Criando usuário do git:  
  
git config --local user.name "Seu nome aqui"

git config --local user.email "seu@email.aqui"

Assim todos os commits executados neste repositório serão atribuídos à pessoa com nome **Nome da pessoa**. Para mais detalhes e outras configurações possíveis (até algumas mais avançadas), você pode conferir este link: <https://git-scm.com/book/en/v2/Customizing-Git-Git-Configuration>.

**Controle de versão**, pois se temos diferentes versões do código precisaremos de um sistema que controle essas versões. E é isso que o Git fará para nós. Este não é o único sistema de controle de versão que existe. Outras alternativas são:

* CVS
* SVN
* Mercurial
* GIT

OBS: Caso o caminho da sua pasta possua espaços, é preciso colocá-lo entre aspas. Exemplo: cd 'Documents/Curso Git e GitHub'

1-

No terminal (ou **Git Bash**, no Windows) navegue até a pasta recém criada (utilize o comando cd para navegar entre pastas);

OBS: Caso o caminho da sua pasta possua espaços, é preciso colocá-lo entre aspas. Exemplo: cd 'Documents/Curso Git e GitHub'

Na pasta do projeto, execute o comando git init para inicializar um repositório Git;

Execute o comando git status para garantir que você está em um repositório Git e que o arquivo **index.html** é reconhecido. Leia com calma a saída deste comando.

* O que são (e para que servem) **sistemas de controle de versões** e como eles podem ajudar o nosso fluxo de desenvolvimento
  + Nos ajudam a manter um histórico de alterações;
  + Nos ajudam a ter controle sobre cada alteração no código;
  + Nos ajudam para que uma alteração de determinada pessoa não influencie na alteração realizada por outra;
  + Etc.
* O que é o Git e como instalá-lo
* Que com o comando git init nós conseguimos criar um repositório Git;
* Como analisar o estado do nosso repositório através do comando git status.

2-

Ao executar o comando git status, recebemos algumas informações que talvez não estejam tão claras, principalmente quando nos deparamos com termos como HEAD, working tree, index, etc.

Apenas para esclarecer um pouco, visto que entenderemos melhor o funcionamento do Git durante o treinamento, seguem algumas definições interessantes:

* HEAD: Estado atual do nosso código, ou seja, onde o Git os colocou
* Working tree: Local onde os arquivos realmente estão sendo armazenados e editados
* index: Local onde o Git armazena o que será *commitado*, ou seja, o local entre a *working tree* e o repositório Git em si.

Além disso, os possíveis estados dos nossos arquivos são explicados com detalhes neste link: <<https://git-scm.com/book/pt-br/v2/Fundamentos-de-Git-Gravando-Altera%C3%A7%C3%B5es-em-Seu-Reposit%C3%B3rio>.

**Sair da tela de scroll**

Você deve ter reparado que ao executar git log -p, o git nos mostrou uma tela onde é possível rolar para baixo e para cima através das setas. Isso não é algo específico do git, mas sim do próprio terminal do sistema operacional. Quando finalizarmos a visualização do log, basta apertar a tecla q para voltar "ao normal" em nossa linha de comando.

* Que um commit é a forma de salvar um estado ou versão do nosso código;
* Como adicionar arquivos para serem *commitados* com git add;
* Como *commitar* arquivos, utilizando o comando git commit;
* Como verificar o histórico de *commits*, através do git log e algumas de suas opções:
* git log (informações do histórico de alçoes)
* git log --oneline (informações resumidas em uma linha)
* git log -q (informações detalhadas das alterações)
* git log --pretty="format:%h %s" (aparece apenas o começo do hash)
* git log --pretty="format:%h %s "ae"
* (nome do autor no fim)
* ( https://devhints.io/git-log - link dos comandos para git log)
* Como fazer o Git não monitorar arquivos, através do **.gitignore**
* **(Fazer um arquivo chamado ( .gitignore ), colocar o nome da sua pasta nessa pasta do git e comitar o gitignore).**
* Que não devemos realizar commit, ou seja, salvar um estado, da nossa aplicação que não esteja funcionando.

3-

* O que são repositórios remotos;
* Como criar um repositório Git sem uma cópia dos arquivos (com --bare) para ser utilizado como servidor;
* Como adicionar links para os repositórios remotos, com o comando git remote add;
* Como baixar um repositório pela primeira vez, clonando-o com o comando git clone;
* Como enviar as nossas alterações para um repositório remoto, com git push;
* Como atualizar o nosso repositório com os dados no repositório remoto, utilizando git pull;
* O que é e para que serve o **GitHub**;
* Como criar um repositório no **GitHub**;
* Como adicionar um repositório do **GitHub** como repositório remoto.

4-

* Que uma *branch* (ou ramo) é uma linha de *commits* separada, e que pode ser utilizada para desenvolver funcionalidades independentes;
* Que com *branches* separados, podemos evitar que o código de uma funcionalidade interfira em outra;
* Como trazer o trabalho realizado em uma *branch* para outra *branch*, como por exemplo, o master, através do comando git merge;
* Que o git merge gera um novo *commit*, informando que houve uma mescla entre duas *branches*;
* Como trazer os *commits* de uma *branch* para outra, com o git rebase
* Que o git rebase não gera um *commit* de merge, simplificando o nosso *log*;
* Como os conflitos são apresentados pelo Git;
* Como resolver os conflitos e manter apenas as alterações desejadas com o Git.

4-

* Que o Git pode nos ajudar a desfazer alterações que não vamos utilizar;
* Que, para desfazer uma alteração antes de adicioná-la para commit (com git add), podemos utilizar o comando git checkout -- <arquivos>;
* Que, para desfazer uma alteração após adicioná-la para commit, antes precisamos executar o git reset HEAD <arquivos> e depois podemos desfazê-las com git checkout -- <arquivos>;
* Que, para revertermos as alterações realizadas em um commit, o comando git revert pode ser a solução;
* Que o comando git revert gera um novo commit informando que alterações foram desfeitas;
* Que, para guardar um trabalho para retomá-lo posteriormente, podemos utilizar o git stash;
* Que, para visualizar quais alterações estão na stash, podemos utilizar o comando git stash list;
* Que, com o comando git stash apply <numero>, podemos aplicar uma alteração específica da stash;
* Que o comando git stash drop <numero> remove determinado item da stash;
* Que o comando git stash pop aplica e remove a última alteração que foi adicionada na stash;
* Que o git checkout serve para deixar a cópia do código da nossa aplicação no estado que desejarmos:
  + git checkout <branch> deixa o código no estado de uma branch com o nome <branch>;
  + git checkout <hash> deixa o código no estado do *commit* com o hash <hash>.

5-

* Que é possível visualizar quais alterações foram realizadas em cada arquivo, com o comando git diff;
* Que, digitando apenas git diff, vemos as alterações em nossos arquivos que não foram adicionadas para commit (com git add);
* Que é possível comparar as alterações entre duas *branches* com git diff <branch1>..<branch2>
* Que é possível comparar as alterações feitas entre um commit e outro, através do comando git diff <commit1>..<commit2>;
* Que o Git nos possibilita salvar marcos da nossa aplicação, como por exemplo, lançamento de versões, através do git tag;
* Que o comando git tag -a é utilizado para gerar uma nova *tag*;
* As ***Releases*** do GitHub, que são geradas para cada *tag* do Git criada em nosso repositório.

Como fazer repositório remoto:  
  
Chegamos à parte de implementação de um **repositório remoto**, um servidor local para onde possamos enviar nossas alterações, que ficarão acessíveis para outras pessoas. Na pasta que contém os arquivos com os quais trabalhamos até então ("vinicius"), utilizaremos o comando cd .. para nos localizarmos na pasta superior, no caso, "git-e-github", e criaremos a pasta "servidor" por meio do comando mkdir servidor.

E então acessaremos esta pasta, com cd servidor, dentro da qual rodaremos git init. Como este servidor será um repositório do Git que somente armazenará as alterações, ou seja, não o acessaremos para editar arquivos, por exemplo, usaremos git init --bare, cujo parâmetro indica que este repositório é **puro**, que contém apenas as alterações dos arquivos, e não uma cópia física de cada um dos arquivos.

Isso nos traz algumas facilidades e permite que adicionemos este repositório remotamente em outro. Após a criação do repositório, o Git nos fornece o caminho para ele, que serve como nosso servidor. Copiaremos o caminho, no caso C:/Users/ALURA/Documents/git-e-github/servidor, e voltaremos à pasta "vinicius", onde se encontra nosso projeto, por meio do comando cd ../vinicius.

Executaremos git status para nos certificarmos de que estamos no repositório correto, e em seguida, uma vez que passamos a trabalhar com dois repositórios, queremos fazer com que o servidor reconheça o repositório remoto, este endereço, para que ele consiga enviar os dados para lá futuramente.

Se executarmos o comando git remote, teoricamente, nada acontece. Mas na verdade, todos os repositórios remotos que o repositório local conhece são listados, que até o momento é nenhum. Portanto, adicionaremos um, com git remote add local C:/Users/ALURA/Documents/git-e-github/servidor, e para quantos repositórios remotos quisermos, poderemos dar algum nome, no caso, local, também incluiremos um caminho, que poderá ser uma URL de um servidor pela internet, um endereço na rede, inclusive de outro computador, qualquer endereço válido para um repositório Git. Neste caso, será uma pasta no próprio servidor.

Depois que pressionamos "Enter", aparentemente nada acontece, e se usarmos o comando git remote, o retorno será local. Se quisermos garantir que o endereço esteja correto, poderemos executar git remote -v, que faz com que o endereço de local seja exibido. Além disso, é indicado que os dados deste caminho serão buscados (fetch), e enviados para este mesmo caminho (push).

Em situações complexas, de uma infraestrutura de redes mais robusta, poderíamos fazer o envio para um local e a busca viria de outro. Não é nosso caso, portanto não nos preocuparemos com isto no momento. Já criamos um repositório remoto, que adicionamos no repositório local, e agora passaremos a imaginar que a Ana está trabalhando conosco e precisa baixar os dados contidos neste repositório.

Voltaremos à pasta "git-e-github" por meio de cd .., e criaremos uma pasta para a Ana, com mkdir ana. Acessaremos a pasta com cd ana, e ela então precisará **clonar o repositório**, é assim que chamamos quando queremos trazer todos os dados de um repositório remoto para o nosso repositório local pela primeira vez.

Sendo assim, executaremos git clone /c/Users/ALURA/Documents/git-e-github/servidor, para que sejam trazidos os dados do repositório localizado neste endereço. Isso fará com que dentro da pasta "ana" seja criada uma pasta chamada "servidor". Porém, não é o que queremos; queremos que a pasta seja "projeto", por exemplo, e para isso executaremos git clone /c/Users/ALURA/Documents/git-e-github/servidor projeto.

Após "Enter", somos informados de que o clone foi realizado, mas há um aviso de que o repositório clonado está vazio. Mas não adicionamos o repositório remoto no repositório do Vinicius? Sim, porém não enviamos os nossos dados para ele! Portanto, a Ana não possui acesso a eles, e é por isto que o repositório dela está vazio.

* git init –bare (Com este comando nós criamos um repositório que não terá a working tree, ou seja, não conterá uma cópia dos nossos arquivos. Como o repositório servirá apenas como servidor, para que outros membros da equipe sincronizem seus trabalhos, poupamos espaço de armazenamento desta forma).
* git remote add nome-repositorio caminho/para/o/repositorio

Antes de sincronizar as nossas mudanças no código com algum repositório remoto, precisamos adicioná-lo ao nosso repositório local.

* git push [repositorio] master

Desta forma, nós enviamos as alterações em nosso branch master (falaremos mais sobre branches já já) para o repositório remoto. Basta substituir [repositorio] pelo nome que demos ao repositório ao adicioná-lo. Já para trazer os dados que estiverem no repositório remoto, podemos utilizar o git pull [repositorio] master.

*Método de autenticação , tem que fazer o token e colar ele no lugar da senha:*

<https://docs.github.com/pt/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token>

$ git clone <https://github.com/username/repo.git>

Username: your\_username  
 Password: your\_token

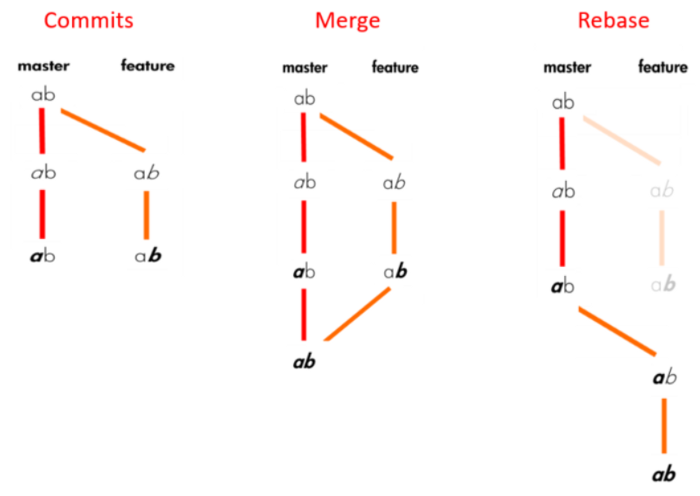
***Branches***("ramos") são utilizados para desenvolver funcionalidades isoladas umas das outras. A *branch* master é a *branch* "padrão" quando você cria um repositório.

É interessante separar o desenvolvimento de funcionalidades em *branches* diferentes, para que as mudanças no código para uma não influencie no funcionamento de outra.

**Diferença entre Merge e Rebase:**

O merge junta os trabalhos e gera um merge commit. O rebase aplica os commits de outra branch na branch atual.

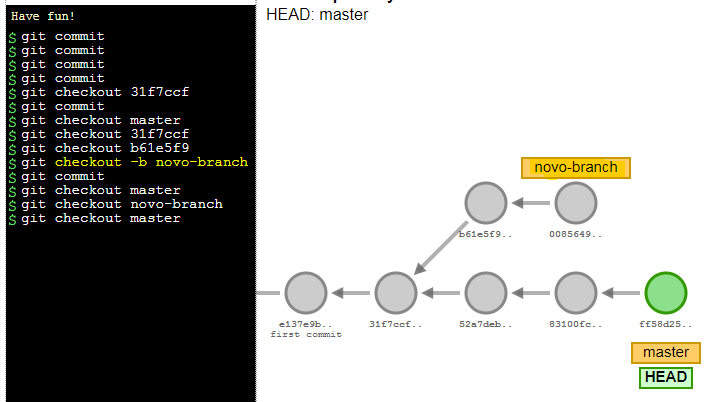
<https://medium.com/datadriveninvestor/git-rebase-vs-merge-cc5199edd77c>



* **git push nomedoserv master (enviar alterações para o servidor)**
* **git pull nomedoserv master (buscar alterações do servidor)**

**Git stash** é usado quando precisamos pausar o desenvolvimento de alguma funcionalidade, ou correção, antes de finalizar, talvez não seja interessante realizar um commit, pois o nosso código pode não estar funcionando ainda. Nesse caso é interessante salvar o trabalho para podermos voltar a ele depois.

**Git checkout** serve Para deixar o nosso código em determinado estado. A descrição do comando git checkout --help, em uma tradução livre é: "Atualizar os arquivos na working tree para ficarem na versão especificada. [...]". Basicamente, podemos deixar o nosso código no estado do último commit de uma branch, de um commit específico, ou mesmo tags (que veremos adiante).



E caso você esteja na branch master querendo mudar para main, pode rodar esses comandos no terminal ou Git Bash:

git branch -m master main

git push -u origin main

**Passo a passo commit com o Git:**

*Já dentro da pasta em que você vai fazer as alterações-*

1. Criar usuário

git config --local user.name "Seu nome aqui"

git config --local user.email “[seu@email.aqui](mailto:seu@email.aqui)”

1. Iniciar o Git e conferir

git init

git status

1. Adicionar os arquivos para comitar

git add .nomedoarquivo ou git add . (para adicionar tudo na pasta)

1. Conferir os arquivos e fazer commit

git status

git commit -m “descrição breve da ação ou alteração”

1. Identificar o histórico de commits

git log

o git log --oneline (informações resumidas em uma linha)

o git log -q (informações detalhadas das alterações)

o git log --pretty="format:%h %s" (aparece apenas o começo do hash)

o git log --pretty="format:%h %s "ae"

(nome do autor no fim)

( https://devhints.io/git-log - link dos comandos para git log)

**Passo a passo ignorando arquivos com o Git:**

1. Verificar quais arquivos são não comitados

git status

1. Fazer arquivo para armazenar o nome dos arquivos a ignorar

.gitignore

1. Colocar o nome do arquivo a ignorar dentro do ignore

Dentro do arquivo **.gitignore** > arquivo\_para\_ignorar.(hmtl/css)

1. Conferir e adicionar ignore ao commit

git status (tem que aparecer apenas o .gitignore)

git add .gitignore

git commit -m “adicionando .gitignore”

**Passo a passo fazendo um repositório remoto no Git:**

*Dentro da pasta onde vai fazer o repositório –*

1. Criando o diretório vazio

git init --bare (copie o diretório)

1. Na pasta onde fez o master

git status (para conferir se está tudo certo)

git remote (conferir se tem algum repositório remoto)

git remote add **nomedoseurepositório** D:/local/do/dire

1. Confirmar o diretório

git remote

*nomedodiretório*

git remote -v (informação de fetch e push)

**Passo a passo sincronizando os dados do repositório:**

*Dentro da pasta onde a pessoa vai sincronizar os dados-*

1. Enviando dados para o repositório (push)

git remote

*nomedodiretório*

git push local master (dentro da pastado dos arquivos a serem enviados inicialmente)

1. Clonar o servidor e fazer uma pasta para salvar o projeto

*Dentro da pasta onde a outra pessoa vai sincronizar os dados-*

git clone /c/local/do/seu/servidor/ **projeto**

**(projeto é a nova pasta que vai ser criada, tem que ter um espaço do local)**

1. Fazendo o pull dentro da pasta projeto

git remote

origin (nome do repositório diferente)

git remote rename origin local

(renomeando o nome para local)

git remote (conferir se foi )

git pull local master

ls (conferir se os arquivos foram)

Caso for fazer alterações não esquecer comitar para salvar, e no fim fazer o git pull para sincronizar a mudança.

**Passo a passo fazendo branchs:**

1. Fazendo a Branch

git branch titulo (o local em que vai ser definido para trabalhar)

git branch

\* master

titulo

git checkout titulo

Switched to branch 'titulo'

(Faz a alteração dos arquivos)

1. Comitando a Branch

git status

git add .nomedoarquivoalterado

git commit -m "Alteração"

git log (ver as alterações na branch dela)

1. Fazendo outra branch para outra pessoa fazer alterações

git checkout -b nomedabranch (atalho)

**Passo a passo unindo e atualizando branchs:**

1. Ir para a master depois de alterar a branch

git checkout master

git merge nomedabranch

:x(enter) para unir os trabalhos

1. Atualizando a branch

git rebase nomedabranch

(git log --graph para mostrar o histórico com as linhas)

**Passo a passo resolvendo conflitos de branchs:**

1. Fazendo o merge

git merge nomedabranche

Auto-merging

CONFLICT(content)

-Visual Studio code mostra o conflito

1. Ver as duas versões do mesmo local em que foi feito a alteração e escolher o melhor código e excluir o outro

git status

git add nomedoarquivo

git commit

**Passo a passo desfazendo alterações no Git:**

1. Se a alteração não foi adicionada para commit

git status

git checkout -- nomedoarquivo.html

1. Se ja foi adicionada mas não foi feito o commit

git status

git reset HEAD nomedoarquivo.html

1. Alterações que ja foram comitadas

git log

Copiar o resh do commit que quer desfazer

git revert 00000000000000 (o resh inteiro ou os 7 primeiros)

:x(enter) para continuar

**Salvando alterações para depois:**

git stash

git stash list ( lista os arquivos stash)

git stash pop (tira da stash e remove as alterações)

**Voltando para o commit anterior:**

1. Pegar a resh que quer voltar

git log --oneline

git checkout 0000000 (resh em que quer voltar)

1. Se for salvar as alterações que fizer na resh anterior

git checkout -b nomedanovabransh

1. Se for apenas voltar sem salvar as alterações

git checkout master

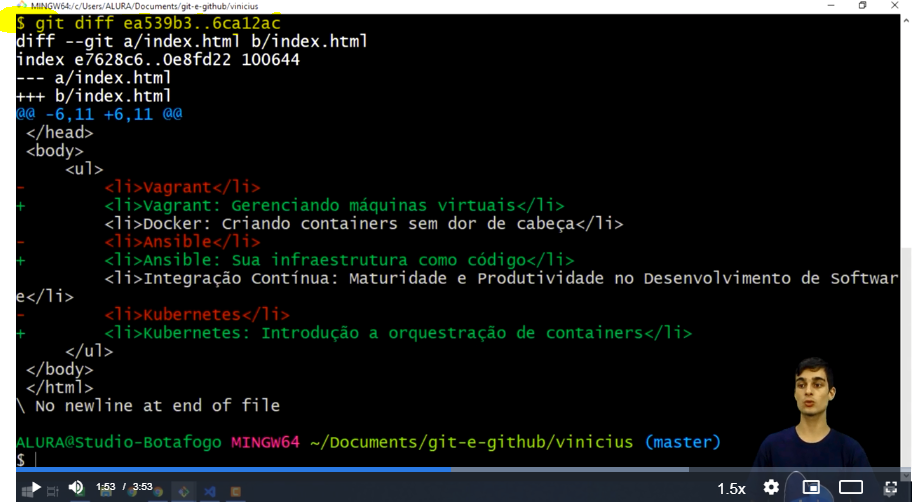
**Passo a passo vendo alterações:**

1. Visualização em linha

git log –oneline (ver alteração resumida em linha e começo do código de commit)

1. Alterações não commitadas e dentro de determinados commits

git diff (se não foi commitado aparece)

git diff 0000000..999999 (inicio e fim dos commits a serem analisados)  


**Passo a passo fazer “check point”:**

1. Nomeando a tag

git tag -a **nomedatag** -m "se quiser descrever o que é a tag"

git tag (aparece todas as versões)

1. Fazendo o push da tag

git push **local** nomedatag

git push **origin** master nomedatag (do github)

1. No git hub a tag aparece como release

(conseguimos baixar um arquivo compactado com o nosso código neste ponto)