**Aula 1 - Comandos Básicos Para um Bom Desempenho no Terminal**

Interagimos com o **Git** por meio da linha de comando (line interface), existem softwares que facilitam essa interação. Vamos aprender:

* Mudar de pastas;
* Listar as pastas;
* Criar pastas/arquivos;
* Deletar pastas/arquivos.

Usuários Windows:

* cd
* dir
* mkdir
* del/rmdir

Usuários Unix (Linux):

* cd
* ls
* mkdir
* rm -rf

Vamos aprender os primeiros comandos de navegação:

**dir <> ls** - Lista os arquivos/pastas no diretório atual.

**cd \** - Vai para o diretório base do diretório C e lista as pastas/arquivos.

**cd <nome\_pasta>** - Vai para a pasta descrita.

**cd ..** - Volta à pasta anterior.

**cls <>** **clear** ou **(Ctrl + L)** - Limpa a tela do prompt de comando.

**Tecla Tab** - Tem a função de autocompletar.

**mkdir <nome\_pasta>** - Cria uma pasta.

**echo <frase>** - Printa no terminal o valor/frase digitado.

**> <nome arquivo.txt>** - É um redirecionador de fluxo, pega a saída de uma função e joga em um arquivo com o nome escolhido.

**del <nome\_pasta> -** Deleta apenas os arquivos na pasta (**Windows**).

**rmdir <nome\_pasta> /s /q** - Deleta a pasta citada (**Windows**).

**rm -rf <nome\_da\_pasta>** - Deleta todas as pastas contidas na pasta citada (**Linux**).

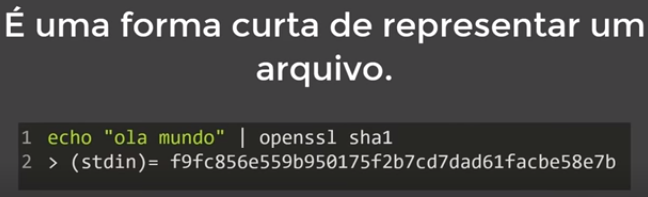
**pwd** - Mostra o caminho (endereço) da pasta.

**Instalar o Git:** <<https://git-scm.com/>>

**Aula 2 - Tópicos Fundamentais Para Entender o Funcionamento do Git**

A sigla **SHA** significa **Secure Hash Algorithm** (Algoritmo de Hash Seguro), é um conjunto de funções *hash* criptográficas projetadas pela NSA (Agência de Segurança Nacional dos EUA). A grosso modo, é um algoritmo que pega o arquivo (imagem, frase, etc), vai embaralhar de uma forma muito específica.

A saída desses dados criptografados gera um conjunto (único) de caracteres identificador de 40 dígitos. O Git faz uso desse algoritmo de encriptação para identificar os arquivos de forma segura e rápida.



| **openssl sha1 <nome\_arquivo>** - Específica que quero usar o ‘*sha1’* (algoritmo de encriptação) e gera um conjunto de 40 caracteres.

**Exemplo no terminal do Git Bash:**

**Abrindo o terminal:** Abra a pasta desejada > Clicar com o botão direito > Git Bash Here

**openssl sha1 texto.txt**

Código identificador gerado: 3515914c5f6a495183e476c516a0d0223217eef8

**Abra o arquivo, substitua alguma informação e salve**

**openssl sha1 texto.txt**

Código identificador gerado: 32028a9e8409ebb772c59f9b468bdbb3f2f52cf1

**Abra novamente o arquivo, desfaça a alteração e salve**

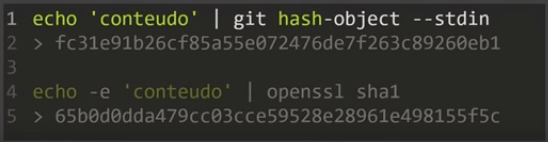
**openssl sha1 texto.txt**

Código identificador gerado: 3515914c5f6a495183e476c516a0d0223217eef8

Esse exemplo mostra uma forma muito eficiente do Git garantir e identificar que os arquivos sofreram modificação ou que não sofreram modificação de uma versão para outra.

**Aula 3 - Objetos Internos do Git**

O *sha1*não gera identificação somente para arquivos, mas também para objetos externos do Git. São eles (objetos): **Blobs, Trees** e **Commits** são os objetos básicos do Git e responsáveis pelo versionamento do nosso código.

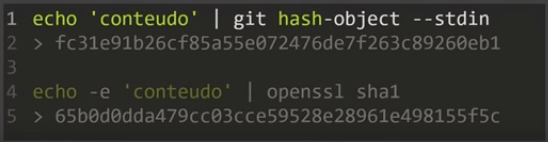


**echo ‘conteudo’** - Vai pegar a string

A output da função vai para uma função do Git:

**git** **hash-object** -

**–stdin** - Serve para que esta função entenda que estamos enviando um texto



Esse conjunto de 40 caracteres diferentes ocorre pois os arquivos ficam guardados dentro do objeto chamado **Blob**, e este objeto contém metadados nele.Conteúdo do objeto: tipo, tamanho, ‘\0’ e o conteúdo do arquivo.



Passando o comando com as características do objeto, o código ficará igual:



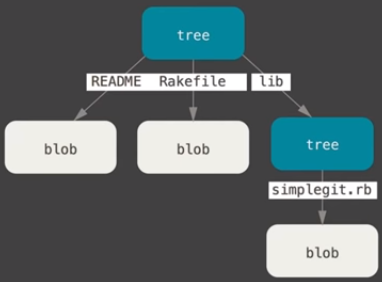
A partir do momento que conhecemos o que o *sha* faz, a gente vai entender que o Git vai guardar esses arquivos, gerando a encriptação deles, mas sabendo que o Git armazena metadados nesses objetos.

As **Tree** Armazenam Blobs, então é uma crescente. A blob sendo o bloco básico de composição, a Tree armazenando e apontando para tipos de blobs diferentes. A Tree também contém metadados, ela tem a ‘\0’, aponta para um Blob e guarda o nome do arquivo. A Tree será responsável por montar toda a estrutura de onde estão os arquivos. Ela aponta tanto para Blobs como para outra Tree.

Diretórios em um sistema operacional podem conter outros diretórios, por isso faz sentido que o Git use um tipo de objeto recursivo.

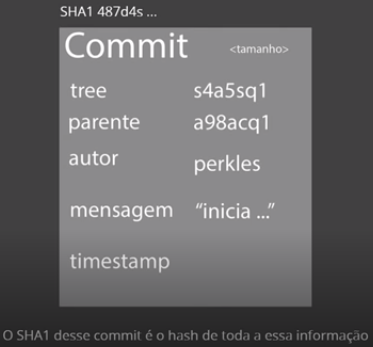


Exemplificando, de forma diferente, uma Tree:

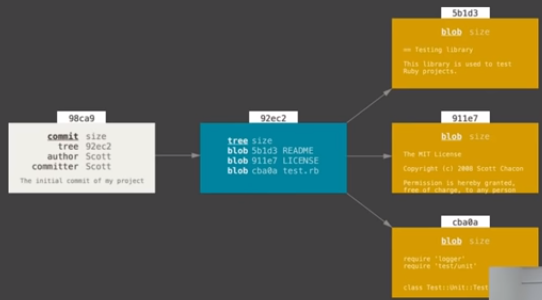


Temos as *trees* que contêm informações de quais arquivos estão apontando (README, Rakefile e lib), apontam também para *blobs* ou outras *trees* que apontam para outras blobs. O blob é um objeto que encapsula esse comportamento de diretórios e é usado para apontar para diretórios que contém arquivos e consequentemente para arquivos também.

Por fim, temos o objeto mais importante de todos, que é o **commit**, é o objeto que vai juntar tudo, o que vai dar sentido para a alteração feita. O *commit* aponta para uma *tree*, aponta para um parente (último *commit* realizado antes dele), aponta para o autor e aponta para a mensagem também. Então o autor e a mensagem fazem parte da ideia de “fazer sentido”.



Os arquivos, representados pelos *blobs*, as pastas, representadas pelas *trees*, significam uma alteração. Então podemos escrever uma mensagem no objeto chamado *commit*, que vai explicar e dar significado para todos esse monte de arquivo dentro desse monte de pasta, esse objeto também leva o nome de um autor. O *commit* também tem um *timestamp* (carimbo de tempo), então esse objeto leva o horário e data de criação. os *commits* também possui *sha1* (encriptação), isso significa que se alteramos uma *blob* ou um dado de um arquivo, vai ser gerado um *sha1* dessa *blob*, essa *blob* tem uma *tree* apontando para ela, então se alterou o *sha1* da *blob*, altera o *sha1* da *tree* e o *commit* aponta para uma *tree*, consequentemente o *sha1* da *commit* será alterado. Por isso o Git é tão confiável, pois quando temos um *commit*, está garantido que ninguém alterou aquele *commit*, pois é formado uma linha do tempo (histórico) de alteração do *commit*. Exemplo:

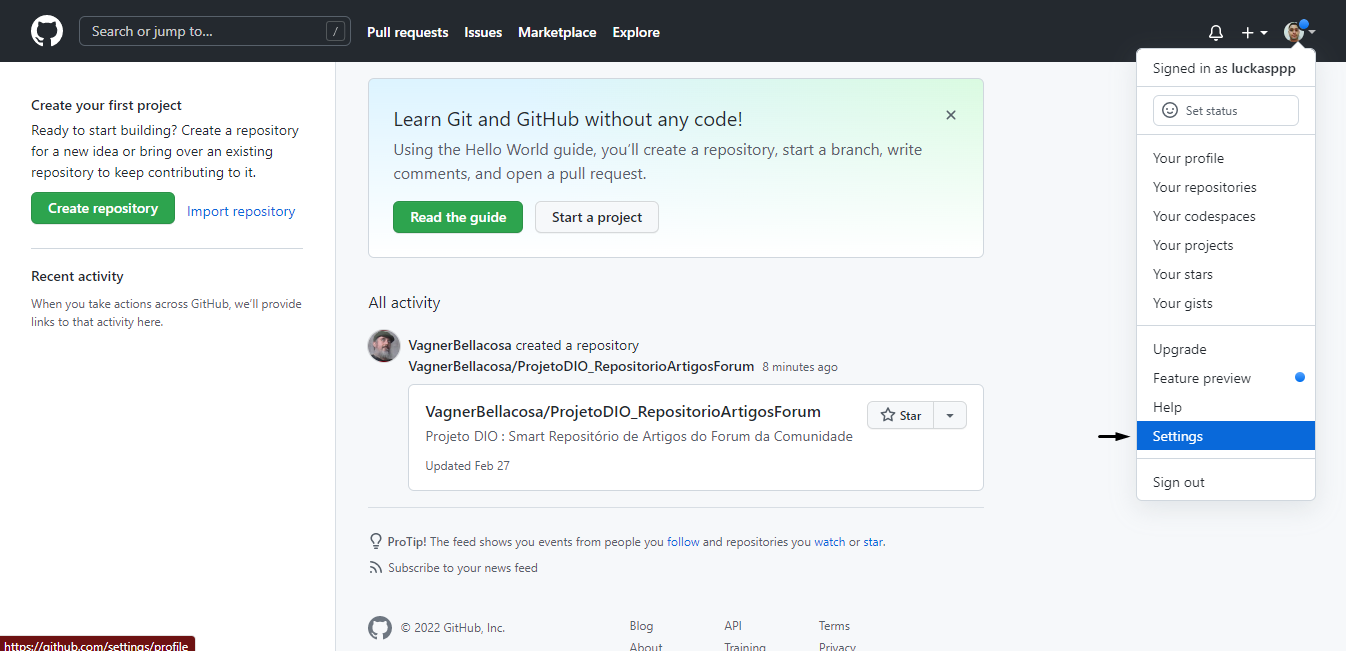


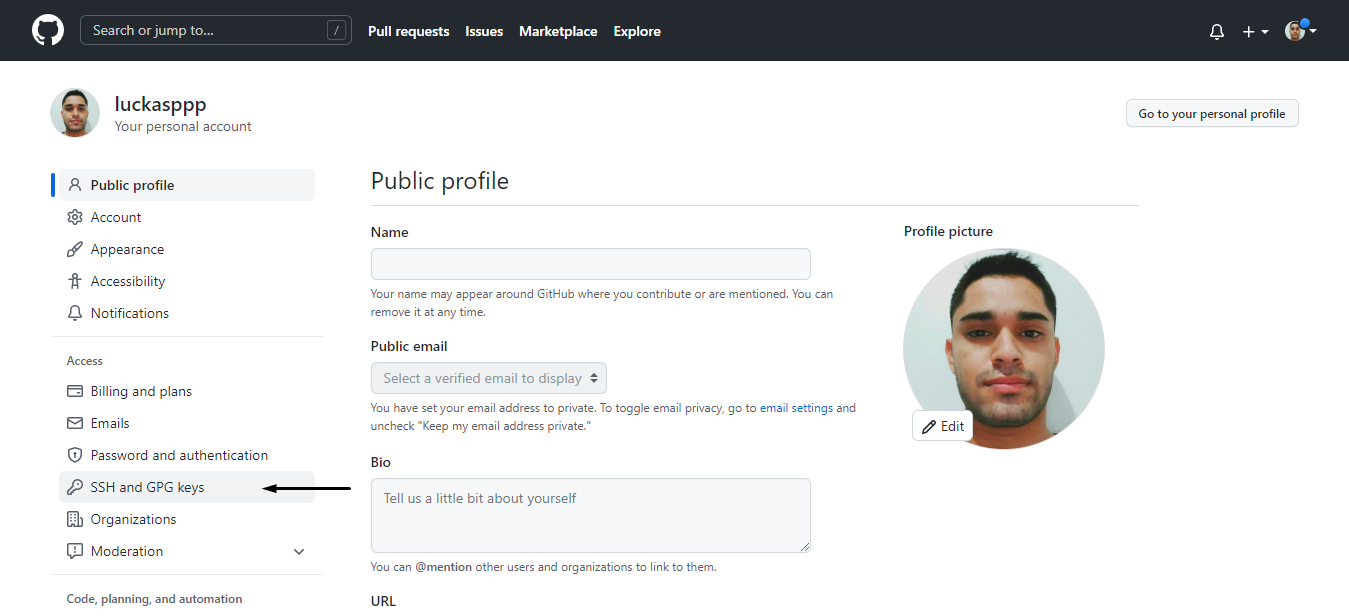
O Git é um **Sistema Distribuído e Seguro** pelo fato dos *commits* serem muito difíceis (impossíveis) de serem alterados, tanto a versão mais recente que está na máquina do servidor como qualquer versão que esteja dentro de outras máquinas de pessoas que mantém o sistema, também são confiáveis. Se der um problema na nuvem do Github e o código deixar de existir, seria preciso acontecer algo com essas pessoas que contém as versões nas máquinas.

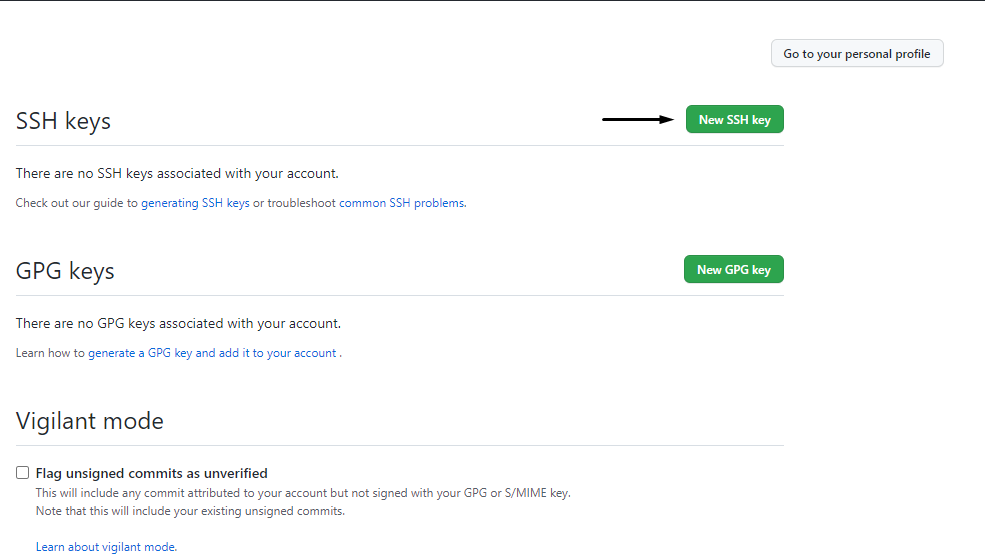
**Aula 4 - Chaves SSH e Tokens**

**Chave SSH** é uma forma de estabelecer uma conexão segura e encriptada entre duas máquinas (servidor do Github x máquina do usuário). Então configuramos nossa máquina local como uma máquina confiável pro Github, estabelecendo essa conexão com duas chaves, uma pública e uma privada. Vamos pegar essa chave pública e por no Github e dessa forma o Github vai reconhecer a nossa máquina, então todos os repositórios que tiver na máquina do usuário por esse processo SSH, quando enviar o código, o Github vai reconhecer a assinatura da máquina do usuário, vai ter uma conexão previamente estabelecida e será possível enviar código sem precisar colocar login e senha.

Passo a passo para essa configuração:







Agora vamos abrir o Git Bash para digitar a sequência de comandos para gerar esse par de chaves:

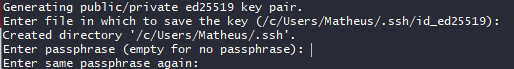
**ssh-keygen -t ed25519 -C <e-mail\_do\_usuário>**

****

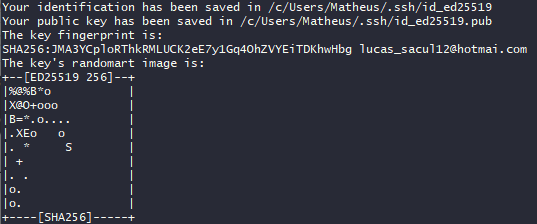
Obs:

* Quando uma pasta começa com ‘**.**’ é porque ela é uma pasta oculta.
* id\_ed25519’ é o nome da chave que está gerando.

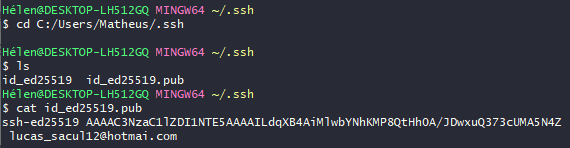
Agora vai pedir para criar uma senha e repetir:



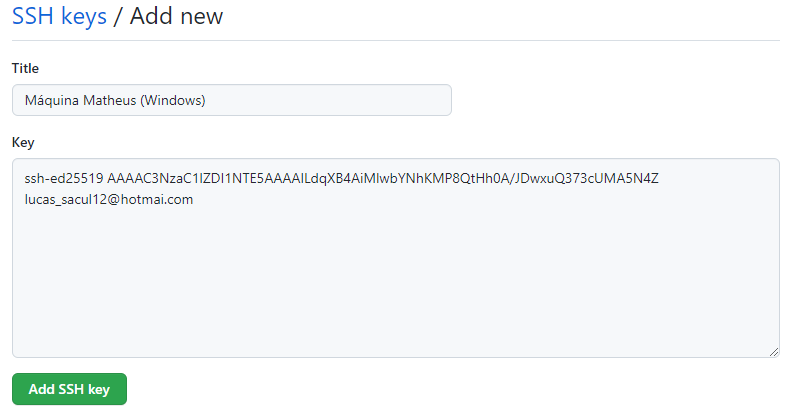
Mostrará onde a identificação e a chave pública foram salvas, dá um *fingerprint* (uma espécie de identidade das chaves), nome e tipo da criptografia. O resultado:



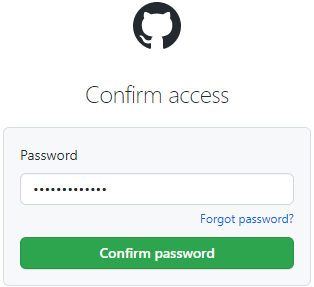
Vamos agora abrir a pasta onde as chaves estão alocadas, expor por meio de comando a chave pública (id\_25519.pub) e copiar para colocar no Github.



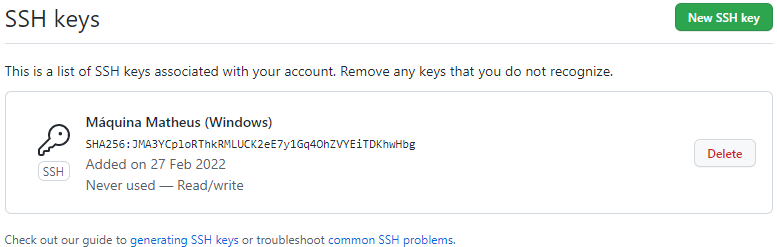
Copiando a chave colocando no Github:



Um título precisa ser colocado para identificar a máquina, recomenda-se colocar “Máquina <nome\_do\_Dono\_da\_Máquina> e <Nome\_Sistema\_Operacional>”, apertar “Add SSH key” e colocar a senha criada no processo de criação de chave.



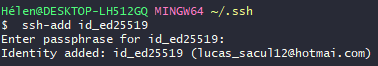
Quando a chave é criada aparecerá na lista de chaves SSH associadas à conta.



É preciso seguir mais uma etapa para que a chave funcione direitinho e tudo fique pronto. Vamos iniciar o **ssh-agent** que é uma entidade encarregada de pegar e lidar com as chaves.



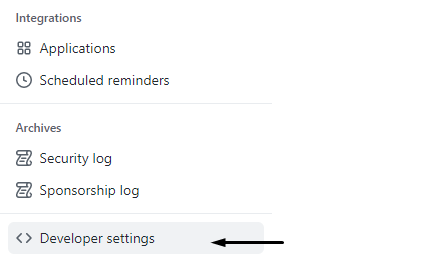
A numeração do *Agent pid* é a numeração de um processo de start de projeto. Agora vamos entregar a nossa **chave privada** para esse *Agent*, pois ele vai usar a nossa chave privada para descriptografar a mensagem. Vai ser necessário informar a senha criada no processo de criação da chave.

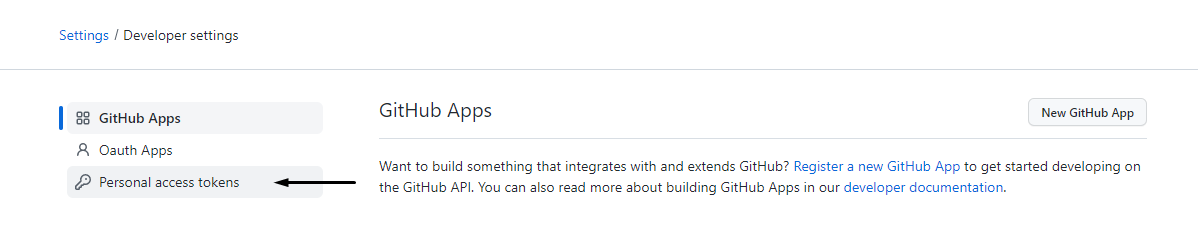


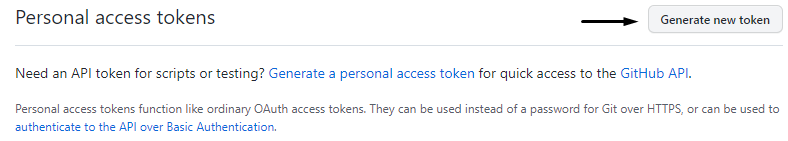
**Informação para o futuro:** Toda vez que for clonar o repositório, quando tem uma chave *ssh* configurado na máquina, não pode simplesmente copiar o link do repositório e usar o Gitclone para poder clonar. Não pode usar a URL, porque essa URL é como se tivesse clonando via https e não é o que queremos. Quando temos um protocolo *ssh* fixo na máquina, devemos usar o caminho *ssh*.

A segunda forma de autenticação segura que o Github nos trás é o **Token de Acesso Pessoal**, o início é bem parecido, mas ele se associa mais ao processo de colocar o *nickname* e a senha, do que ter a vantagem de que o *shh* tem de não requerer senha.

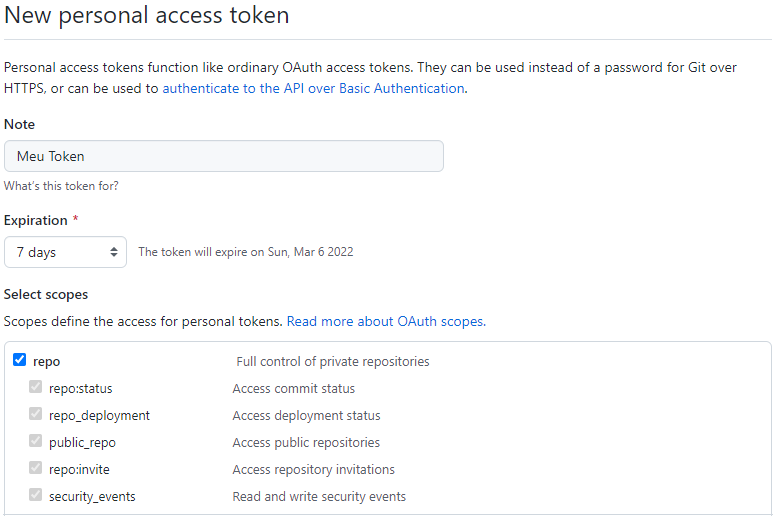
No caso do Token de acesso pessoal, vamos gerar um token no Github, guardar na máquina ou em algum lugar, e sempre que for fazer o commit, o Git vai pedir o usuário e a senha, e na hora da senha, usamos o Token de acesso pessoal. Vamos ver como fazer:



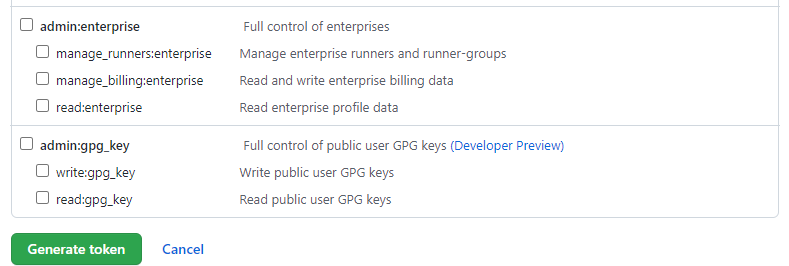




Após esse processo podemos configurar o Token em si. Nele podemos configurar um título para o token, uma data de expiração. Vamos marcar a opção ‘repo’:



Vamos rolar a página até o final e apertar ‘Generate Token’.



Após criado, aparecerá a página com o token. É muito importante copiar e salvar em algum lugar o código que aparece, pois não aparecerá novamente.



**Aula 5 - Primeiros Comandos com o GIT**

Vamos aprender a iniciar um repositório Git, dar início de fato ao versionamento de código e criar o primeiro *commit*.

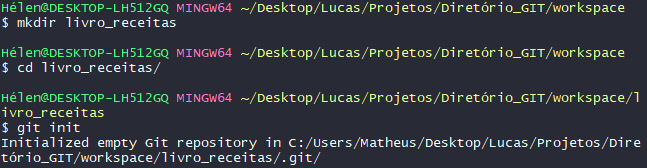
**git init** - Inicia o repositório do Git.

**git add** - Para mover arquivos.

**git commit** - Cria um commit.

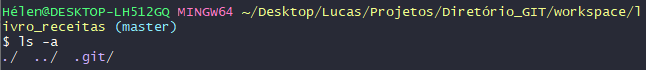
Quando estamos lidando com o terminal, colocamos primeiramente o nome do programa na frente, pois estamos chamando pelo terminal comandos específicos do Git, e por isso todo comando do Git leva o nome dele na frente.

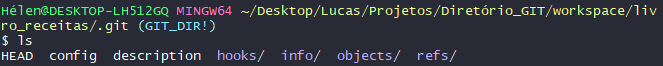
Criando um repositório:



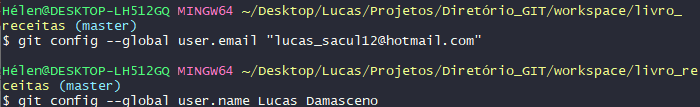
Dentro da pasta que criamos, foi criado uma pasta chamada **.git** que é uma pasta oculta e gerencial do Git, e para exibi-la, precisamos usar um *flag* específica:

**-a** - Mostrar itens ocultos.



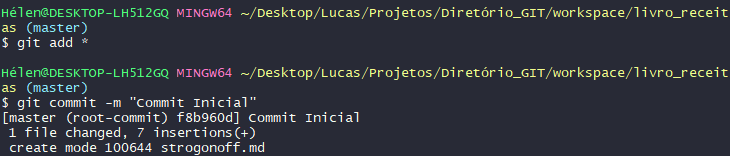
Verificamos dentro da pasta **.git** e podemos ver que temos estruturas referentes ao próprio Git e uma pasta chamada ‘objects’, pasta onde o Git armazena os objetos.

Precisamos fazer umas configurações iniciais (no primeiro uso do Git), para informarmos um username e um e-mail, informações de autor, que serão atrelados à alteração/criação dos *commits*.



Adicionando um arquivo dentro da pasta, para isso vamos utilizar um tipo de arquivo chamado **Markdown** (arquivo.md), que é a forma mais humana de escrever um arquivo HTML, e por sua vez, o arquivo HTML é o esqueleto (estrutura básica) de qualquer página na web.

Vamos aprender a adicionar um *commit*:



**git add <nome\_arquivo.txt>** - Adiciona individualmente o arquivo mencionado ao repositório.

**git add .** ou **git add \*** - Adiciona todos os arquivos existentes ao repositório.

**git commit -m “comentário”** - Identifica e armazena o container no repositório.

**Aula 6 - Ciclo de Vida dos Arquivos**

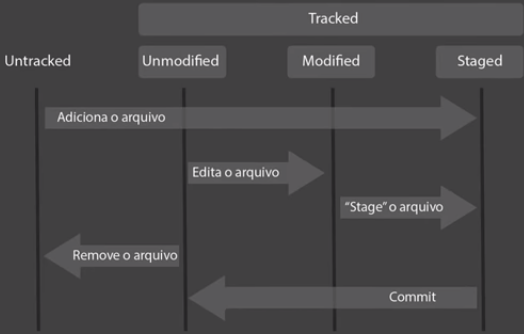
**git init** - Inicializa o conceito do git de repositório o cria dentro da pasta/diretório definida.

Dentro de **Tracked**, dentro dos arquivos que são rastreáveis e que são conhecidos pelo Git, pode subdividir-se em três estágios diferentes: **Unmodified**, **Modified** e **Staged**. Temos outro agrupamento chamado **Untracked**, são arquivos que o git ainda não tem conhecimento.

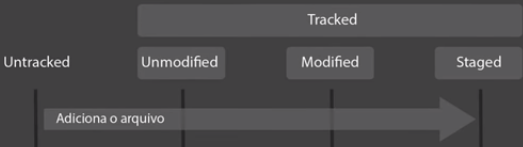
**Unmodified** - É o arquivo que ainda não foi modificado.

**Modified** - É o arquivo que sofreu alguma modificação.

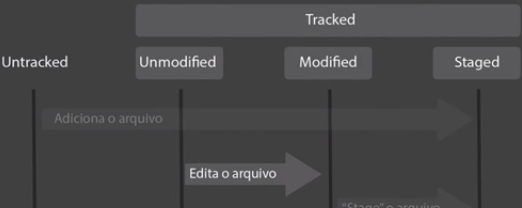
**Staged** - É onde ficam os arquivos que estão se preparando para fazer parte de outro tipo de agrupamento.



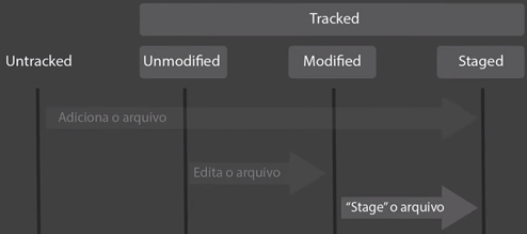
Temos um arquivo que estava *Untracked* (arquivo que acabou de ser criado). Quando roda o segundo comando **git add**, o arquivo é movido de *Untracked* para *Staged* (área que está esperando)*.*



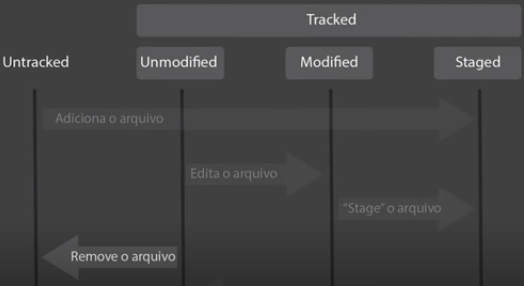
Nos arquivos **Unmodified**, temos um arquivo, dentro do repositório do Git, que ainda não sofreu nenhuma modificação. Quando abrimos o arquivo e modificamos algo nele, automaticamente vira **Modified**. Isso é feito quando o Git compara o *sha1* dos arquivos e vai descobrir que tem modificação.



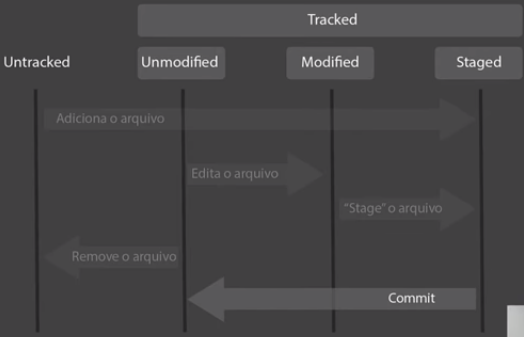
Se rodarmos o **git add** novamente nesse arquivo **Modified,** ele vai para **Staged**, então ele entra para essa fase de espera, para entrar em ação.



Quando temos um arquivo **Unmodified** e removemos o arquivo, ele vira **Untracked**.



Quando movemos um **Unmodified** ou **Modified** para o **Staged,** o arquivo está se preparando para fazer parte de um *commit.* O *commit* é um objeto chave do Git. Então quando colocamos esses arquivos no **Staged** e damos um *commit* neles, ou seja, envelopamos todas as modificações com significância, escrevemos uma mensagem pro *commit*, essa mensagem carrega um autor, uma data e integram de fato o objeto *commit*, deixam de ser **Staged** e se tornam *commit* e retorna todos esses arquivos para um **Unmodified** e aí começa o ciclo de novo.



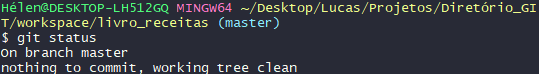
Vamos entender o que o **repositório** significa. Temos dois tipos de ambiente, o de **Ambiente de Desenvolvimento**, que representa tudo o que está na nossa máquina, e temos o **Servidor**.



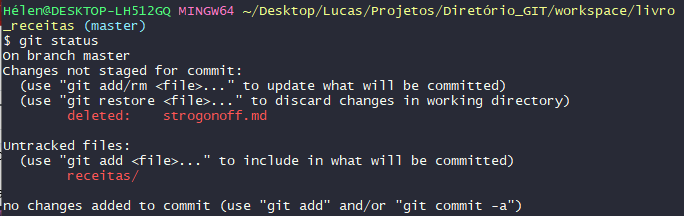
O Git é um sistema distribuído, então vai ter a versão dele no servidor, no caso o Github, e tem a versão que está na máquina local. Então as alterações que fazemos na máquina local não repercutem imediatamente na versão do repositório remoto, a não ser que executamos um conjunto de códigos específico para empurrar a alteração do repositório local para o remoto.

Vamos aprender na prática como esses conceitos são aplicados.

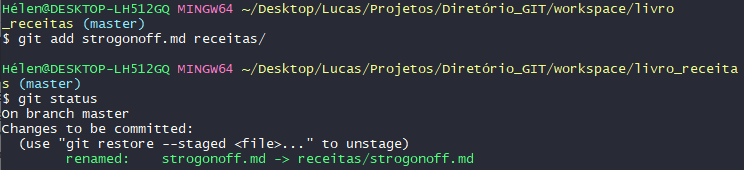
**git status** - Permite vermos o status dos arquivos (untracked, unmodified, modified, staged).



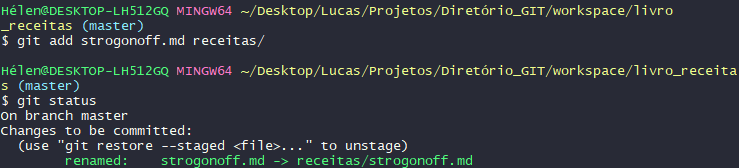
Feitas algumas manipulações de arquivo para outra pasta e colocando novamente o comando **git status**, vemos o seguinte resultado:



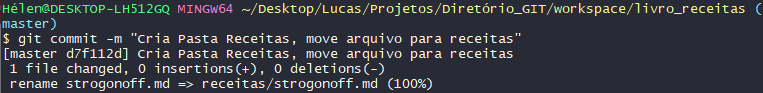
A imagem mostra as informações obtidas por meio da função. Nos informa que o arquivo foi deletado da pasta (antiga), colocou a pasta criada com status *Untracked* e que esta ainda não faz parte do *commit. Ao* ver do sistema operacional, parece que o arquivo foi excluído, porém foi movido. Quando o git não sabe de onde um arquivo/pasta veio, ele põe no estágio *Untracked* e o Git dá umas sugestões do que pode ser feito.



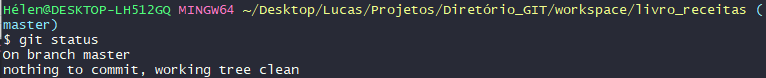
Após adicionar o arquivo à pasta, usamos a função **git add** para que fiquem no status *Staged*.

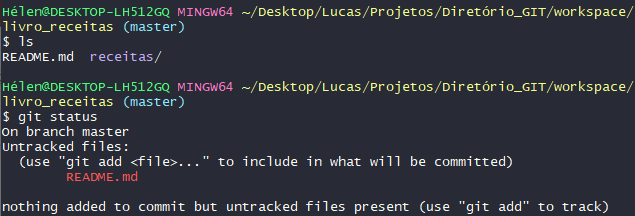


Obs.: Os arquivos que antes tinham a cor vermelha, agora tem a cor verde. Isso significa que os arquivos estão com status *Staged* e agora estão prontos para se tornarem *commits*.

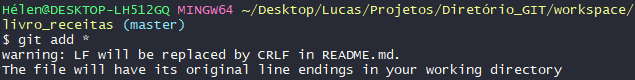


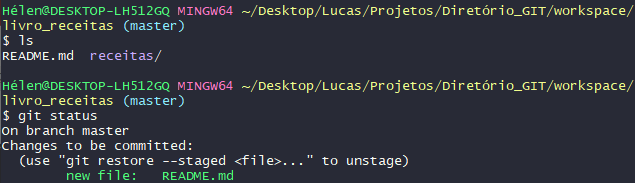
*Commit* criado e agora com a função **git status** aparece que não há nada para fazer *commit*.

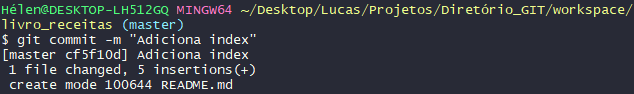




Modificamos o arquivo ‘README.md’. e adicionamos ao *Staged* com a função **git add \***.







**mkdir <nome\_pasta>** - Cria uma pasta.

**mv <nome\_arquivo> ./<nome\_nova\_pasta>** - Move o arquivo para dentro da pasta selecionada.

**git add <nome\_arquivo>** **<nome\_da\_pasta>** - Move o arquivo para o repositório local..

**git add \*** - Adiciona todos os arquivos da pasta à *Staged* (repositório local).

**git add .** - Adiciona todos os arquivos da pasta à *Staged* (repositório local).

**git restore –staged <arquivo>** - Para restaurar de Staged para Unstaged (Untracked).

**git commit -m “comentário”** - Cria o *commit*.

**echo > <arquivo>** - Cria um arquivo.

**git config –list** - Lista as configurações feitas no Git. (letra ‘q’ para sair)

**Aula 7 - Trabalhando com o GitHub**

**git config –global –unset user.email** -Reseta a configuração de e-mail.

**git config –global –unset user.name** -Reseta a configuração de nome.

**git config –global user.email** -Define um e-mail de usuário.

**git config –global user.name** -Define um nome de usuário.

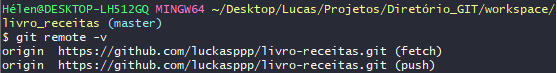
Além de ser um repositório remoto onde vamos guardar os códigos, trabalhar em equipe, outras pessoas terão acesso ao seu código e outras funcionalidades. É acima de tudo uma rede social onde podemos ver o código produzido de outras pessoas, ver em quais projetos estão engajadas, entre outros.

Empurrando a versão do repositório local para uma versão remota.

**git remote add <origin> <link\_repositório\_remoto>** - Cadastra um repositório remoto.

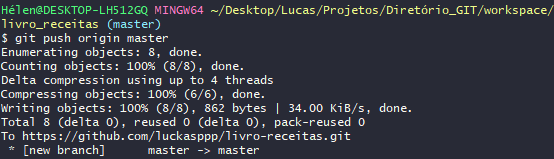
****

**git remote -v** - Lista os repositórios remotos cadastrados.



**origin** - É apenas um apelido, mas por convenção é colocado dessa forma.

**git push origin master** - Empurra o repositório local para o repositório remoto.



Obs.:Master é uma **branch**

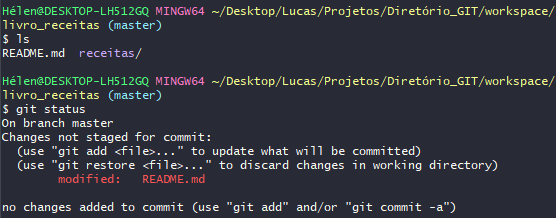
**Aula 8 - Como os Conflitos acontecem no GitHub e como Resolvê-los**

Conflitos irão acontecer em vários momentos da nossa vida profissional, porém não serão difíceis de resolver.

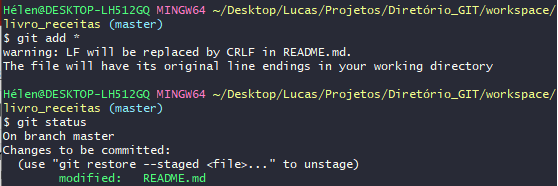
Vamos aprender a criar um **repositório local**  e usar comandos fundamentais do Git. Quando temos um código no *GitHub,* outras pessoas podem acessar, sugerir modificações e podem clonar. Quando baixamos o código do *GitHub* para a nossa máquina ou acabamos de enviar ele para o *GitHub*, temos exatamente a mesma versão de código nos dois repositórios. Quando outra pessoa vai lá no *GitHub* e faz um clone dele, nesse momento os dois estão sincronizados (considerando que ninguém mexeu no código). Quando os dois abrem o mesmo arquivo e fazem uma edição na mesma linha (considerando que não vão escrever exatamente a mesma coisa), as versões estão diferentes e é nesse momento que acontece o problema, quando a edição é feita exatamente na mesma linha. E o código desse outro editor é enviado novamente para o GitHub, neste momento o código que está atualizado no *GitHub* é o código do outro editor e o na nossa máquina local está desatualizado. Quando quisermos empurrar o código na máquina local (desatualizado), o *GitHub* vai pedir para puxarmos o código do *GitHub*, juntar a modificação do outro editor com a nossa modificação e empurrar para o *GitHub*. Nesse momento ocorre o “**Conflito de Merge**”.

**Conflito de Merge** - Ocorre quando o *GitHub* tenta juntar duas alterações que contém alterações na mesma linha. Ele não vai tentar fazer nenhuma alteração drástica, não vai inserir nada. Vai deixar que você mesmo abra o arquivo, resolva o conflito e empurre para o *GitHub*.

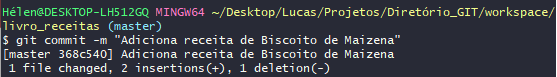
Exemplo: Vamos abrir o arquivo “README.md”, fazer uma modificação nele e visualizar com o comando **git status** (assumindo que o GitHub contém um arquivo alterado na mesma linha por outro usuário).



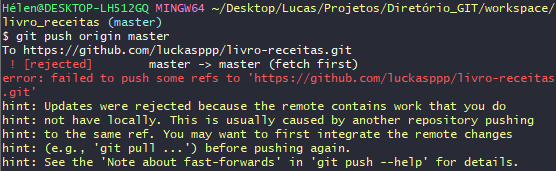
Precisamos adicionar o arquivo ao status de **Staged** para poder fazer o *commit*.



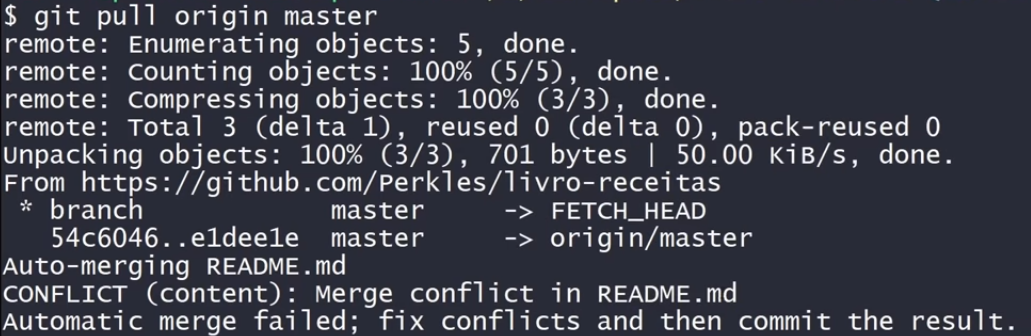
Agora o arquivo está pronto para ser “commitado”.



Agora vamos fazer um *push* (empurrar), que é enviar o código para o *GitHub*.



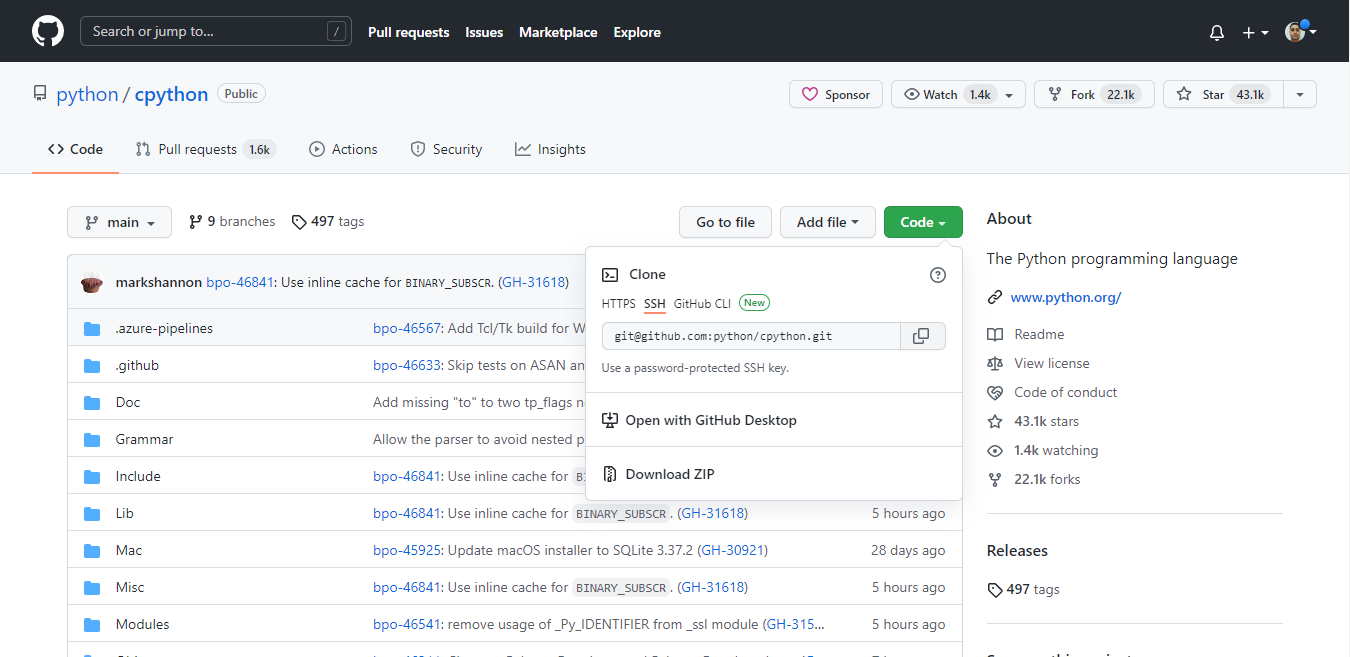
Deu erro de “**conflito de merge**”. Vamos resolvê-lo com o **pull** e depois com **push**.



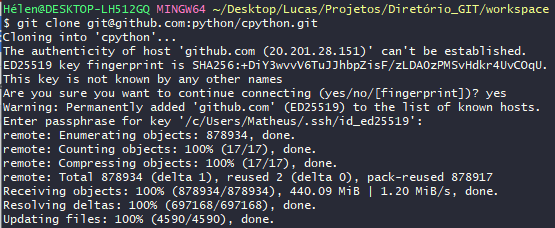
Agora vamos editar e ver as edições relevantes para poder empurrar (push) novamente o arquivo para o *GitHub*.

**Clonando um repositório do *GitHub***

Primeiramente vou em algum repositório no *GitHub* e coloco pra copiar o código.



Depois vou no *Git* e coloco o comando git **clone <código\_copiado>** para clonar o repositório.



O Git cria um repositório local no diretório do projeto. Este repositório local pode ser clonado, posteriormente, para um repositório remoto como o GitHub, por exemplo, e também é possível fazer o contrário, criando um repositório remoto no GitHub e clonando este repositório no computador.

Digitando no Git:

**clear** - Limpa a tela do Git Bash.

**ls** - Lista os arquivos existentes no diretório

**ls -a** - Exibe itens ocultos

**git init** - Cria uma pasta chamada ‘.git’. Este ponto indica que este diretório é oculto. É dentro da pasta ‘.git’ que o conteúdo do projeto vai ser armazenado e identificado.

Obs.: Quando abrimos a pasta **.git** temos outra pasta chamada **objects**. É nessa pasta ‘objects’ que o git vai criar o **Grafo de commits**.

No **Grafo de commits** o git armazena o conteúdo do projeto, representando as mudanças através de um grafo de commits. **Grafo** é uma estrutura que contém objetos relacionados. **Coommits** é um comentário que escrevemos sempre que alguma coisa é modificada no projeto.



Antes de armazenar o conteúdo e formar o *Grafo de Commits*, o *Git* precisa gerenciar as mudanças do projeto. Para isso existe uma espécie de **container** onde devemos armazenar todos os arquivos e pastas que foram criados ou modificados, usamos o **git add** para fazer isso. O comando **git status** verifica o que está armazenado no **container**. O comando **git commit** identifica e armazena este container no repositório local, o **.git** é o repositório e dentro desse repositório vão ser armazenados containers,que são identificados, dentro deles temos o conteúdo do projeto.

**git status** - Informa se tem algum arquivo ou modificação nova para armazenar no repositório (container).

**touch <nome\_arquivo.txt>** - Cria um arquivo no diretório.

**git add <nome\_arquivo.txt>** - Adiciona individualmente o arquivo mencionado ao repositório.

**git add .** ou **git add \*** - Adiciona todos os arquivos existentes ao repositório.

Obs.: Os arquivos que antes tinham a cor vermelha, agora tem a cor verde. Isso significa que os arquivos estão com status *Staged*.

O próximo passo é identificar o container e armazenar no repositório:

**git commit -m “comentário”** - Identifica e armazena o container no repositório.

Obs.: Precisa ficar claro que o repositório **.git** guarda uma cópias dos arquivos.

**git log** - Mostra a identificação do commit, as informações sobre o autor, data e mostra o comentário feito pelo autor.

**git log --oneline** - mostra a identificação do commit com o comentário feito por quem modificou de forma simplificada.