substituir esse “2” por qualquer outro número;

<**git log --oneline**> Mostra cada commit apenas em uma linha. Só aparecerá os 7 primeiros dígitos do hashcode;

É possível utilizar dois comandos (ou mais) em conjunto. Exemplo: <**git log --oneline -3**> (ou -3 --oneline);

<**git log --before=”YY-MM-DD**”> Para mostrar os comits antes de certa data. Exemplo: <**git log --before=”2020-05-29”**>. Veja que é ano-mês-dia.

<**git log --after=”YY-MM-DD”**> Para mostrar os commits depois de certa data.

No comando --after ou --before posso escrever “1 week ago”, ou “7 days ago”. Ex: <**git log –after=”1 week ago”**>

<**git log --author=”Seu João”**> Para mostrar os commits de determinado usuário. Não precisa ser o nome completo.

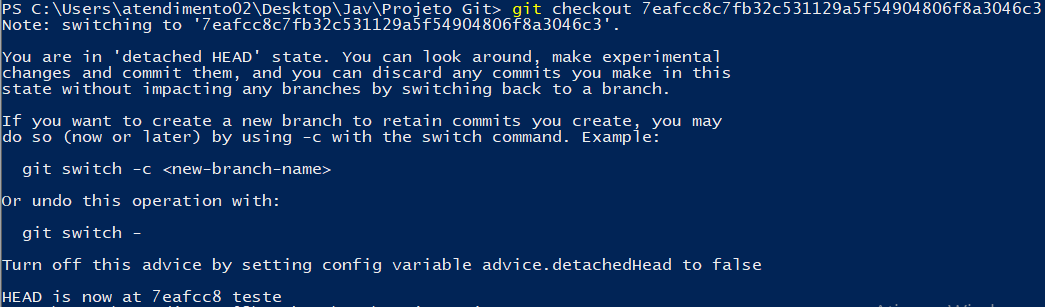
<**git help log**> Para mostrar mais comandos de log.

**Aula 22. Voltando no tempo – Navegando pelos commits**:

Antes de voltar para um commit, é necessário commitar as mudanças ou apaga-las.

Use o comando <**git checkout 00000000**> onde os zeros equivalem ao número hash do commit que se quer voltar.

Após o usar esse comando, você sai do ramo (branch) atual. MASTER é o ramo inicial.



Se, por exemplo, você está no 6º commit e deseja voltar ao 4º e seguir o trabalho de lá, você deve criar um novo ramo. Então, esse ramo partirá desse commit e você poderá depois navegar pelos commits desse ramo novo. Se você está no 3º commit desse ramo novo e deseja retomar o trabalho a partir do 2º commit, deverá criar um novo ramo para partir desse. É possível navegar pelos commits e ramos. Pode-se voltar ao 6º commit do primeiro ramo (MASTER é o ramo inicial, e HEAD é em qual commit você está no momento).

Se você foi até determinado commit para observa-lo e agora quer voltar para o último commit, digite o comando:

<**git switch –**> ou <**git checkout master**> (ou, no lugar do “master”, o nome do ramo em que você está)

Se você foi até o commit e quer criar um novo ramo para seguir seu trabalho a partir dele, deve-se usar o comando:

<**git switch –c novo-nome-do-ramo**> sem espaços no nome do ramo.

Obs sobre um problema que notei... Quando faço o <git checkout> e vou até certo commit, mudo ele e depois faço o <git switch -> para voltar ao commit anterior, a mudança continua.

**Aula 23. Conhecendo novos comandos**:

**Renomear** arquivo: <**git mv arquivoExemplo.txt novoNome.txt**>

O arquivo não sai da área de stage, e, após renomea-lo, já é possível realizar o commit.

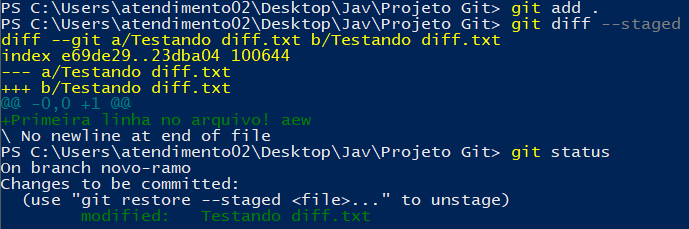
Se você renomear manualmente (sem ser pelo git), então deverá coloca-lo novamente na staging area para poder comitar. Faça isso com <git add .> ou <git add nomeAntigoDoArquivo.txt>

Remover um arquivo: <**git rm nomeDoArquivo.txt**>

Importante: se remover manualmente na pasta, você precisará dar o add no arquivo deletado para atualizar e poder commitar a mudança. Com o comando rm, a atualização já ocorre de maneira automática e não será preciso dar o add para commitar.

Esses comandos podem ser utilizados tanto para arquivos quanto para diretórios.

<**git diff --staged**> Compara o que está na staged area com o arquivo comitado por último, mostrando os arquivos criados/removidos desde o último commit (e que foram mandados para stage).



<**git diff 0000000**> onde os zeros representam o número do commit desejado a se comparar. Esse comando compara a diferença entre o commit atual (ou o que está em staged atualmente mas não foi commitado) e o commit alvo.

<git diff **0000000..0000000**> compara dois commits. Os zeros representam os números hash de cada commit.

Tanto faz a ordem dos hashs. Os dois pontos “..” são necessários!

**Aula 24. Desfazendo as coisas pelo Git – Modificando o commit atual**:

Caso você tenha escrito uma mensagem errada na hora do commit, é possível modifica-la. O comando:

<**git commit --amend –m “Sua nova mensagem aqui”**>

Esse comando muda a mensagem do commit atual. Ele também commita! Então, caso tenha arquivos em staged, eles serão commitados! Isso é bom pois, caso você tenha se esquecido de fazer uma mudança no commit, você pode faze-la, adicionar os arquivos à staged area, e então commitar sobre o commit!

**Aula 25. Tirando arquivos da staged area – “Ctrl + z” nos arquivos**:

Para remover certo arquivo da staged area, utilize o seguinte commando:

<**git restore --staged nomeDoArquivo.txt**> ou utilize um ponto “**.**” Para remover todos arquivos da staged area.

Se certo arquivo for deletado, utilize o <**git restore nomeDoArquivo.txt**> para retorna-lo.

É possível retornar o arquivo modified para o que ele tinha no commit anterior.

<**git checkout nomeDoArquivo.txt**>

Obs: Se o nome do arquivo contém espaços, coloque-o dentro de aspas duplas! Ex:

<git checkout “Arquivo teste.txt”>

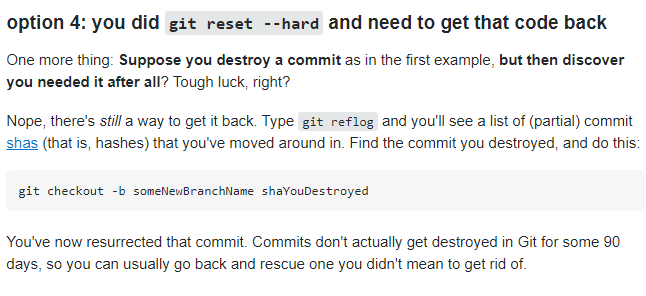
<**git reset HEAD --hard**> Faz todos os arquivos modified retornarem ao que eram no commit atual. Basicamente é um retorno ao que foi recém commitado.

<**git reset HEAD^ --hard**> Descarta o commit atual e retorna para um commit anterior. Todas modificações salvas no commit são descartadas também.

<**git reset HEAD~1**> Descarta o commit atual, porém deixa os arquivos commitados como unstaged.

<**git reset HEAD~1 --soft**> Descarta o commit atual, porém deixa os arquivos commitados como staged.

* Voltando atrás após fazer um reset --hard:

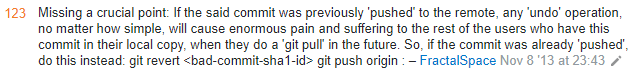


**O git reflog mantém uma lista de todos commits em que a HEAD já apontou. Acho que essa lista é mantida por 90 dias.**

**Link**: <https://stackoverflow.com/a/6866485> (o thread inteiro possui outras respostas muito boas)

Basicamente você vai criar um novo branch com o commit que você quer recuperar. Aí você pode voltar para a branch em que você estava e dar o merge.

Caso já tenha mandado para o repositório, ler essa resposta:



Mais alguns links sobre o assunto:

Undo a git reset hard (com arquivos na staged): <https://stackoverflow.com/questions/7374069/undo-git-reset-hard-with-uncommitted-files-in-the-staging-area>

Git basics: Undount things

<https://git-scm.com/book/en/v2/Git-Basics-Undoing-Things>

**Seção 4 – Git Intermediário**

**Aula 28. Trabalhando com branches**:

Um branch é uma ramificação do projeto. É como um galho de uma árvore. O ramo MASTER é o ramo inicial. Se você dá um git checkout e volta alguns commits e começa a trabalhar a partir daí (e fazendo novos commits) esse será um novo ramo. Você necessitará nomear esse novo ramo. É possível retomar ao ramo MASTER, voltar a trabalhar lá e criar até mesmo novos branches.

O ponto onde um novo branch inicia é chamado de base.



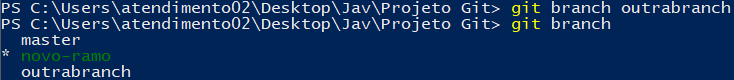
É possível consultar os branches do projeto pelo comando <**git branch**>

O asterisco indica o ramo em que se está trabalhando.



Para criar um novo branch, digite <**git branch nomeDoNovoBranch**>

Esse comando criará um novo branch a partir do commit atual. Há como criar um branch a partir de um commit anterior ao atual. Esse comando apenas criará a branch, mas não será a atual em que você está trabalhando.



Para mudar para a branch alvo, gitie <**git checkout nomeDoNovoBranch**>

Ou <git checkout –b nomeDoNovoBranch> para criar o branch e se mudar para lá automaticamente.

Se você tinha arquivos modificados (ou em staged) e mudou de ramo, ao commitar eles ficarão salvos no novo ramo.

As ramificações são muito boas para testar novas funcionalidades do app. É uma boa prática nomear cada branch com a funcionalidade que se quer implementar, caso esse seja o motivo da sua criação.

Para deletar um branch, utilize o comando <**git branch –d nomeDoBranch**>

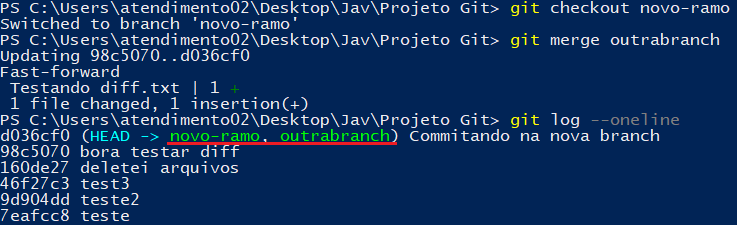
Esse comando será executado caso o branch esteja fundigo (merged) com o principal. Caso queira, mesmo assim, deleta-lo, então utilize o **–D** maiúsculo.

**Aula 29. Trabalhando com merge**:

Vá até a sua branch principal (a que você quer receber determinada branch) e digite o comando:

<**git merge nomeDaBranch**>

Se você estiver na MASTER, juntará a branch nomeDaBranch à MASTER.



O commit d036cf0 era o único na branch *outrabranch*, enquanto o *novo-ramo* parou no commit 98c5070. Esse commit era a base da *outrabranch*. Dando o merge, esse commit d036f0 começa onde se iniciou sua base.

Mas surge uma dúvida: e ser houver mais commits na branch principal que ocorreram após a ramificação (após a base da nova branch)? O que ocorrerá ao dar o merge? R: vai dar conflito caso os arquivos modificados na nova ramificação tenham também sofrido modificações no ramo master após a criação do novo ramo.

Duvida: e se eu gostar de um branch e quiser continuar trabalhando nele? Lembrando q a base dele não foi o último commit do máster. Como q ficará no merge?

O que eu percebi: se você criar uma nova branch e commitar novos arquivos nela e voltar para a Master, pode até criar mais commits na Master e dar o merge depois e não haverá conflito DESDE QUE NÃO ALTERE OS ARQUIVOS QUE SOFRERAM ALTERAÇÕES NO NOVO RAMO.

Se você retrocedeu para um commit na MASTER, criou um novo ramo a partir daí e modificou arquivos commitados após a base na MASTER, e depois quer ir pra máster dar merge, então terá que, dentro da máster, dar reset hard até o commit base e então dar o merge. Caso contrário ocorre conflito.

Para abortar um merge que deu conflito, digite o comando <**git merge --abort**>

**Aula 30. Trabalhando com rebase**:

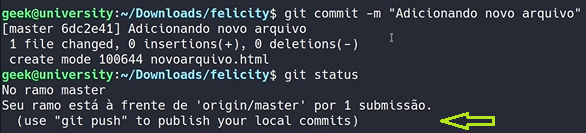
Rebase é parecido com o merge. A diferença é que o rebase “remonta” a base. Na minha visão, isso dá quase na mesma. Isso não evita conflitos, só deixa o histórico mais bem visível.

Se você está na branch X e dá um rebase <git rebase y>, então vai colocar o ‘y’ abaixo do início de ‘x’.

Explicação (vá em “entendendo passo a passo”): <https://pt.stackoverflow.com/questions/81221/como-funciona-o-git-rebase/81241#81241>

**Aula 31. Trabalhando com clone e push**:

O clone é a cópia de um projeto que utiliza o git. Este projeto pode estar em servidor remoto ou local. Ao clonar um projeto, é possível fazer alterações nele e enviar novamente para o repositório alvo através do push. É possível ver quando seu repositório está com commits a mais do que o original, ou quando está desatualizado.



Exemplo do comando: <**git clone *home/Daniel/projetos/meuapp* .** >

Deve-se colocar após o ‘clone’ o caminho do diretório ou o link do projeto do qual se queira clonar (mais pra frente, no curso, será tratado sobre o assunto GitHub) seguido de um ponto final.

Ao tentar dar um push para enviar seu projeto para o repositório central, não será possível pois, de início, este é um repositório ‘non-bare’. Este assunto será tratado após a próxima aula.

**Atenção**: Se o diretório alvo conter espaços no nome (Exemplo: Minha Pasta Exemplo), então coloque o endereço entre aspas. Exemplo: <git clone **“**home/Daniel/projetos/Minha Pasta Exemplo**”** .>

**Aula 32. Trabalhando com fetch e pull**:

O fetch baixa os últimos commits do repositório mas não faz o merge. Dessa forma, é possível fazer um rebase dos arquivos ao invés do merge. Apenas <git rebase> após um fetch já faz o serviço.

O pull é a junção do fetch com o merge. Esses dois comandos são realizados “em uma tacada só”.

“De modo simplório, o git fetch buscas as diferenças em relação ao ramo atual, mas não altera nada nesse branch. Já git pull faz o git fetch e faz o merge das diferenças.”

**Aula 33. Trabalhando com bare repository**:

Bare repository é um repositório central. Ele pode ser local ou remoto (o Github, por exemplo, é um bare repository remoto. Os programadores trabalham num projeto e submetem seus arquivos para o bare repository).

Para criar um repositório bare local, abra a pasta do repositório e escreva o comando no terminal:

<**git init --bare**>

Pode-se clonar os arquivos desse repositório e fazer um push. Após clonar um repositório, não há mais necessidade de colocar o endereço do repositório após <git merge> ou <git pull>.

Você não irá conseguir clonar já tenha feito um git init na pasta.

**Aula 34. Trabalhando com tags**:

Utiliza-se Tags quando uma versão “entregável” do projeto é alcançada. Assim, você cria uma tag para identificar esse trecho do projeto.

Comando: <**git tag suaTag**>

Exemplo: <git tag V1.0>

Utilize <git tag> para ver todas as tags existentes no projeto. É possível dar checkout para ir a uma tag.

Quando se tem uma tag e queira dar um pull, utilize o comando <**git pull origin suaTag**>. Assim, os arquivos e a tag vão juntos.

Se der apenas <git pull> a tag não vai junto.

No GitHub, na seção “releases”, ficarão disponíveis as versões do projeto.



É possível criar tags com anotações:

<**git tag -a suaTag -m mensagem**>

Exemplo: <git tag -a V.10 –m “Versão pronta com night mode”>

Para ver informações sobre a tag, use o comando <**git show suaTag**>

Para excluir uma tag, utilize o comando: <**git tag -d suaTag**>

E excluir no repositório: <**git push --delete origin suaTag**>

GIT STASH:

Basicamente é um “backup dos arquivos modificados”. Imagine que você começou o desenvolvimento numa branch errada. Você pode dar o git stash, ir para o ramo principal e migrar pra lá os arquivos que foram colocados no stash.

<https://medium.com/wooza/git-stash-conhecendo-e-utilizando-um-dos-comandos-mais-práticos-para-o-versionamento-de-seu-código-a4dab3ac70da>

**Seção 5 – Git Intermediário**

**Aula 38. Criando um repositório no GitHub**:

Se você tem um projeto local, pode criar um repositório no Github e adicionar seu projeto à essa pasta remota. Vá no Github e crie o repositório. Para adicionar seu projeto lá, execute o comando no terminal:

<**git remote add origin https://www.github.com/daniel/projeto1**>

Para dar push, use o comando:

<**git push -u origin master**> Esse comando só irá dar push no ramo master. Você pode escolher outro branch para dar o push.

Para verificar se já há algum repositório (remoto ou local) registrado no seu projeto, utilize o comando:

<**git remote -v**>

Deve-se informar sempre o usuário e senha do Github a cada push. Para não precisar, utilize o comando <**git config credential.helper store**>. No próximo push você precisará informar as credenciais, mas na seguinte elas já ficarão salvas nesse projeto local.

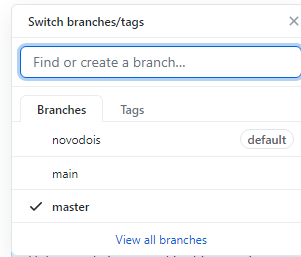
Não é recomendado utilizar esse comando em um computador público pois sua senha fica salva nos arquivos git.

**Aula 39. Fazendo uso do clone e do pull**:

Comando: <**git clone https://github.com/seuusuario/seurepositorio.git**>

Não é necessário dar <git init> na pasta para dar o clone. Esse comando cria um diretório dentro da pasta aberta no terminal com o mesmo nome do repositório. Ou seja, o clone é nada mais do que um download. Na verdade, você pode fazer o download do .zip

É possível que o clone baixe nenhum arquivo caso o default no github esteja no main e lá ter nenhum arquivo. Você deve, então, mudar o default para o ramo desejado ou fazer o download manualmente.



Ao abrir a pasta baixada pelo git clone ou download manual, não será necessário dar <git init> nela. Ela já foi iniciada.

Para atualizar seu projeto com o que há no github, utilize o comando:

<**git pull origin suaBranch**>

A branch pode ser a que você desejar. Eu fiz um teste e apenas digitei <git pull>, e foi atualizada a branch em que eu estava.

* Para verificar se há alguma atualização no repositório remoto, primeiro utilize os comandos seguintes:

<**git remote update**> Para atualizar as referências remotas;

E depois

<**git status –uno**> Esse comando dirá se o seu ramo no seu repositório local está atrás, a par, ou na frente do repositório remoto.

Se no comando remote update você utilizar o –v <git remote -v update> então não precisará mais de nenhum comando, pois ele indicará quais branches foram atualizadas (visualmente é meio ruim).

**Aula 40. Adicionando colaboradores no seu projeto**:

Colaboradores podem dar pull no projeto. Mesmo que um projeto seja público, apenas colaboradores podem altera-lo.

Para adicionar colaboradores, vá ao seu repositório -> settings -> manage access -> Invite a collaborator

Adicione o e-mail da conta do colaborador desejado.

**Aula 41. Consultando o histórico de commits**:

A