

DIGITAL INNOVATION ONE – GIT E GITHUB

1. INTRODUÇÃO AO GIT

O foco desse curso é ensinar as funcionalidades do Git e as funcionalidades do repositório GitHub. O Git é um sistema de repositório distribuído, criado por Linus Torvalds, que foi quem criou o SO Linux. Linus criou o Git porque ele sentiu a necessidade de um Software que monitorasse diferentes versões de um código e comportasse a colaboração de diversas pessoas. Os programas de versionamento de código da época não possuíam diversas funcionalidades que Linus achava importante.

O Git cuida dessa parte de versionamento de códigos e outros softwares, como o GitHub e o GitLab guardam esses códigos em repositórios de forma online. O GitHub é uma empresa da Microsoft e muito bem estabelecida no mercado e por isso ele aparece muitas vezes lado a lado do Git, porém são softwares diferentes. Ao trabalhar com Git e GitHub nós iremos aprender 5 tópicos importantes no desenvolvimento de software:

1. Controle de Versão
2. Armazenamento em Nuvem
3. Trabalho em Equipe
4. Melhorar o Código
5. Reconhecimento

Nesse curso iremos treinar na prática a utilizar o Git e o GitHub. Vamos desenvolver uma aplicação, que é a criação de um livro de receitas simples. Através da criação desse projeto simples iremos aprender conceitos mais abstratos do Git.

2. Navegação via Command Line Interface e Instalação

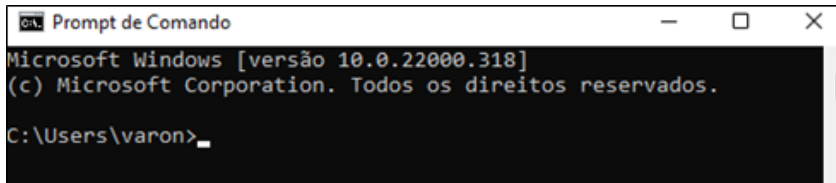
Navegação básica no terminal e instalação

A maioria dos programas operacionais possuem uma interface gráfica, sendo programas do tipo GUI (*graphic use interface*). O Git não possui uma interface gráfica, sendo um *software* do tipo CLI (*command line interface*). Existem programas que pegam o GIT e implementam uma interface gráfica, porém não iremos utilizar esse recurso no curso. Nós iremos interagir apenas com linhas de comando.

Nessa primeira parte aprenderemos a usar comandos comum do terminal, que permitem:

- Mudar de pastas;
- Listar as pastas;
- Criar pastas e arquivos;
- Deletar pastas e arquivos.

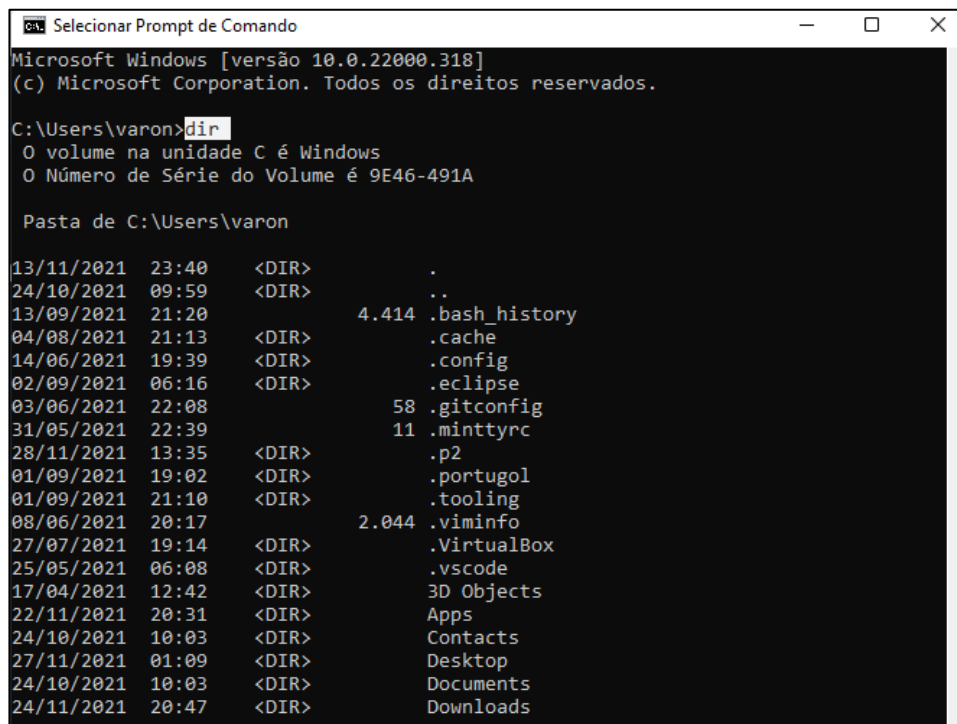
O terminal do Windows é chamado de **Prompt de comando**. Para abrir ele basta pesquisar no Windows por “cmd”.



```
Microsoft Windows [versão 10.0.22000.318]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\varon>
```

- O primeiro comando que iremos aprender é o comando de listar os arquivos e diretórios dentro da pasta que estamos executando o prompt. No cmd digite “dir” e tecele enter.



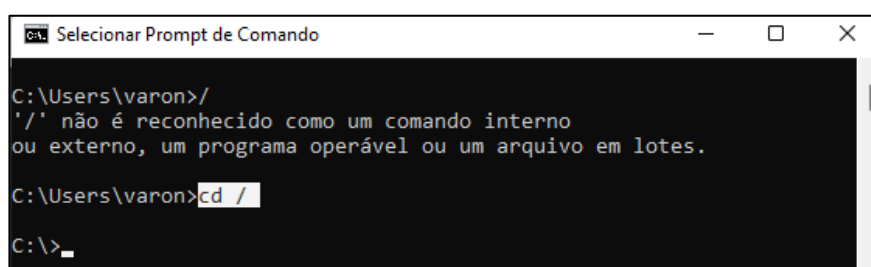
```
Microsoft Windows [versão 10.0.22000.318]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\varon>dir
O volume na unidade C é Windows
O Número de Série do Volume é 9E46-491A

Pasta de C:\Users\varon
13/11/2021  23:40    <DIR>          .
24/10/2021  09:59    <DIR>          ..
13/09/2021  21:20             4.414 .bash_history
04/08/2021  21:13    <DIR>          .cache
14/06/2021  19:39    <DIR>          .config
02/09/2021  06:16    <DIR>          .eclipse
03/06/2021  22:08             58 .gitconfig
31/05/2021  22:39             11 .minttyrc
28/11/2021  13:35    <DIR>          .p2
01/09/2021  19:02    <DIR>          .portugol
01/09/2021  21:10    <DIR>          .tooling
08/06/2021  20:17             2.044 .viminfo
27/07/2021  19:14    <DIR>          .VirtualBox
25/05/2021  06:08    <DIR>          .vscode
17/04/2021  12:42    <DIR>          3D Objects
22/11/2021  20:31    <DIR>          Apps
24/10/2021  10:03    <DIR>          Contacts
27/11/2021  01:09    <DIR>          Desktop
24/10/2021  10:03    <DIR>          Documents
24/11/2021  20:47    <DIR>          Downloads
```

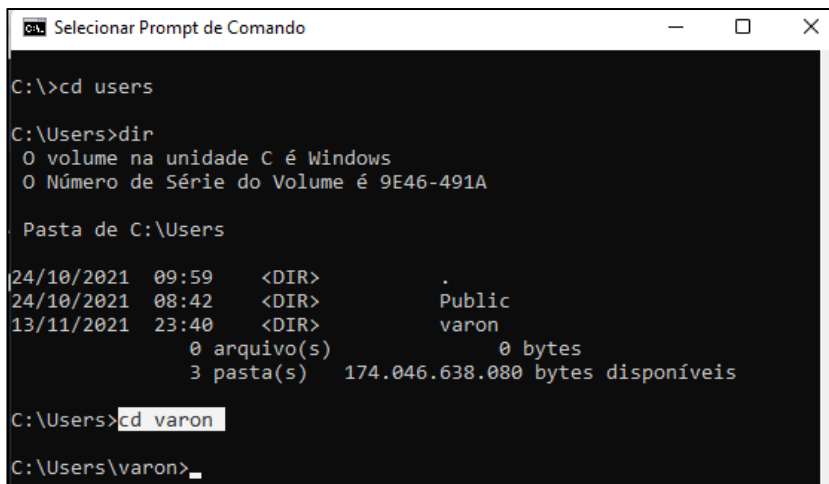
Todos os comandos que iremos aprender apresentam variâncias, ou *flags*, que são complementos que passamos para esses comandos e esse complementos acrescentam, modificam ou formatam a forma que esses comandos são devolvidos.

Depois de aprender o comando para listar os diretórios, vamos ver como subimos ou descemos de níveis nesses diretórios. O próximo comando que iremos aprender é o “cd”, que vai possibilitar que naveguemos entre as pastas. Para mudar de diretório basta digitar o nome do diretório junto do comando e para voltar um diretório, deve-se digitar “..”. Para ir direto na base do diretório. basta com o comando cd, digitar “/”.



```
C:\Users\varon>/
'/' não é reconhecido como um comando interno
ou externo, um programa operável ou um arquivo em lotes.

C:\Users\varon>cd /
C:\>
```



```
C:\>cd users

C:\Users>dir
O volume na unidade C é Windows
O Número de Série do Volume é 9E46-491A

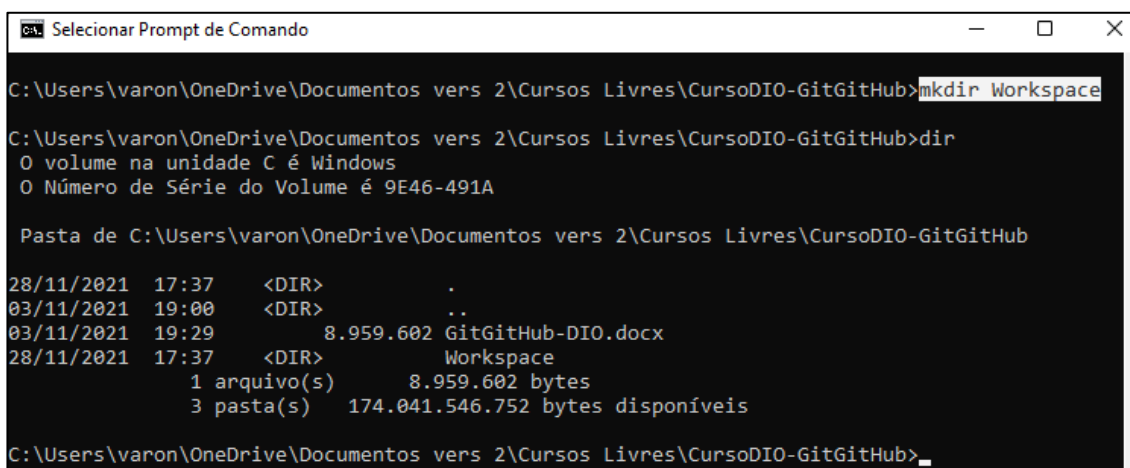
Pasta de C:\Users

24/10/2021  09:59    <DIR>        .
24/10/2021  08:42    <DIR>        Public
13/11/2021  23:40    <DIR>        varon
               0 arquivo(s)          0 bytes
               3 pasta(s)    174.046.638.080 bytes disponíveis

C:\Users>cd varon
C:\Users\varon>
```

Sempre que estamos interagindo com o terminal podemos optar em deixar a tela do terminal limpa. Para isso utilizamos o comando “cls” (*clear scream*). Outra função que facilita o trabalho no terminal é a função de autocompletar, que é executada apenas com a tecla Tab.

Vamos agora navegar entre as pastas no terminal e acessar a pasta do curso de Git e GitHub da DIO. Dentro dessa pasta vamos criar um diretório chamado “workspace”.



```
C:\Users\varon\OneDrive\Documentos vers 2\Cursos Livres\CursoDIO-GitGitHub>mkdir Workspace

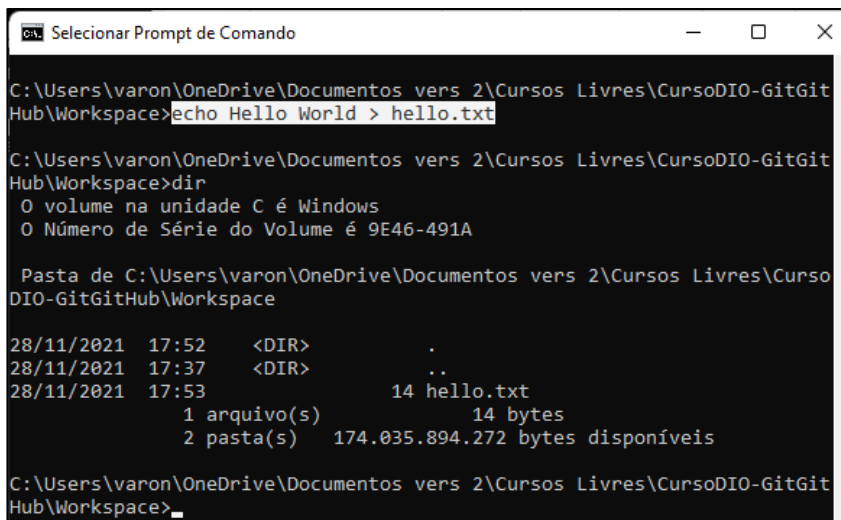
C:\Users\varon\OneDrive\Documentos vers 2\Cursos Livres\CursoDIO-GitGitHub>dir
O volume na unidade C é Windows
O Número de Série do Volume é 9E46-491A

Pasta de C:\Users\varon\OneDrive\Documentos vers 2\Cursos Livres\CursoDIO-GitGitHub

28/11/2021  17:37    <DIR>        .
03/11/2021  19:00    <DIR>        ..
03/11/2021  19:29      8.959.602 GitGitHub-DIO.docx
28/11/2021  17:37    <DIR>        Workspace
               1 arquivo(s)      8.959.602 bytes
               3 pasta(s)    174.041.546.752 bytes disponíveis

C:\Users\varon\OneDrive\Documentos vers 2\Cursos Livres\CursoDIO-GitGitHub>
```

Agora vamos criar alguns arquivos dentro dessa pasta para aprender como deletar arquivos e repositórios. Podemos criar um arquivo txt dentro do terminal mesmo. Vamos criar uma arquivo txt chamado de hello e dentro dele vamos inserir a frase “Hello World”. Para isso, basta digitar o comando “echo Hello World > hello.txt”



```

C:\Users\varon\OneDrive\Documentos vers 2\Cursos Livres\CursoDIO-GitGit\Workspace>echo Hello World > hello.txt

C:\Users\varon\OneDrive\Documentos vers 2\Cursos Livres\CursoDIO-GitGit\Workspace>dir
O volume na unidade C é Windows
O Número de Série do Volume é 9E46-491A

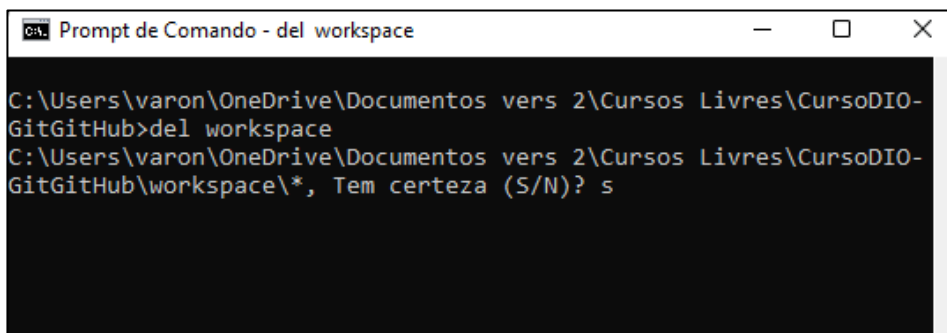
Pasta de C:\Users\varon\OneDrive\Documentos vers 2\Cursos Livres\CursoDIO-GitGit\Workspace
28/11/2021  17:52    <DIR>          .
28/11/2021  17:37    <DIR>          ..
28/11/2021  17:53                14 hello.txt
               1 arquivo(s)                14 bytes
               2 pasta(s) 174.035.894.272 bytes disponíveis

C:\Users\varon\OneDrive\Documentos vers 2\Cursos Livres\CursoDIO-GitGit\Workspace>

```

Perceba na imagem acima que o arquivo hello.txt foi criado com sucesso. O termo *Silence on success* é outro conceito onde se acontecer tudo certo o programa fica em silêncio. Para verificar se o arquivo foi realmente criado, basta digitar o comando “dir” para listar os arquivos.

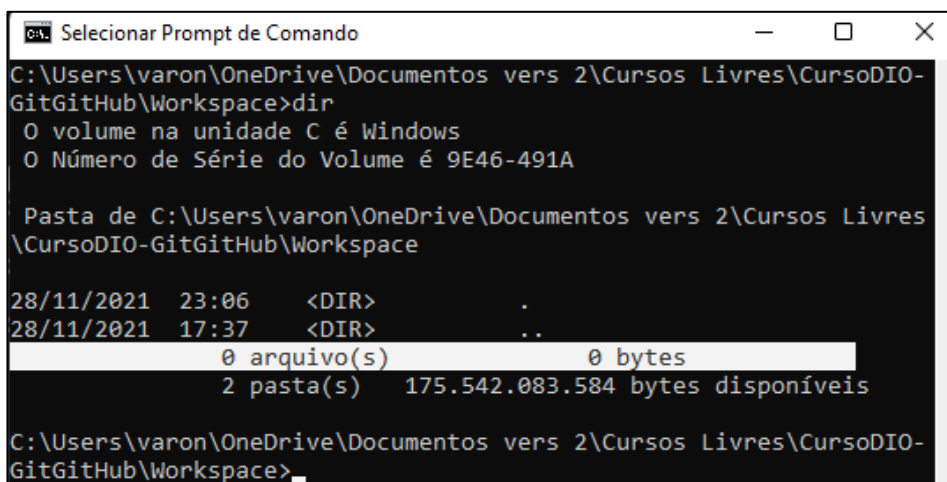
- Para apagar todos os arquivos que estão dentro de um diretório usamos o comando “del”. Porém esse comando não irá apagar o diretório.



```

C:\Users\varon\OneDrive\Documentos vers 2\Cursos Livres\CursoDIO-GitGit\Workspace>del workspace
C:\Users\varon\OneDrive\Documentos vers 2\Cursos Livres\CursoDIO-GitGit\workspace\*, Tem certeza (S/N)? s

```



```

C:\Users\varon\OneDrive\Documentos vers 2\Cursos Livres\CursoDIO-GitGit\Workspace>dir
O volume na unidade C é Windows
O Número de Série do Volume é 9E46-491A

Pasta de C:\Users\varon\OneDrive\Documentos vers 2\Cursos Livres\CursoDIO-GitGit\Workspace
28/11/2021  23:06    <DIR>          .
28/11/2021  17:37    <DIR>          ..
               0 arquivo(s)                0 bytes
               2 pasta(s) 175.542.083.584 bytes disponíveis

C:\Users\varon\OneDrive\Documentos vers 2\Cursos Livres\CursoDIO-GitGit\Workspace>

```

- Para apagar diretamente uma pasta e todos os arquivos dentro dele usamos o comando “rmdir” e o nome da pasta a ser deletada.

```

C:\Users\varon\OneDrive\Documentos vers 2\Cursos Livres\CursoDIO-GitGitHub>rmdir workspace

C:\Users\varon\OneDrive\Documentos vers 2\Cursos Livres\CursoDIO-GitGitHub>dir
O volume na unidade C é Windows
O Número de Série do Volume é 9E46-491A

Pasta de C:\Users\varon\OneDrive\Documentos vers 2\Cursos Livres\CursoDIO-GitGitHub

28/11/2021  23:12    <DIR>          .
03/11/2021  19:00    <DIR>          ..
03/11/2021  19:29                8.959.602 GitGitHub-DIO.docx
               1 arquivo(s)            8.959.602 bytes
               2 pasta(s)  175.543.365.632 bytes disponíveis

```

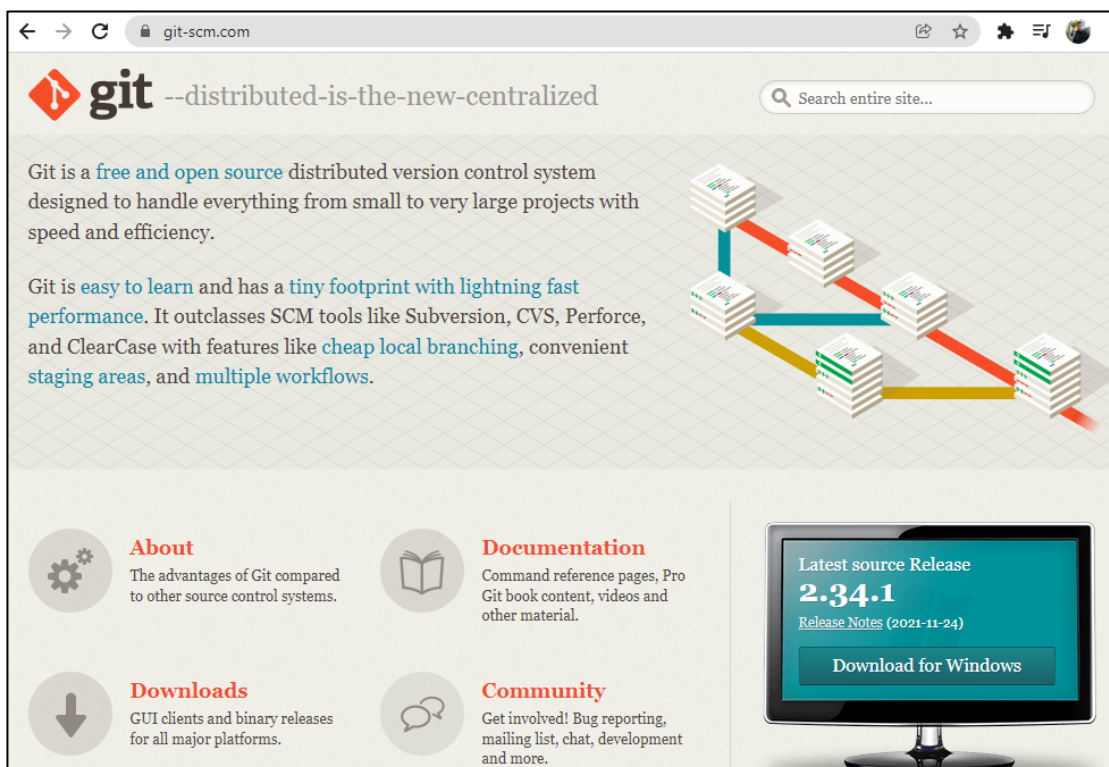
RESUMINDO

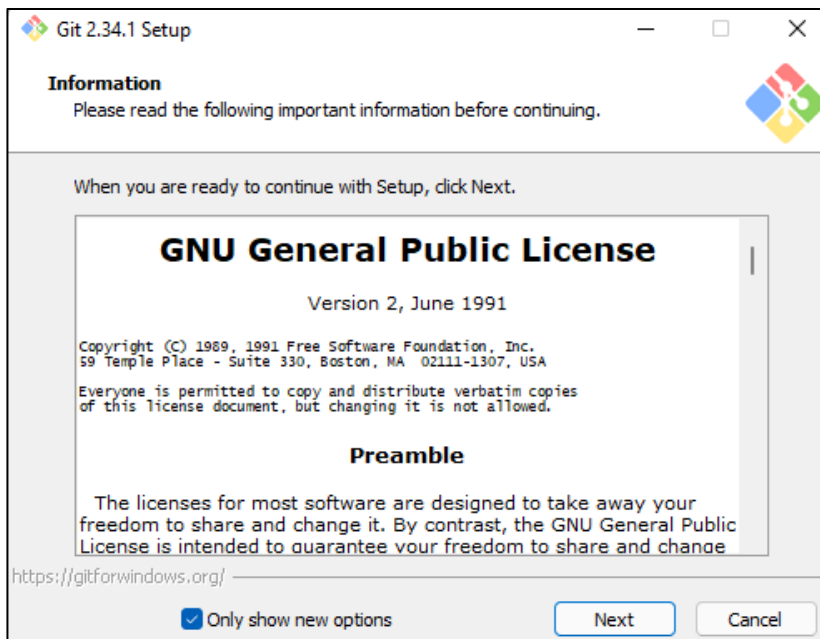
Comando básico para aprendermos a navegar pelo terminal:

- **cd**: muda de diretório para diretório.
- **dir**: lista os diretórios e arquivos contidos dentro da pasta.
- **mkdir**: comando que cria diretórios.
- **del / rmdir**: deleta permanentemente os arquivos e diretórios.

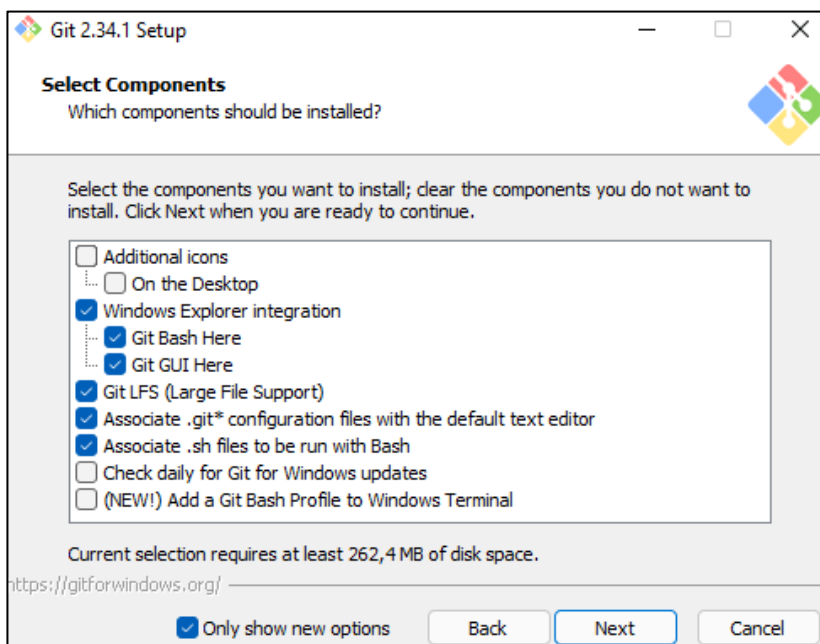
Realizando a Instalação do GIT

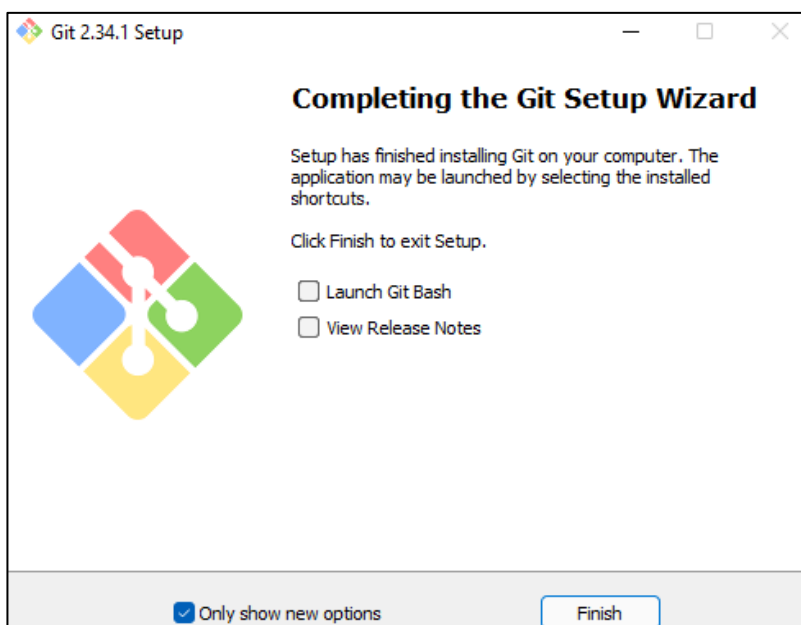
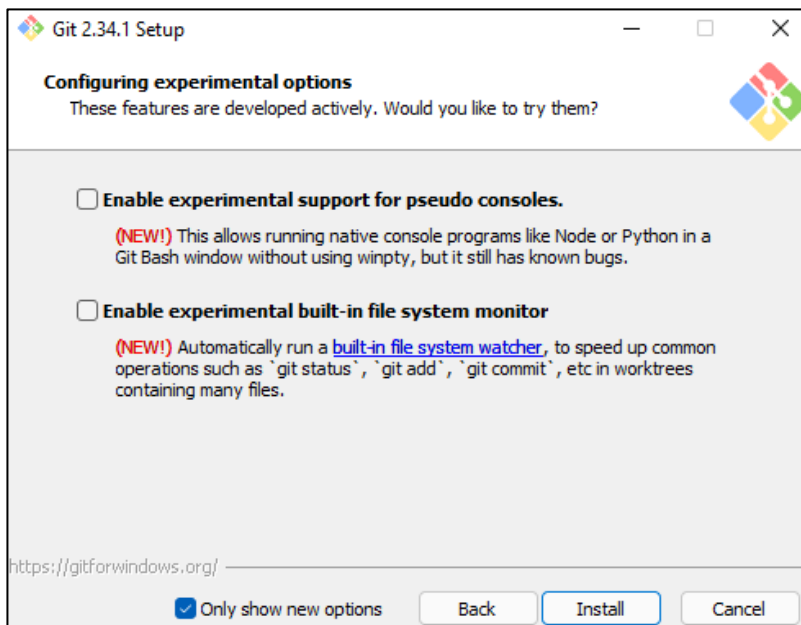
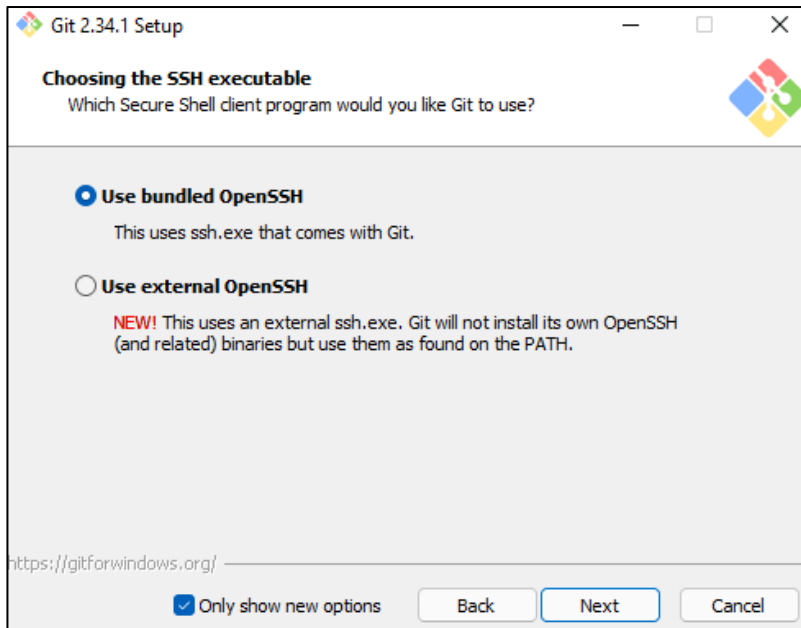
O link do site para baixarmos o Git é o <https://git-scm.com/>. Basta escolher a versão de acordo com o SO da máquina. Clique no botão “Download for Windows que irá abrir uma janela para escolher onde o instalador será baixado.





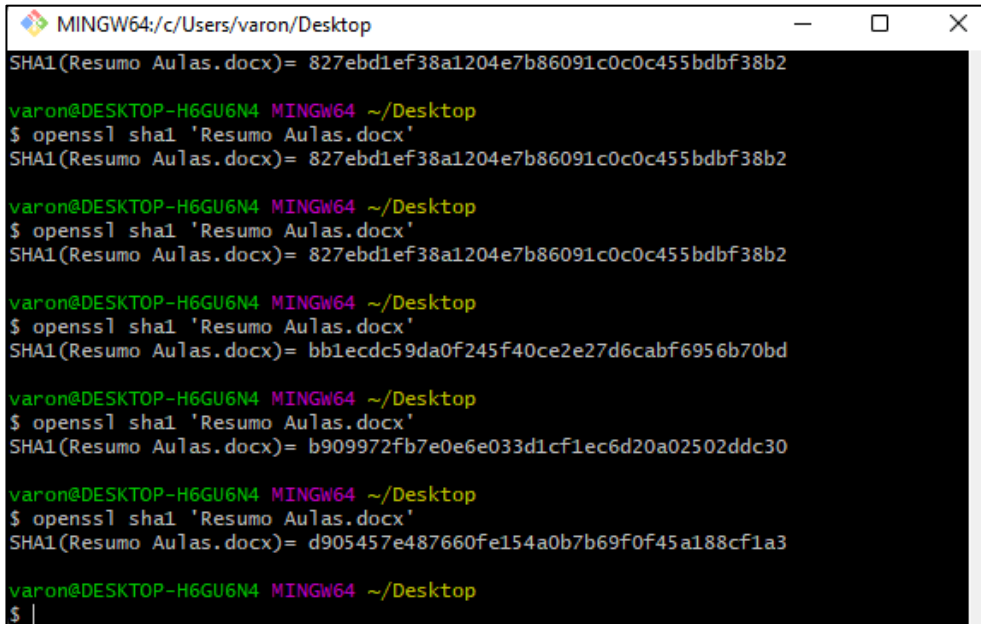
- Certifique-se que as opções “Git Bash Here” e “Git GUI Here” estão marcadas.





3.Entendendo como o Git Funciona por Baixo dos Panos

- **SHA1:** é a sigla para *Secure Hash Algorithm* (Algoritmo de Hash seguro). É um conjunto de funções hash criptográficas projetadas pela NSA (Agência de Segurança Nacional dos EUA). Essa encriptação gera um conjunto identificador de caracteres de 40 dígitos.



```

MINGW64:/c/Users/varon/Desktop
SHA1(Resumo Aulas.docx)= 827ebd1ef38a1204e7b86091c0c0c455bdbf38b2

varon@DESKTOP-H6GU6N4 MINGW64 ~/Desktop
$ openssl sha1 'Resumo Aulas.docx'
SHA1(Resumo Aulas.docx)= 827ebd1ef38a1204e7b86091c0c0c455bdbf38b2

varon@DESKTOP-H6GU6N4 MINGW64 ~/Desktop
$ openssl sha1 'Resumo Aulas.docx'
SHA1(Resumo Aulas.docx)= 827ebd1ef38a1204e7b86091c0c0c455bdbf38b2

varon@DESKTOP-H6GU6N4 MINGW64 ~/Desktop
$ openssl sha1 'Resumo Aulas.docx'
SHA1(Resumo Aulas.docx)= bb1ecd59da0f245f40ce2e27d6cabf6956b70bd

varon@DESKTOP-H6GU6N4 MINGW64 ~/Desktop
$ openssl sha1 'Resumo Aulas.docx'
SHA1(Resumo Aulas.docx)= b909972fb7e0e6e033d1cf1ec6d20a02502ddc30

varon@DESKTOP-H6GU6N4 MINGW64 ~/Desktop
$ openssl sha1 'Resumo Aulas.docx'
SHA1(Resumo Aulas.docx)= d905457e487660fe154a0b7b69f0f45a188cf1a3

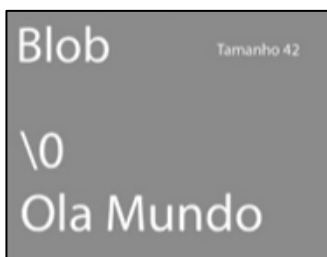
varon@DESKTOP-H6GU6N4 MINGW64 ~/Desktop
$
  
```

Objetos Fundamentais

Existem 3 tipos básicos de objetos do Git: BLOBS, TREES e COMMITS.

- **Blobs:** o jeito que o Git lida e manipula é através de objetos específicos. Os arquivos ficam guardados dentro de objetos chamados **blob**. Esse objeto possui metadados do Git nele.

O objeto “blob” vai ter o tipo do objeto, nesse caso blob, o tamanho dessa String ou desse arquivo, um “\0” e o conteúdo desse arquivo, que são os metadados do Git.



- **Trees:** as **trees** armazenam os blobs. Verificamos aqui uma crescente, onde o “blob” é o bloco básico de composição, e a “tree” é quem aponta e armazena os diferentes tipos de blobs.

As “trees” também contêm metadados, possuindo o “\0”. Elas apontam para o blob, que por sua vez tem o SHA1. Enquanto o blob só guarda o SHA1 do arquivo, a “tree” guarda o nome do arquivo e é a responsável por montar toda a estrutura de onde estão os arquivos.

Assim como no SO um diretório pode conter outro diretório, uma tree pode apontar para outras trees e elas também tem um SHA1 desse metadado.

Os objetos ficam armazenados entre si para sempre. Se mudarmos um caractere no arquivo, o SHA1 da blob irá mudar e SHA1 da tree também irá mudar.

Tree	<tamanho>
\0	
blob	sa4d8s texto.txt

- **Commit:** esse é o objeto mais importante de todos pois é ele que vai juntar tudo o que vai dar sentido para a alteração que está sendo feita. O **commit** pode apontar para uma **tree** ou para um “parente”, que é o último commit realizado antes dele. O commit também aponta para um autor e para uma mensagem.

Commit	<tamanho>
tree	s4a5sq1
parente	a98acq1
autor	perkles
mensagem	"inicia ..."
timestamp	

A mensagem é passada no objeto commit e é ela que vai explicar e dar significado para as alterações feitas nesses arquivos e nessas pastas. O autor que esse objeto leva é o nome de quem fez a alteração. Os commits também possuem um *timestamp*, que é um carimbo do tempo (data e hora) de quando ele foi criado. Os commits também possuem a encriptação do SHA1 que é o identificador dos seus metadados.

Se alterarmos um dado dentro de uma blob de um arquivo, vai ser gerado um novo SHA1 dessa blob. Como tem uma tree apontando para essa blob, os metadados da tree também serão alterados. Consequentemente, como o commit aponta para a tree, se o metadado dessa tree é alterado, o metadado do commit também será alterado, gerando uma nova encriptação do SHA1. O commit é único para cada autor que está realizando a alteração.

O Git se torna um **sistema distribuído seguro** porque os **commits** são impossíveis de serem alterados. Em um projeto de software onde diversas pessoas contribuem no contigo, tanto na versão que está no servidor quanto na versão que está com qualquer uma das pessoas do grupo, também são versões confiáveis.

Chaves SSH e Tokens

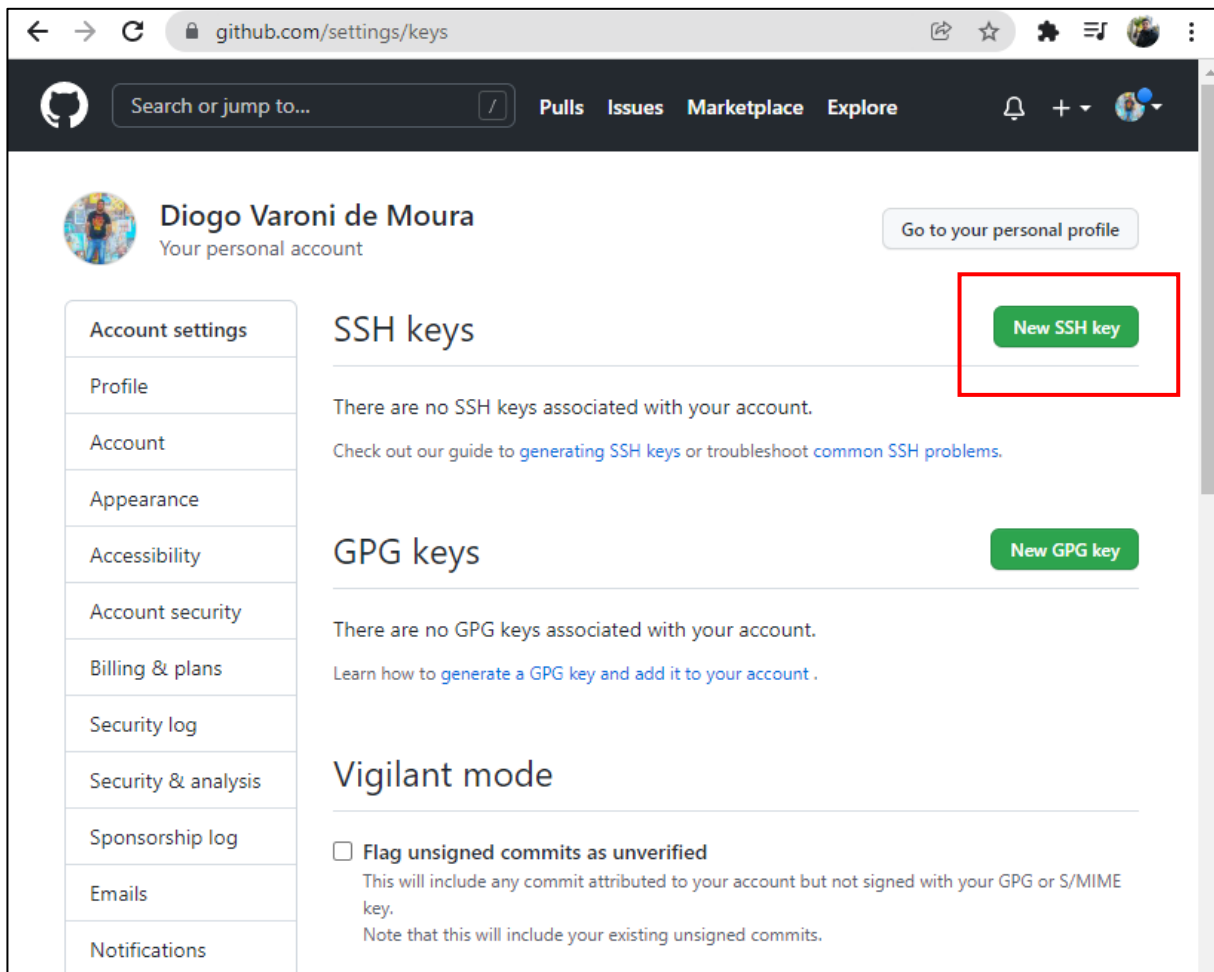
Esses processos de autenticação surgiram para fornecer mais segurança na hora de jogar o código para o GitHub. Antes esse processo era feito apenas com nome de usuário e senha mas o GitHub viu a necessidade de aumentar a segurança criando métodos de autenticação de seus usuários.

- **Chave SSH:** é uma forma de estabelecer uma conexão segura e encriptada de duas máquinas. Como iremos nos conectar com o servidor do GitHub, nós iremos configurar nossa máquina local como uma máquina confiável para o GitHub, estabelecendo essa conexão com duas chaves, sendo uma máquina pública e outra privada. Siga os passos a seguir para criar a chave SSH.

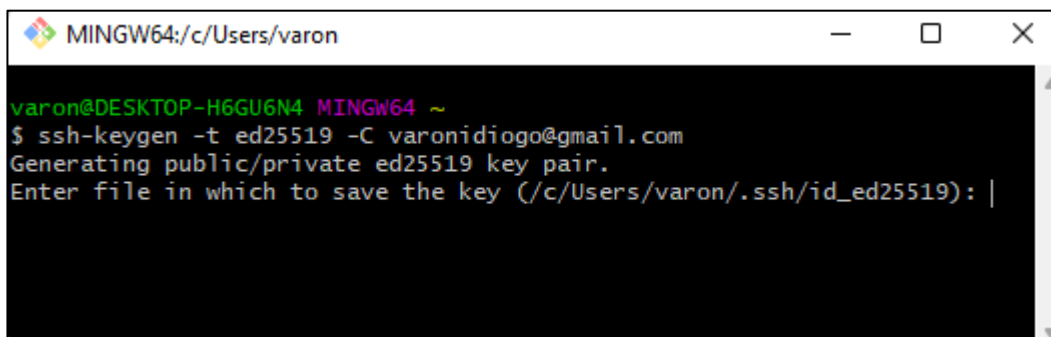
1º Passo: na plataforma do GitHub clique na imagem do perfil e em seguida no botão *Settings*.

The screenshot shows the GitHub profile page for Diogo Varoni de Moura. The profile picture is a circular image of a man standing in front of a wall covered in posters. The name 'Diogo Varoni de Moura' and the username 'diogovaroni' are displayed below the profile picture. A button labeled 'Edit profile' is visible. The page shows 'Popular repositories' and a message stating 'You don't have any public repositories yet.' Below this, it indicates '22 contributions in the last year' with a calendar grid showing contributions for June, July, August, and September. A dropdown menu is open on the right side of the page, showing the user is signed in as 'diogovaroni'. The menu includes options like 'Set status', 'Your profile', 'Your repositories', 'Your codespaces', 'Your projects', 'Your stars', 'Your gists', 'Upgrade', 'Feature preview', 'Help', 'Settings' (highlighted in blue), and 'Sign out'. A 'Less' button is also visible at the bottom of the menu.

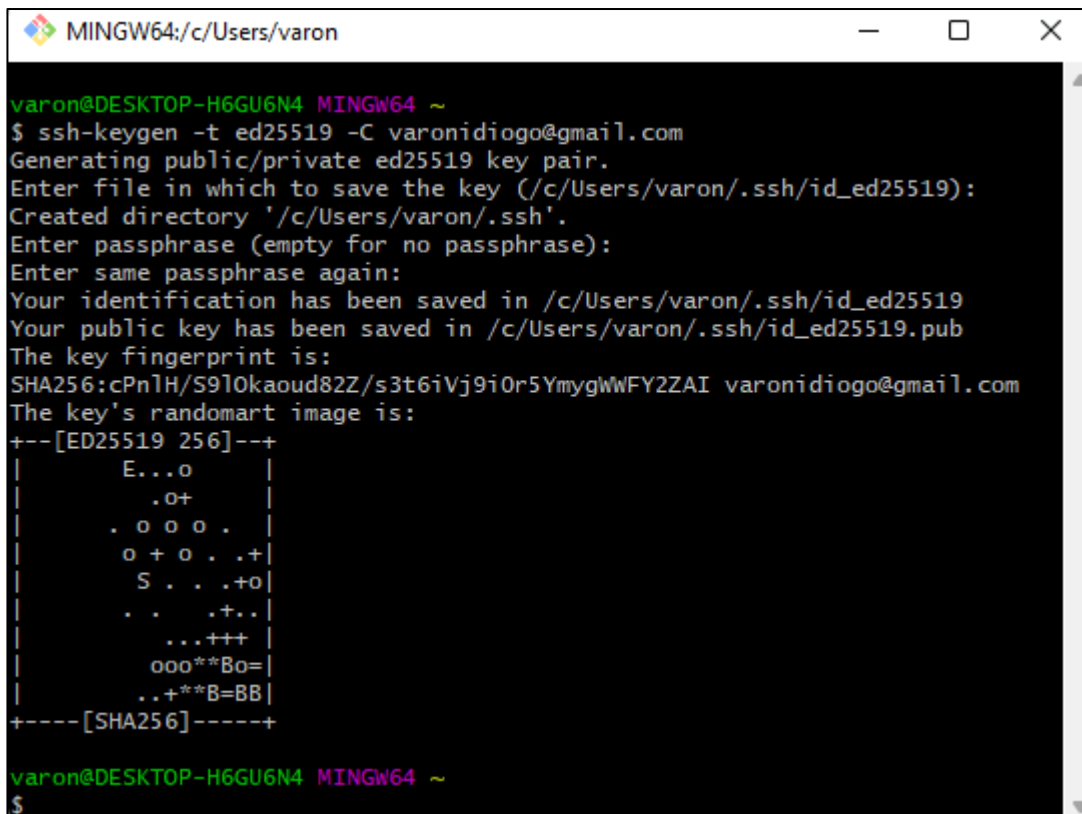
2º Passo: clique na opção “SSH and GPG Keys”. Em seguida clique no botão verde “New SSH key”.



3º Passo: agora vamos fazer o processo pelo terminal (CLI) do Git para gerar essa chave e deixar ela ligada na nossa máquina. Abra o Git Bash na área de trabalho e digite o comando **ssh-keygen -t ed25519 -c 'e-mail'**. Com esse comando serão criadas as chaves e irá mostrar o local onde essas chaves irão ‘morar’.



Perceba que na última linha do CLI é mostrado onde a chave gerada será salva, sendo nesse caso salva na pasta “usuarios”, dentro do diretório “varon”. Esse é o melhor caminho para salvar a chave gerada. Ao teclar “Enter” o CLI irá solicitar a senha do usuário para gerar a chave SSH e caso não haja senha, basta teclar Enter mais uma vez.



```

MINGW64:/c/Users/varon
varon@DESKTOP-H6GU6N4 MINGW64 ~
$ ssh-keygen -t ed25519 -C varonidiogo@gmail.com
Generating public/private ed25519 key pair.
Enter file in which to save the key (/c/Users/varon/.ssh/id_ed25519):
Created directory '/c/Users/varon/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/varon/.ssh/id_ed25519
Your public key has been saved in /c/Users/varon/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:cPn1H/S9l0kaoud82Z/s3t6iVj9i0r5YmyglWwFY2ZAI varonidiogo@gmail.com
The key's randomart image is:
+--[ED25519 256]--+
|      E...o      |
|      .o+       |
|    . o o o .   |
|    o + o . .+  |
|    S . . .+o   |
|    . . . .+..  |
|    ...+++     |
|    ooo**Bo=   |
|    ..+**B=BB  |
+-----[SHA256]-----+

varon@DESKTOP-H6GU6N4 MINGW64 ~
$

```

Finalizado esse processo de gerar a chave, vamos navegar entre os diretórios até onde a chave foi salva e visualizar essa chave. Para navegar direto para esse diretório, basta digitar o comando **“cd /c/Users/varon/.ssh/”**. Em seguida digite o comando **“ls”** para listar as chaves pública e privada.



```

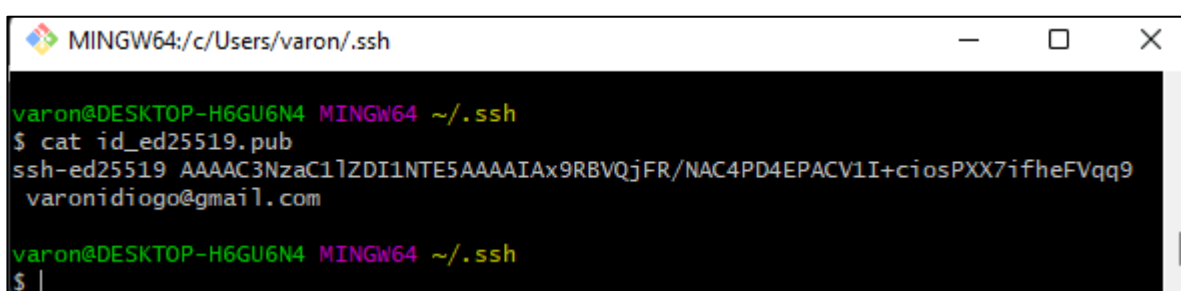
MINGW64:/c/Users/varon/.ssh
varon@DESKTOP-H6GU6N4 MINGW64 ~
$ cd .ssh/

varon@DESKTOP-H6GU6N4 MINGW64 ~/.ssh
$ ls
id_ed25519  id_ed25519.pub

varon@DESKTOP-H6GU6N4 MINGW64 ~/.ssh
$ |

```

Agora vamos usar um comando especial para visualizar o conteúdo dessas chaves. Esse é o conteúdo que iremos utilizar no Git Hub. Digite o comando **“cat id_ed25519.pub”**, lembrando que a chave que iremos utilizar e expor no Git Hub é a chave pública.



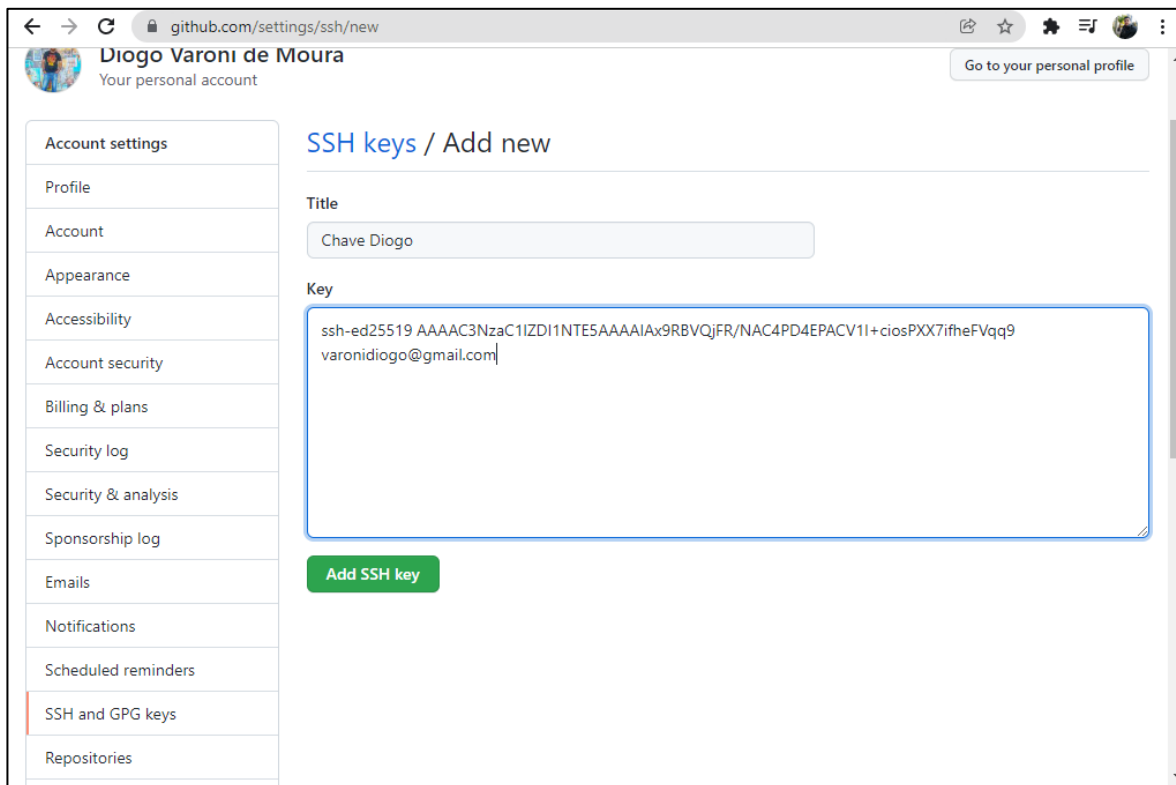
```

MINGW64:/c/Users/varon/.ssh
varon@DESKTOP-H6GU6N4 MINGW64 ~/.ssh
$ cat id_ed25519.pub
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIAx9RBVQjFR/NAC4PD4EPACV1I+ciosPXX7ifheFVqq9
varonidiogo@gmail.com

varon@DESKTOP-H6GU6N4 MINGW64 ~/.ssh
$ |

```

Vamos copiar esse conteúdo e ir lá no GitHub para colar ele. O GitHub irá solicitar que o usuário se autentique com senha.



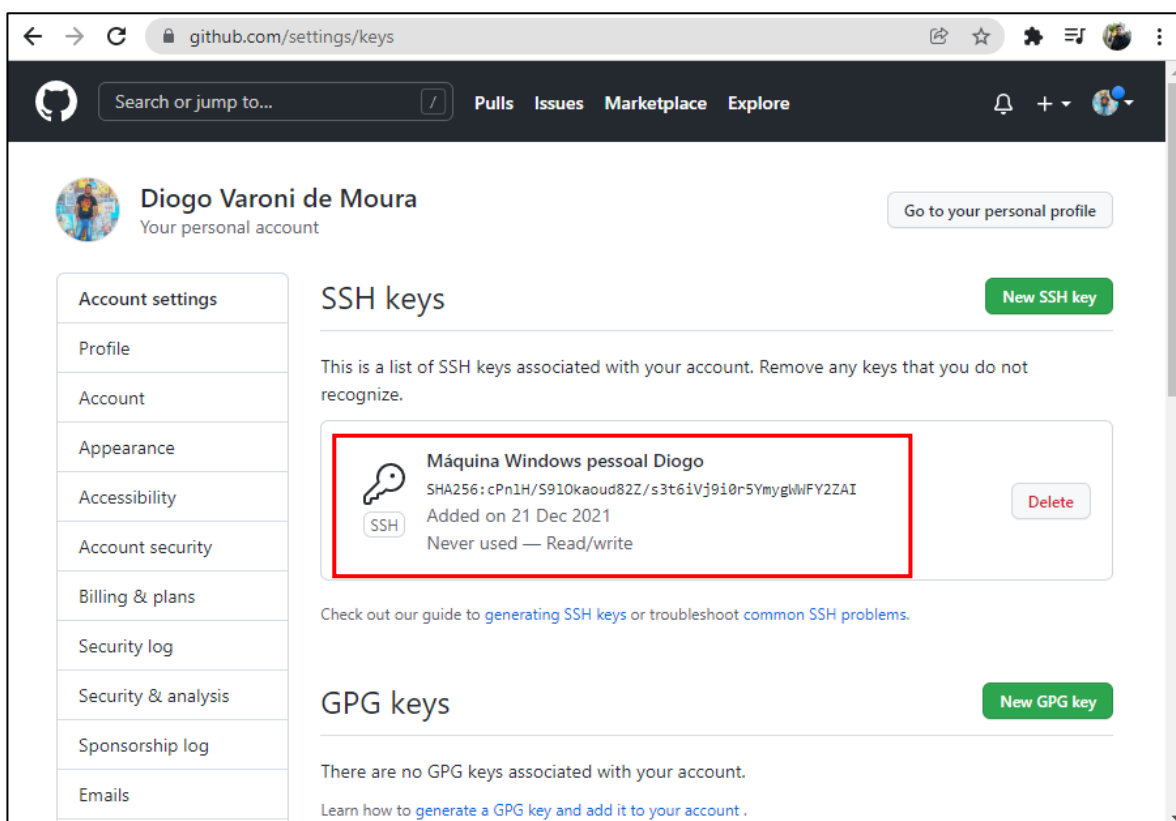
The screenshot shows the GitHub 'SSH keys / Add new' page. On the left is a sidebar with account settings. The main area has a form with a 'Title' field containing 'Chave Diogo' and a 'Key' text area containing a long SSH key. A green 'Add SSH key' button is at the bottom.

Account settings / SSH keys / Add new

Title: Chave Diogo

Key: ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIAx9RBVQJfR/NAC4PD4EPACV1I+ciosPXX7ifheFVqq9varonidiogo@gmail.com

Add SSH key



The screenshot shows the GitHub 'SSH keys' page. It lists the SSH key added in the previous step, highlighted with a red box. The key is named 'Máquina Windows pessoal Diogo' and was added on 21 Dec 2021. A 'Delete' button is next to it. Below the list is a link to a guide on generating SSH keys. The 'GPG keys' section below is empty.

SSH keys

This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.

SSH key icon	Key name	Key ID	Added on	Usage	Action
	Máquina Windows pessoal Diogo	SHA256:cPn1H/S910kaoud82Z/s3t6iVj9i0r5YmygwWfY2ZAI	Added on 21 Dec 2021	Never used — Read/write	Delete

Check out our guide to [generating SSH keys](#) or troubleshoot [common SSH problems](#).

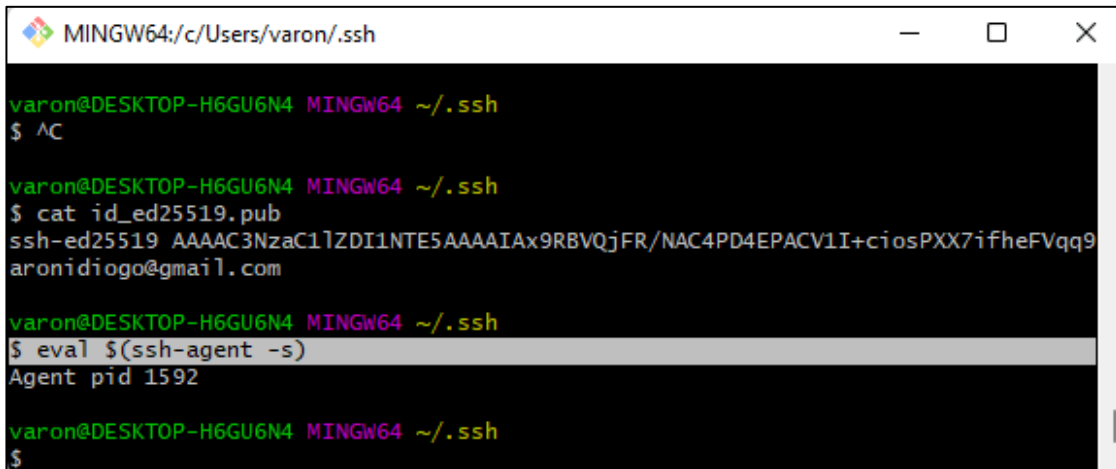
GPG keys

There are no GPG keys associated with your account.

Learn how to [generate a GPG key and add it to your account](#).

Pronto. A chave SSH foi gerada. Na página do GitHub é mostrado a lista de chaves criadas na conta, assim como a chave atual criada. Agora precisamos fazer mais um processo no **CLI** para que a chave funcione sem erros. Vamos inicializar o SSH Agent, que é uma

entidade que será encarregada de pegar as chaves e trabalhar com elas. Digite no CLI o comando “**eval \$(ssh-agent -s)**”.



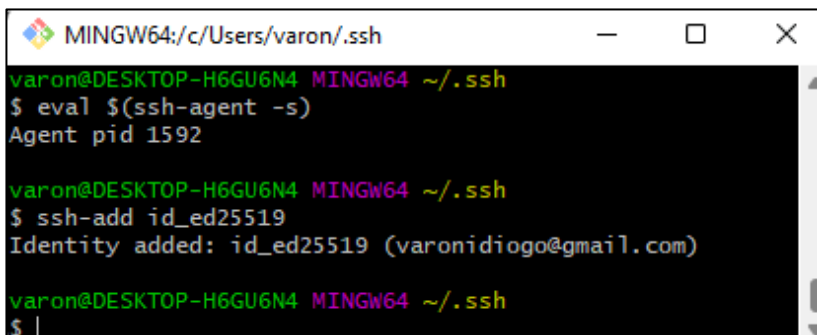
```
MINGW64:/c/Users/varon/.ssh
varon@DESKTOP-H6GU6N4 MINGW64 ~/.ssh
$ ^C

varon@DESKTOP-H6GU6N4 MINGW64 ~/.ssh
$ cat id_ed25519.pub
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIAx9RBVQjFR/NAC4PD4EPACV1I+ciosPXX7ifheFVqq9
aronidiogo@gmail.com

varon@DESKTOP-H6GU6N4 MINGW64 ~/.ssh
$ eval $(ssh-agent -s)
Agent pid 1592

varon@DESKTOP-H6GU6N4 MINGW64 ~/.ssh
$
```

Ao gerar o agente, vamos entregar a nossa chave **privada** para ele. Vamos digitar o comando “**ssh-add id_ed25519**”. Caso a chave não esteja no mesmo diretório em que o CLI está aberto, basta navegar até o diretório ou digitar o caminho direto no CLI. Toda vez que chegar a criptografia com essa chave, o agente irá descriptografar ela.



```
MINGW64:/c/Users/varon/.ssh
varon@DESKTOP-H6GU6N4 MINGW64 ~/.ssh
$ eval $(ssh-agent -s)
Agent pid 1592

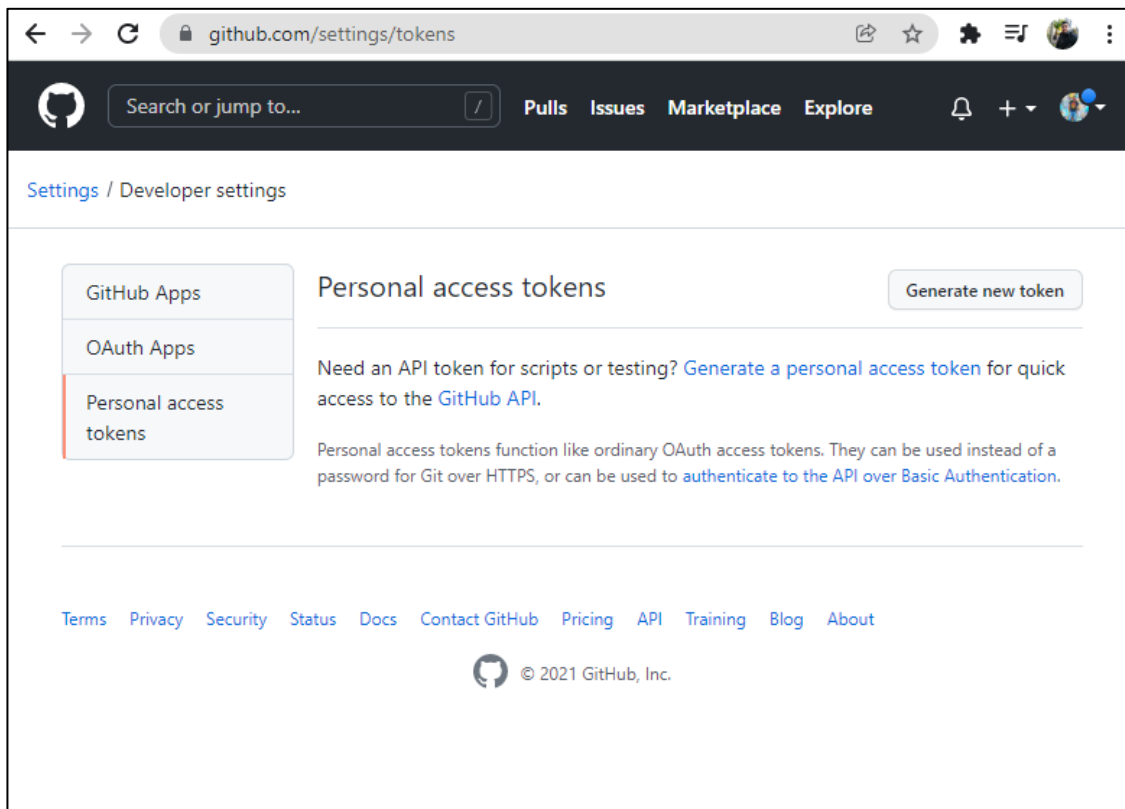
varon@DESKTOP-H6GU6N4 MINGW64 ~/.ssh
$ ssh-add id_ed25519
Identity added: id_ed25519 (varonidiogo@gmail.com)

varon@DESKTOP-H6GU6N4 MINGW64 ~/.ssh
$ |
```

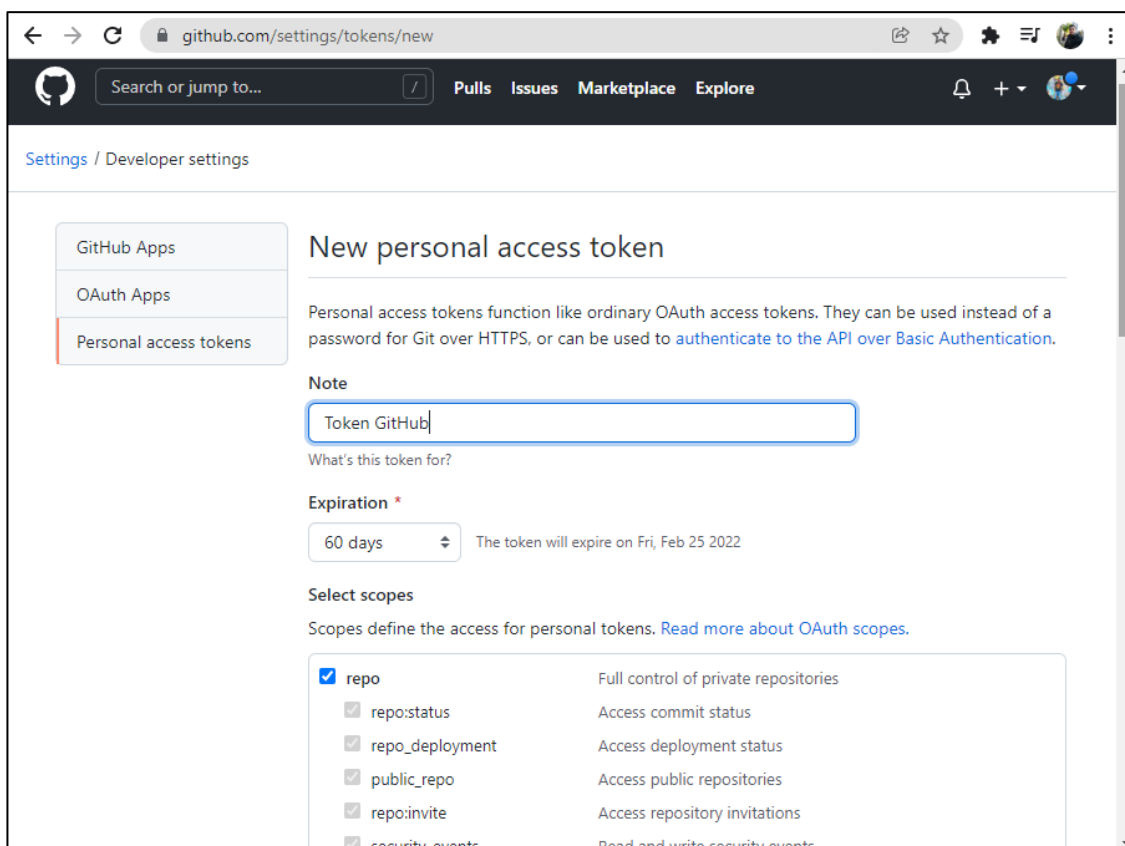
Pronto! O processo foi finalizado. Perceba que o CLI retornou que a identidade foi adicionada e mostrou o e-mail da conta no GitHub, que neste caso é varonidiogo@gmail.com.

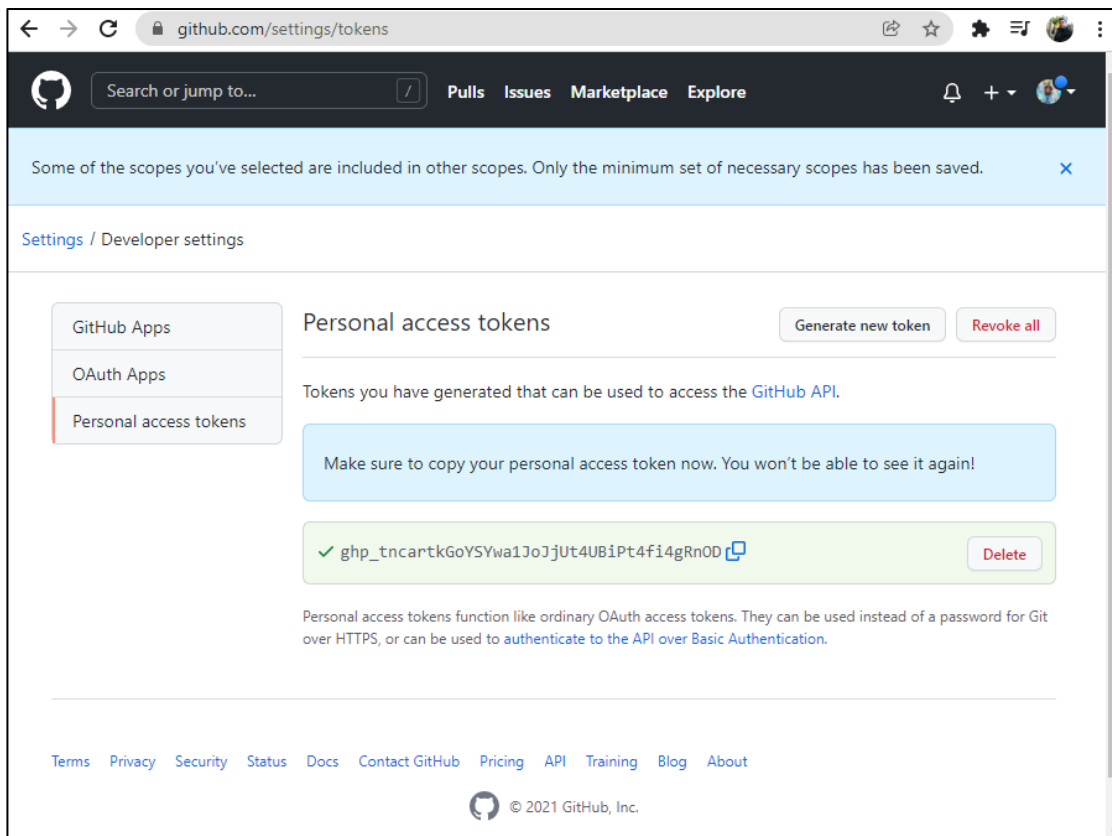
- **Token:** o token de acesso pessoal é a segunda forma de identificação segura que o GitHub oferece para os usuários. É um processo que se assemelha mais ao procedimento usando antes, que era digitar o nome de usuário e a senha. Iremos gerar o *Token* no GitHub, guardar ele na máquina e sempre que fizermos um commit, o Git irá solicitar o nome de usuário e a senha e em seguida o *Token* de acesso.

1º Passo: no GitHub selecione a opção *Settings* e em seguida a opção *Developer settings*. Agora selecione a opção **Personal access tokens** e clique no botão **Generate new token**.



2º Passo: preencha o campo **Note** e selecione a data que o token criado irá expirar. Em seguida marque a opção **Repo**. Agora clique no botão **Generate token**.





3º Passo: após gerar o Token copie e salve em algum arquivo seguro pois ele não fica salvo no GitHub. Se sairmos da página sem salvar o Token, teremos que gerar novamente. A seguir é dado o token criado: **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**.

Para clonar o repositório privado no GitHub usando o Token de acesso, devemos usar o protocolo HTTPS.

4. Primeiros Comandos com Git

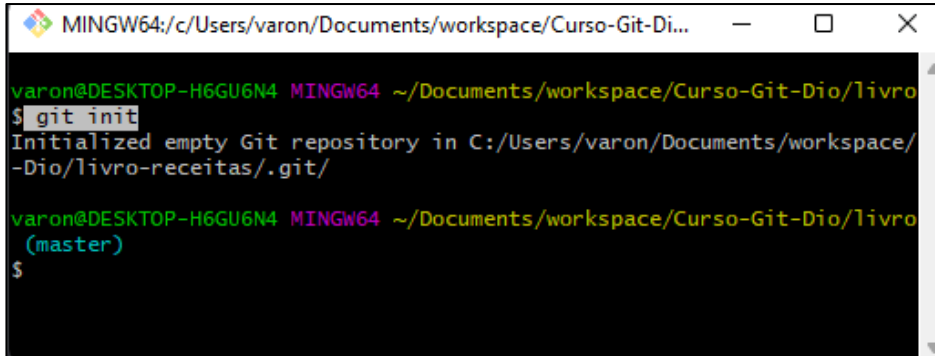
Nessa aula iremos os primeiros comandos com o Git. Com esses comandos será possível:

- Iniciar o Git;
- Iniciar o versionamento;
- Criar um commit.

- * Para iniciar o repositório do Git vamos utilizar o comando **git init**.
- * Para mover arquivos e começar o versionamento utilizamos o comando **git add**.
- * Para criar o primeiro commit utilizamos o comando **git commit**.

Criando um Repositório

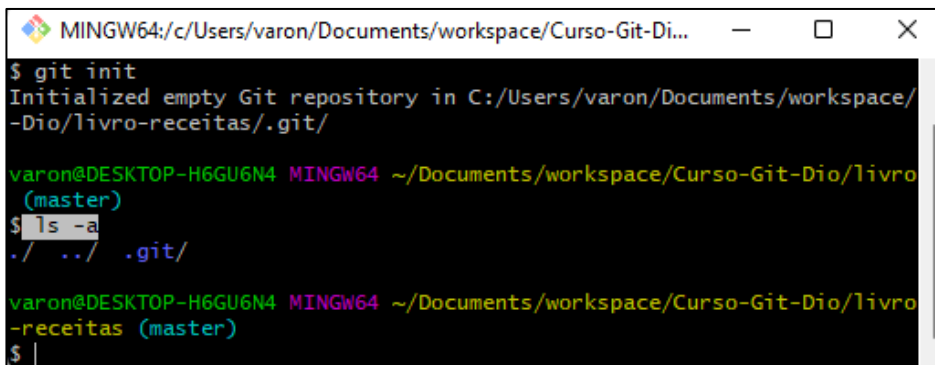
Uma forma didática de aprender a trabalhar com o Git é criar um diretório local chamada **Workspace** e dentro desse diretório guardar os arquivos que serão criados e versionados com o Git. Dentro desse diretório vamos criar uma pasta chamada de “**livro-receitas**”. Agora abra o Git Bash dentro dessa pasta e vamos rodar o comando **git init** que vai gerenciar e versionar o nosso código.



```
MINGW64:/c/Users/varon/Documents/workspace/Curso-Git-Di...
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
$ git init
Initialized empty Git repository in C:/Users/varon/Documents/workspace/Curso-Git-Dio/livro-receitas/.git/
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas (master)
$
```

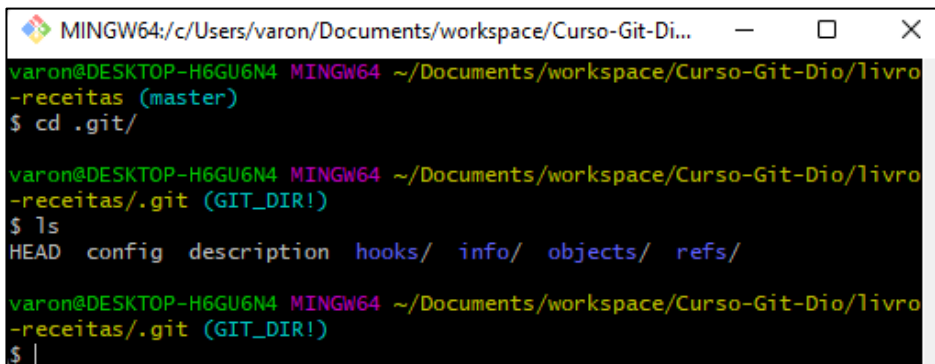
- Perceba que o próprio Git avisa que foi inicializado um repositório vazio dentro do diretório livro-receitas.

Obs.: usando o comando “**ls -a**” iremos ver os arquivos ocultos criados ao inicializar o Git, sendo que o arquivo principal é o “**.git/**”, que é a pasta gerencial do Git.



```
MINGW64:/c/Users/varon/Documents/workspace/Curso-Git-Di...
$ git init
Initialized empty Git repository in C:/Users/varon/Documents/workspace/Curso-Git-Dio/livro-receitas/.git/
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas (master)
$ ls -a
./ ../ .git/
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas (master)
$
```

Se entrarmos na pasta **.git/** no terminal e listarmos os conteúdos contidos dentro dela, veremos a estrutura referente ao próprio Git, incluindo o diretório chamado “**objects**”, que é o assunto da aula anterior. A medida que vamos criando nosso repositório, os arquivos dentro dessa pasta vão aumentando.



```
MINGW64:/c/Users/varon/Documents/workspace/Curso-Git-Di...
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas (master)
$ cd .git/
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas/.git (GIT_DIR!)
$ ls
HEAD config description hooks/ info/ objects/ refs/
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas/.git (GIT_DIR!)
$
```

Vamos voltar um nível no CLI, utilizando o comando “**cd..**”, retornando para a pasta “**livro-receitas**”.

Agora precisamos rodar algumas configurações iniciais. Primeiro, vamos rodar o comando “**git config -global user.email “varonidiogo@gmail.com”**” para informar o e-mail e em seguida vamos rodar o comando “**git config -global user.name**” para informar o nome de usuário. Essa configuração é necessária porque quando o objeto commit é criado, ele deve conter um autor atrelado a ele.

```
MINGW64:/c:/Users/varon/Documents/workspace/Curso-Git-Di...
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas (master)
$ git config --global user.email "varonidiogo@gmail.com"

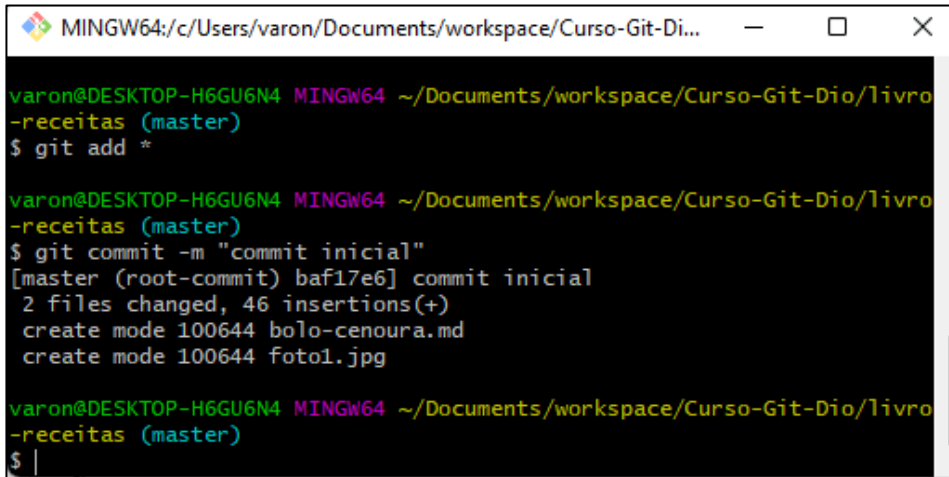
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas (master)
$ git config --global user.name Varoni

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas (master)
$ |
```

Após configurar o repositório iremos criar o primeiro arquivo dentro desse repositório. O arquivo que será do tipo *markdown*, utilizando o editor de texto **Typora**. Vamos criar uma receita de um bolo de cenoura, informando os ingredientes e o passo a passo.



Depois de salvar o arquivo, volte no CLI e faça o commit do arquivo, rodando o comando “**git add ***” e em seguida rode o comando “**git commit -m “commit inicial”**”. A mensagem que passamos entre aspas ao rodar o comando é a identificação do commit que foi realizado.



```

MINGW64:/c:/Users/varon/Documents/workspace/Curso-Git-Di...
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro
-receitas (master)
$ git add *

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro
-receitas (master)
$ git commit -m "commit inicial"
[master (root-commit) baf17e6] commit inicial
 2 files changed, 46 insertions(+)
 create mode 100644 bolo-cenoura.md
 create mode 100644 foto1.jpg

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro
-receitas (master)
$ |
  
```

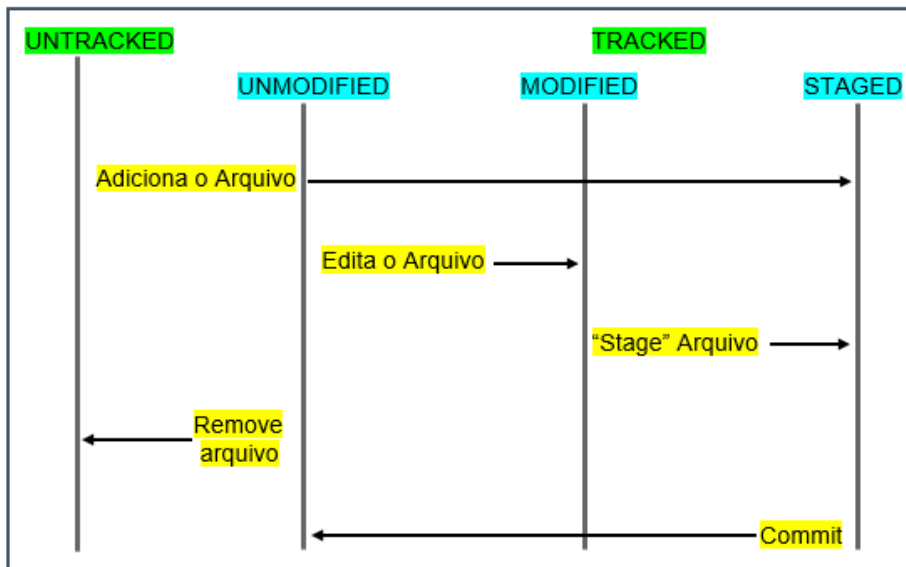
Pronto! Realizamos o primeiro commit do nosso repositório. Após gerar esse commit, o Git informa os arquivos alterados e o SHA1 do commit.

5. Ciclo de Vida dos Arquivos no Git

Git Init

Desde o início do curso falamos sobre diretórios e repositórios, porém essa não é a terminologia correta. Na última aula aprendemos a usar o **Git Init** e esse comando, além de criar a pasta **/git.**, inicializa o repositório do Git.

Quando usamos o “**git init**”, criamos um repositório no Git dentro de um diretório. Agora vamos aprender os conceitos de “**Tracked/Untracked**”. Observe a imagem a seguir:



- Dentro de “UNTRACKED” temos os arquivos que de fato o Git ainda não tem ciência da existência deles.
- Dentro de “TRACKED”, ou seja, dentro dos arquivos que são rastreáveis pelo Git, temos 3 estágios diferentes: “Unmodified”, “Modified” e “Staged”.

- **Unmodified:** é o arquivo que ainda não foi modificado.
- **Modified:** é o arquivo “unmodified” que sofreu alteração.
- **Staged:** é onde ficam os arquivos que estão se preparando para fazer parte de outro tipo de agrupamento que iremos ver mais para a frente.

Na aula anterior nós já manipulamos esses estados do Git de forma prática. O “**git init**” inicializa o repositório Git e quando usamos o comando “**git add**” estamos mudando o arquivo de “UNTRACKED” para “TRACKED”, deixando o Git ciente da existência dele, movendo o arquivo direto para “Staged”, que é a área que o arquivo está esperando pelo próximo passo.

Os arquivos “Unmodified” são os arquivos que estão dentro do repositório Git, mas que ainda não sofreram modificação. Quando abrimos o arquivo e editamos, ele muda automaticamente de “Unmodified” para “Modified”, pois o Git compara o SHA1 do arquivo e assim descobre que o arquivo tem modificação.

Se rodarmos o comando “**git add**” nesse arquivo “Modified”, ele será movido para o “Staged”. Se removermos o arquivo “Unmodified”, ele irá direto para “UNTRACKED”, onde o Git não tem ciência dele.

Quando movemos esse arquivo para o “Staged”, nós estamos preparando-o para fazer parte de um **Commit**, onde envelopamos todas essas modificações e escrevemos uma mensagem para o nosso “commit” com o autor e a data da alteração, que fazem parte do objeto commit.

O Commit retorna todos esses arquivos para “Unmodified”, começando o ciclo de novo, pois eles ficam aguardando novas modificações. Se abrimos novamente esse arquivo e o editarmos, ele irá voltar para “Modified” e se rodar o comando “**git add**” novamente, o arquivo volta para “Staged”.

Esse é o ciclo de vida dos arquivos no Git: “Unmodified”, “Modified” e “Staged”. Ele volta para “Unmodified” toda vez que fazemos o Commit desse arquivo e se abrimos o arquivo e fizermos uma alteração, ele volta para “Modified” e quando adicionamos um **Commit**, ele vai para “Staged”, e quando escrevemos as informações do **Commit**, ele volta para “Unmodified”.

Anteriormente foi dito que quando rodamos o comando “**git init**” nós estamos iniciando um repositório dentro do Git. Agora vamos entender o que os repositórios significam.

Quando estamos trabalhando com o Git, temos a separação em 2 ambientes: o **ambiente de desenvolvimento** e o **servidor**.

1. Ambiente de Desenvolvimento: representa todos os arquivos que estão na máquina local. Possui 3 áreas: repositório de trabalho, Staging área e repositório local.

2. Servidor: é onde está o repositório remoto, como por exemplo o **GitHub** e o **GitLab**.

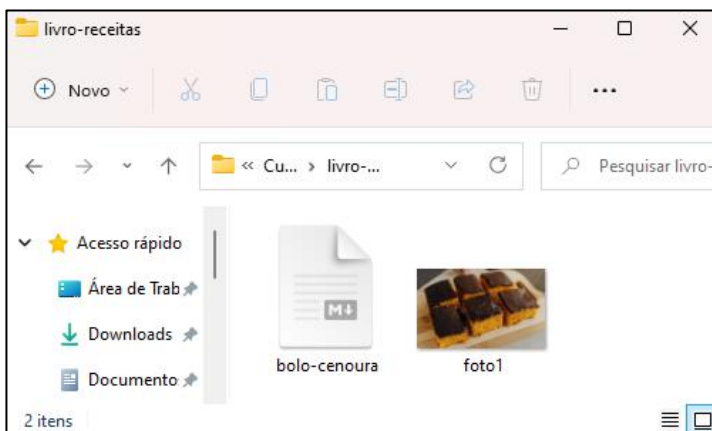
O Git é um sistema distribuído, no qual nós possuímos a versão dele no servidor (GitHub, GitLab) e a versão que está na máquina local (computador pessoal).

As alterações que são feitas no código e nos arquivos, que estão na máquina local, não repercutem diretamente no repositório remoto. Para isso, é necessário executar um conjunto de comandos específicos, que iremos ver mais para a frente.

Resumindo

- No ambiente de desenvolvimento temos o **repositório de trabalho** dentro da máquina local.
- A **Staging área** também faz parte do ambiente de desenvolvimento. Como vimos anteriormente, os arquivos modificados estão prontos para receberem um **Commit**.
- Após realizar um Commit, os arquivos modificados passam a fazer parte **do repositório local**.
- O repositório local, por sua vez, pode ser empurrado para o **repositório remoto**, que faz parte do servidor. O repositório local é composto apenas por **Commits**.

Agora vamos abrir o repositório que estávamos desenvolvendo o projeto do livro de receitas. Vamos realizar algumas alterações e usar alguns comandos novos. Digite o comando **“git status”** para verificar a situação inicial do nosso repositório.

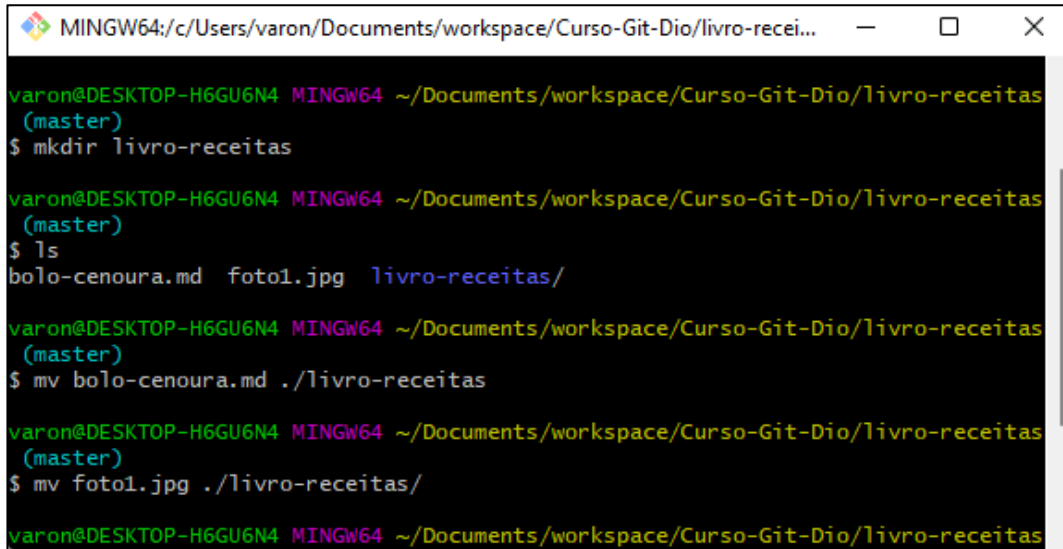


```
MINGW64:/c:/Users/varon/Documents/workspace/Curso-Git-Dio/livro-recei...
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ git status
On branch master
nothing to commit, working tree clean

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$
```

A mensagem *“working tree clean”* significa que o diretório de trabalho está limpo, sem nenhuma alteração (árvore de trabalho limpa).

- Crie um diretório chamado livro-receitas dentro do repositório e mova os arquivos para este diretório. Para criar esse repositório direto pelo CLI basta rodar o comando “**mkdir livro-receitas**”. Em seguida vamos mover os arquivos já criados para dentro desse diretório, utilizando o comando “**mv**”, que vem da palavra *move* (mover em inglês).



```
MINGW64:/c/Users/varon/Documents/workspace/Curso-Git-Dio/livro-recei...
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ mkdir livro-receitas

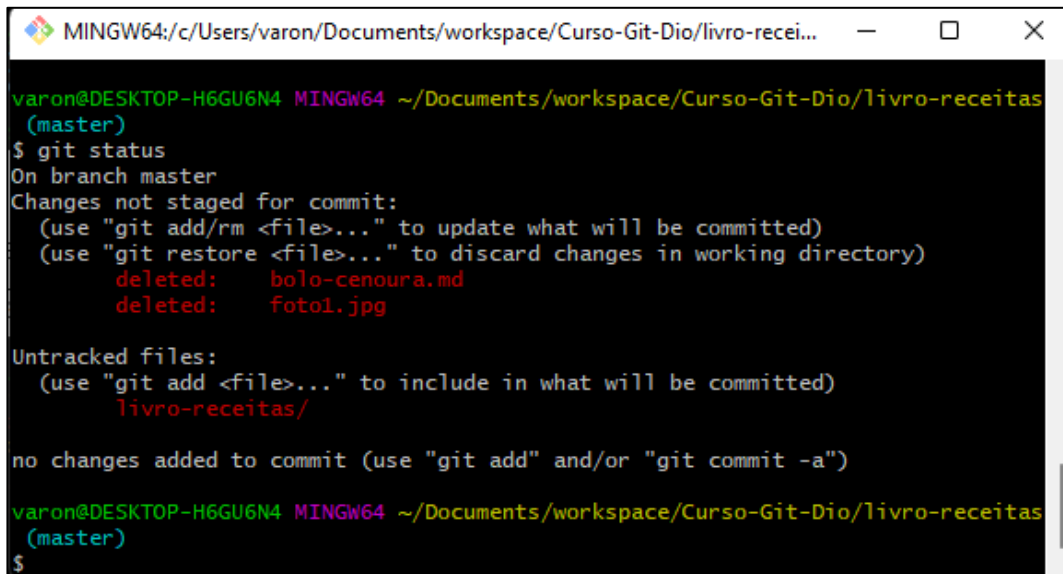
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ ls
bolo-cenoura.md  foto1.jpg  livro-receitas/

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ mv bolo-cenoura.md ./livro-receitas

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ mv foto1.jpg ./livro-receitas/

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
```

- Após essas alterações digite novamente o comando “**git status**”.



```
MINGW64:/c/Users/varon/Documents/workspace/Curso-Git-Dio/livro-recei...
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    bolo-cenoura.md
        deleted:    foto1.jpg

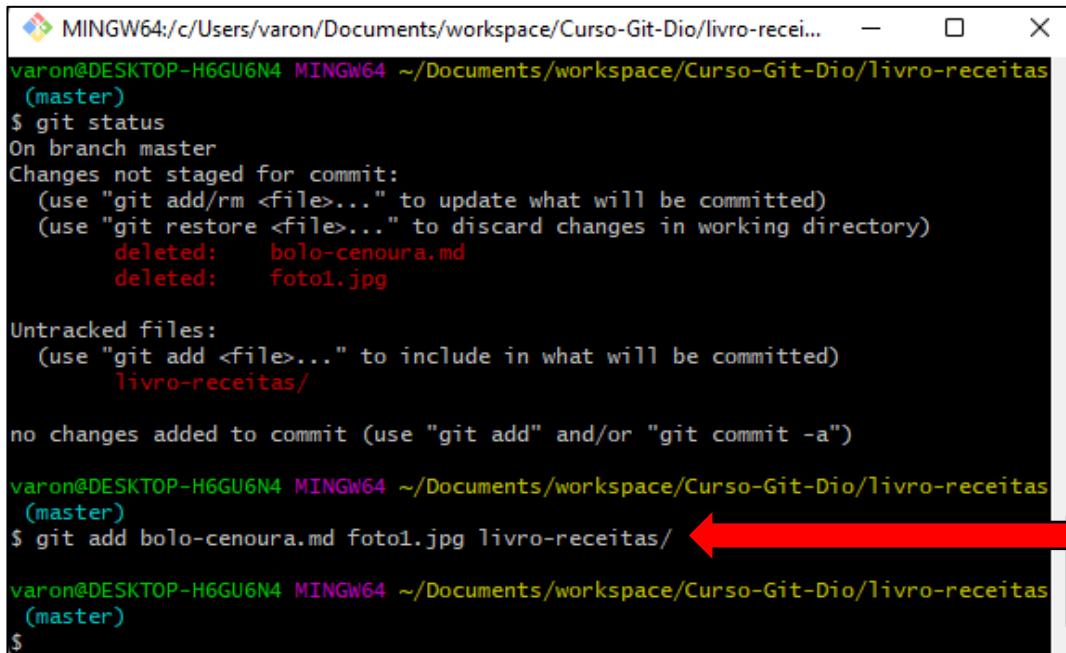
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        livro-receitas/

no changes added to commit (use "git add" and/or "git commit -a")

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$
```

Perceba na imagem a mensagem “*Changes not Staged for commit*”, indicando que foram feitas alterações e essas alterações não foram preparadas para receber o Commit. Na mensagem o Git mostra as sugestões de comandos para serem feitos nesse caso, que é adicionar os arquivos que sofreram mudanças e alterações ao “Staged” para serem preparadas para o commit. Após rodar o comando “**git commit**” vamos criar o *Snap Shot* do nosso repositório, que é como se fosse uma imagem congelada de todas as alterações e modificações que foram feitas.

- Agora vamos adicionar essas modificações para a área de Staged com o comando “**git add**”, seguido do novo dos arquivos alterados.



```

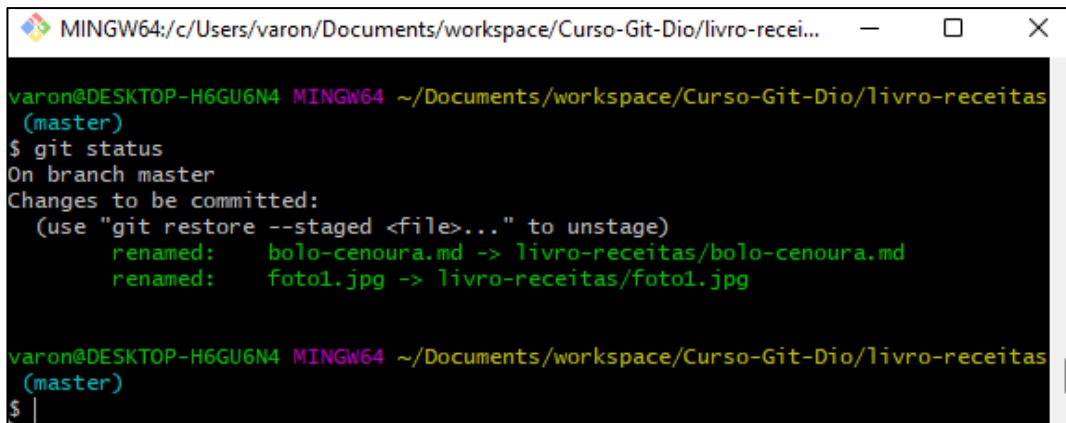
MINGW64:/c/Users/varon/Documents/workspace/Curso-Git-Dio/livro-recei...
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    bolo-cenoura.md
        deleted:    foto1.jpg

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        livro-receitas/

no changes added to commit (use "git add" and/or "git commit -a")

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ git add bolo-cenoura.md foto1.jpg livro-receitas/
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$
  
```

- Digite o comando “**git status**” para verificar se os arquivos modificados estão na área de Staged e prontos para receberem o Commit.



```

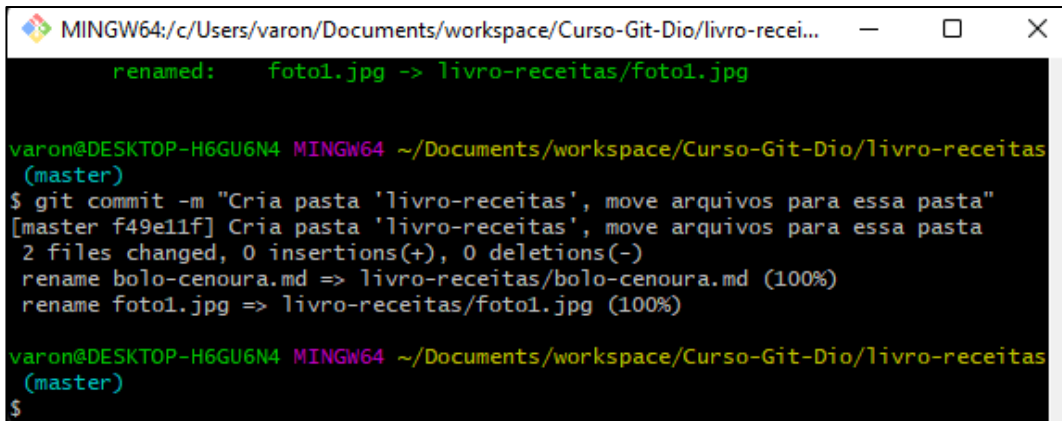
MINGW64:/c/Users/varon/Documents/workspace/Curso-Git-Dio/livro-recei...
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    bolo-cenoura.md -> livro-receitas/bolo-cenoura.md
        renamed:    foto1.jpg -> livro-receitas/foto1.jpg

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$
  
```

A mensagem “*Changes to be committed*” significa que as modificações que foram feitas já estão prontas para receber o commit. Se quisermos restaurar esses arquivos e tirar da área de Staged, devemos rodar o comando “**git restore --staged**” seguido do nome do arquivo.

- Agora vamos criar o primeiro **Commit** do nosso repositório. Vamos digitar o comando “**git commit -m**” seguido de uma mensagem que identifique esse Commit. Veja o exemplo:

<git commit -m “cria pasta livro-receitas, move arquivos para pasta”>



```

MINGW64:/c/Users/varon/Documents/workspace/Curso-Git-Dio/livro-recei...
renamed:    foto1.jpg -> livro-receitas/foto1.jpg

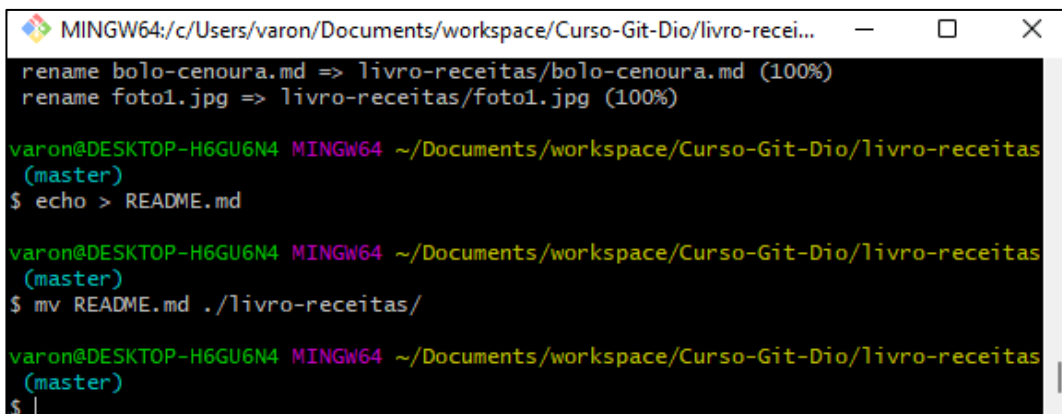
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ git commit -m "Cria pasta 'livro-receitas', move arquivos para essa pasta"
[master f49e11f] Cria pasta 'livro-receitas', move arquivos para essa pasta
2 files changed, 0 insertions(+), 0 deletions(-)
rename bolo-cenoura.md => livro-receitas/bolo-cenoura.md (100%)
rename foto1.jpg => livro-receitas/foto1.jpg (100%)

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$

```

Veja na imagem que o Git mostra a mensagem do Commit e mostra o seu SHA1 também, apresentando também o nome dos arquivos modificados. Digitado novamente o comando “**git status**” o Git vai mostrar que não tem mais nenhum arquivo “untracked” ou “modified” que precisa ser colocado em “Staged” para receber o Commit. Irá aparecer a mensagem que a área de trabalho (*work tree*) está limpa.

- Agora vamos criar e adicionar mais um arquivo, que vai servir como um Index da nossa lista de receitas. Esse será o arquivo que as pessoas vão ver inicialmente e que vai conter todas as receitas do nosso livro. No CLI digite o comando < **echo > README.md** >



```

MINGW64:/c/Users/varon/Documents/workspace/Curso-Git-Dio/livro-recei...
rename bolo-cenoura.md => livro-receitas/bolo-cenoura.md (100%)
rename foto1.jpg => livro-receitas/foto1.jpg (100%)

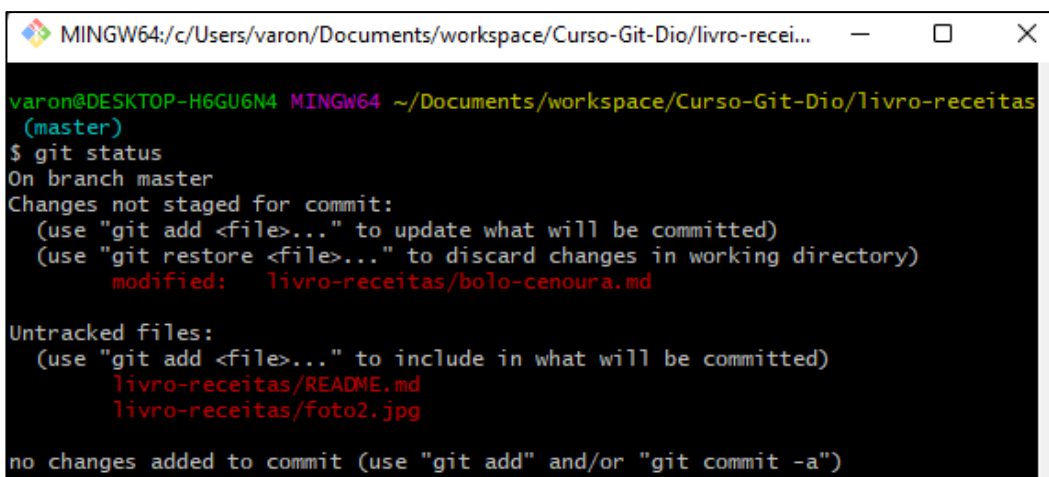
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ echo > README.md

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ mv README.md ./livro-receitas/

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$

```

- Vamos abrir o arquivo e colocar a lista de receitas que o nosso livro irá ter. Também podemos adicionar imagens nesse arquivo. Em seguida execute o comando “**git status**”.



```

MINGW64:/c/Users/varon/Documents/workspace/Curso-Git-Dio/livro-recei...

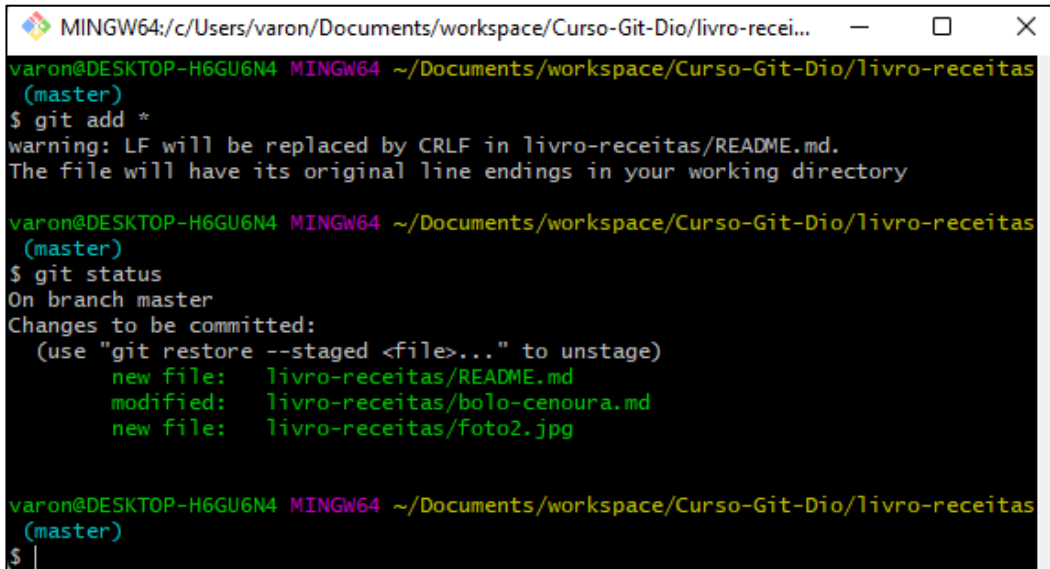
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   livro-receitas/bolo-cenoura.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    livro-receitas/README.md
    livro-receitas/foto2.jpg

no changes added to commit (use "git add" and/or "git commit -a")

```

Na imagem anterior o Git mostra que um arquivo foi modificado (*modified*) e que precisa ser colocada em “Staged” para receber o Commit. O Git também mostra que existem dois arquivos “Untracked”, que são aqueles arquivos que ainda não estão no repositório do Git. Esses arquivos também precisam ser colocados em “Staged” para ambos receberem o Commit. Vamos agora usar o comando “**git add ***”, que irá pegar todas as alterações feitas no diretório de trabalho e adicioná-las no “Staged”.

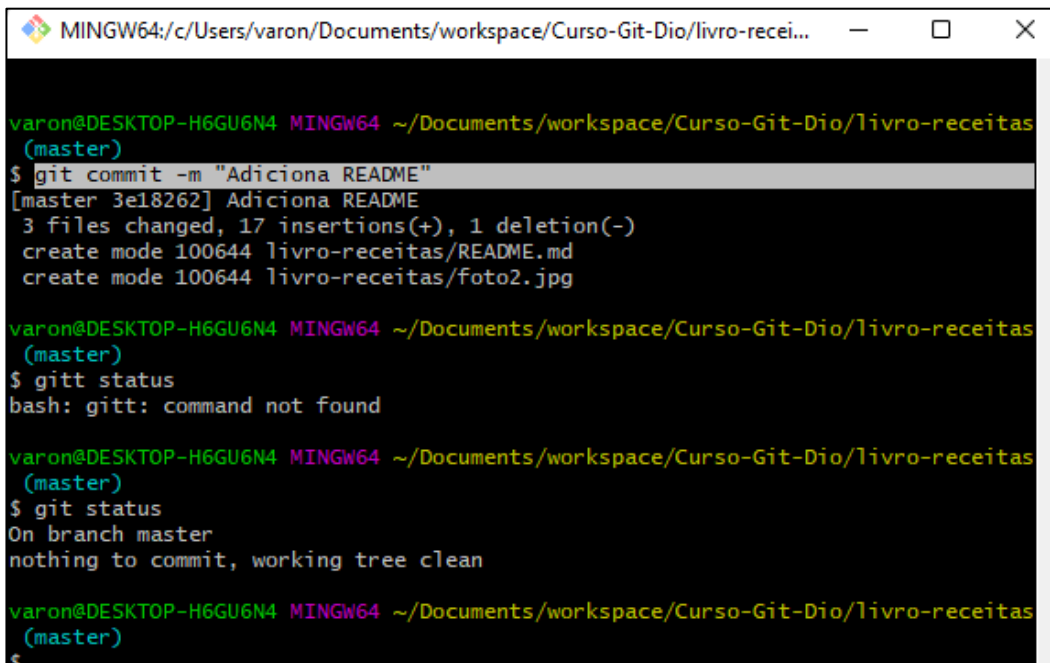


```
MINGW64; c:/Users/varon/Documents/workspace/Curso-Git-Dio/livro-recei...
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ git add *
warning: LF will be replaced by CRLF in livro-receitas/README.md.
The file will have its original line endings in your working directory

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   livro-receitas/README.md
        modified:   livro-receitas/bolo-cenoura.md
        new file:   livro-receitas/foto2.jpg

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$
```

- O Git mostra os arquivos que estão em Staged e prontos para receberem o Commit. Vamos rodar o comando “**git commit -m**” seguido da mensagem “Adiciona README” para identificar o Commit realizado.



```
MINGW64; c:/Users/varon/Documents/workspace/Curso-Git-Dio/livro-recei...
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ git commit -m "Adiciona README"
[master 3e18262] Adiciona README
3 files changed, 17 insertions(+), 1 deletion(-)
create mode 100644 livro-receitas/README.md
create mode 100644 livro-receitas/foto2.jpg

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ gitt status
bash: gitt: command not found

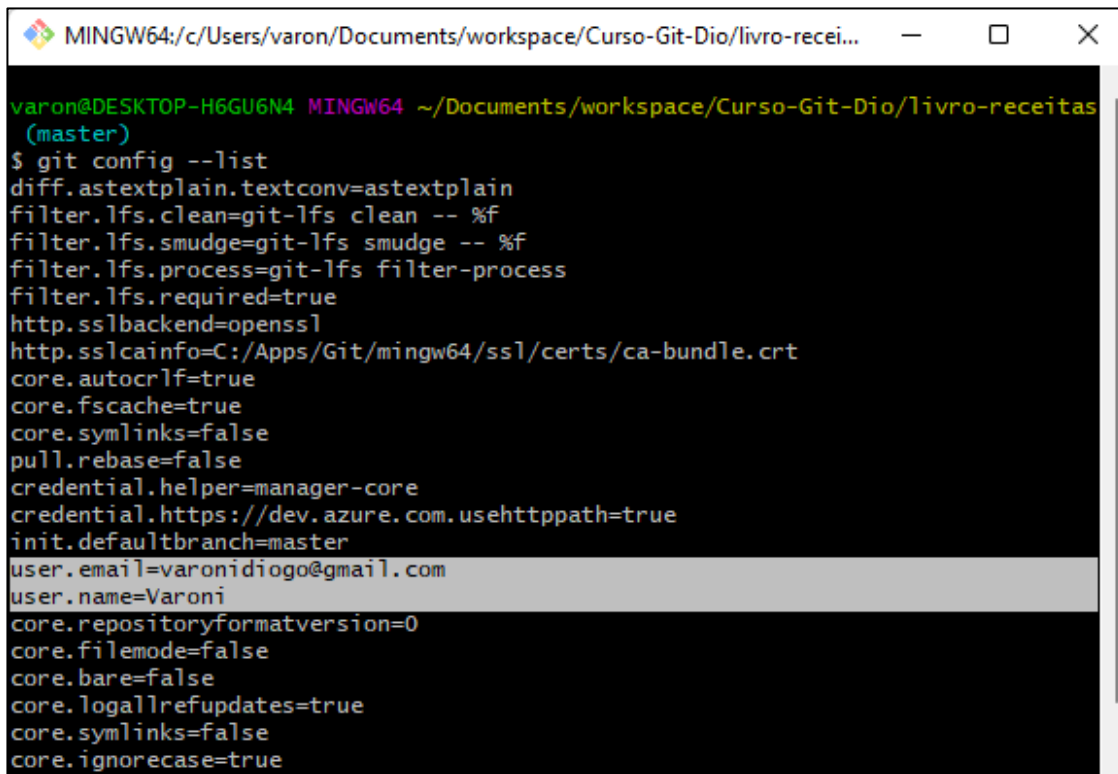
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ git status
On branch master
nothing to commit, working tree clean

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$
```

6. Introdução ao GitHub

Trabalhando com o GitHub

Nessa aula iremos aprender a criar um repositório remoto no GitHub e como apontar o nosso repositório local para o repositório remoto no GitHub. Primeiramente precisamos verificar se o e-mail associado ao Git é o mesmo e-mail usado no GitHub. O e-mail precisa ser exatamente o mesmo. Para verificar vamos digitar o comando **“git config --list”**.

A screenshot of a Windows terminal window titled 'MINGW64: /c/Users/varon/Documents/workspace/Curso-Git-Dio/livro-recei...'. The terminal shows the command '\$ git config --list' and its output. The output lists various Git configuration settings, including diff, filter, http, core, pull, credential, init, user, and core settings. The user's email 'varonidiogo@gmail.com' and name 'Varoni' are highlighted in the output.

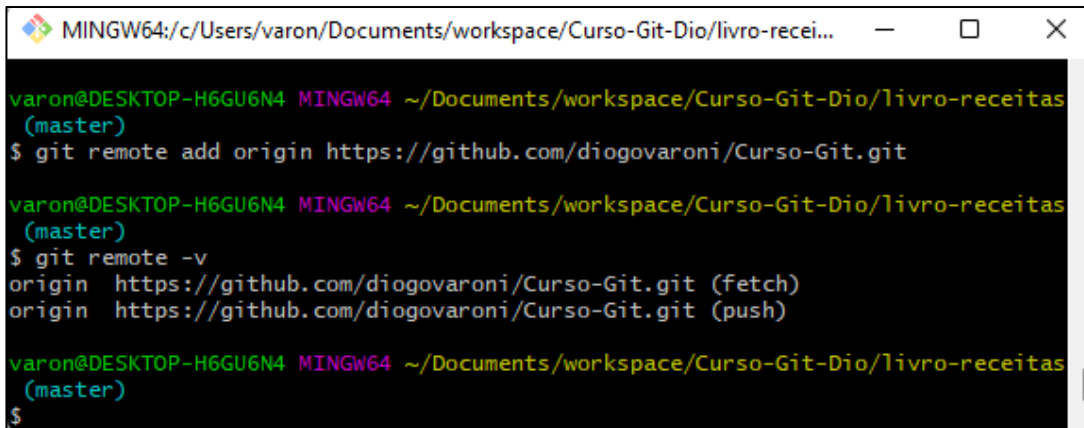
```
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Apps/Git/mingw64/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
pull.rebase=false
credential.helper=manager-core
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=master
user.email=varonidiogo@gmail.com
user.name=Varoni
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
core.symlinks=false
core.ignorecase=true
```

Nesse caso o e-mail é o mesmo associado ao GitHub. Caso precise alterar alguma configuração, como o e-mail, digite o comando **“git config --global --unset user.email”**. Se depois de rodar esse comando tentarmos fazer um Commit, o Git não irá achar o autor, pois essa configuração foi alterada. O Git vai pedir que o autor se identifique. Agora digite o comando **“git config --global user.email “seu e-mail”** para adicionar novamente o e-mail.

O ideal é que quando estivermos instalando uma configuração nova do Git, o user.name e o e-mail já estejam associados com o repositório local, que nesse caso é o GitHub.

Agora vamos entrar na conta do GitHub e criar um repositório. O nome desse repositório será **“Curso Git”**. Vamos deixar ele em modo privado e a *checkbox* “Add a Readme file” será desmarcada, pois no nosso repositório local já criamos o arquivo README.

Depois de criar o repositório remoto vamos apontar o repositório da máquina local para este repositório. Vamos copiar o endereço (URL) desse repositório e no CLI do Git digite o comando **“git remote add original https://github.com/diogovaroni/Curso-Git.git**. Após rodar esse comando, digite o comando **“git remote -v”** para verificar se o repositório remoto foi associado.



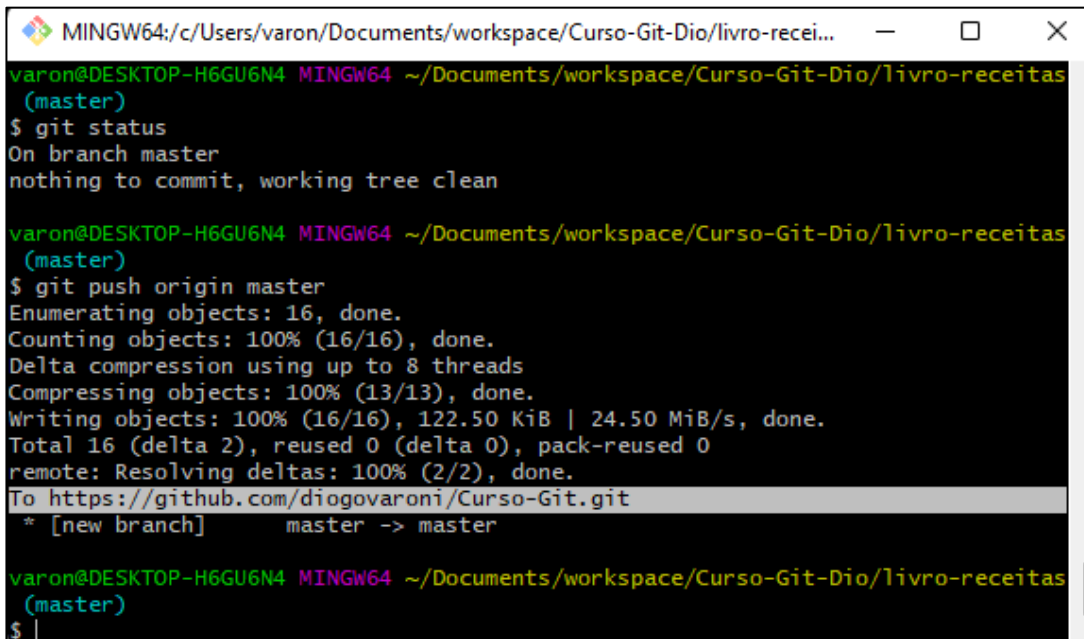
```
MINGW64:/c/Users/varon/Documents/workspace/Curso-Git-Dio/livro-recei...
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ git remote add origin https://github.com/diogovaroni/Curso-Git.git

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ git remote -v
origin https://github.com/diogovaroni/Curso-Git.git (fetch)
origin https://github.com/diogovaroni/Curso-Git.git (push)

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$
```

A palavra **origin** é apenas um apelido para que não precisemos digitar novamente todo o endereço e o link do repositório remoto. Poderia ser colocado qualquer nome nesse apelido mas por convenção usamos a palavra **origin**.

Agora vamos empurrar para o repositório remoto todos os arquivos feitos no repositório local. Digite o comando “**git push origin master**” para empurrar esse repositório.



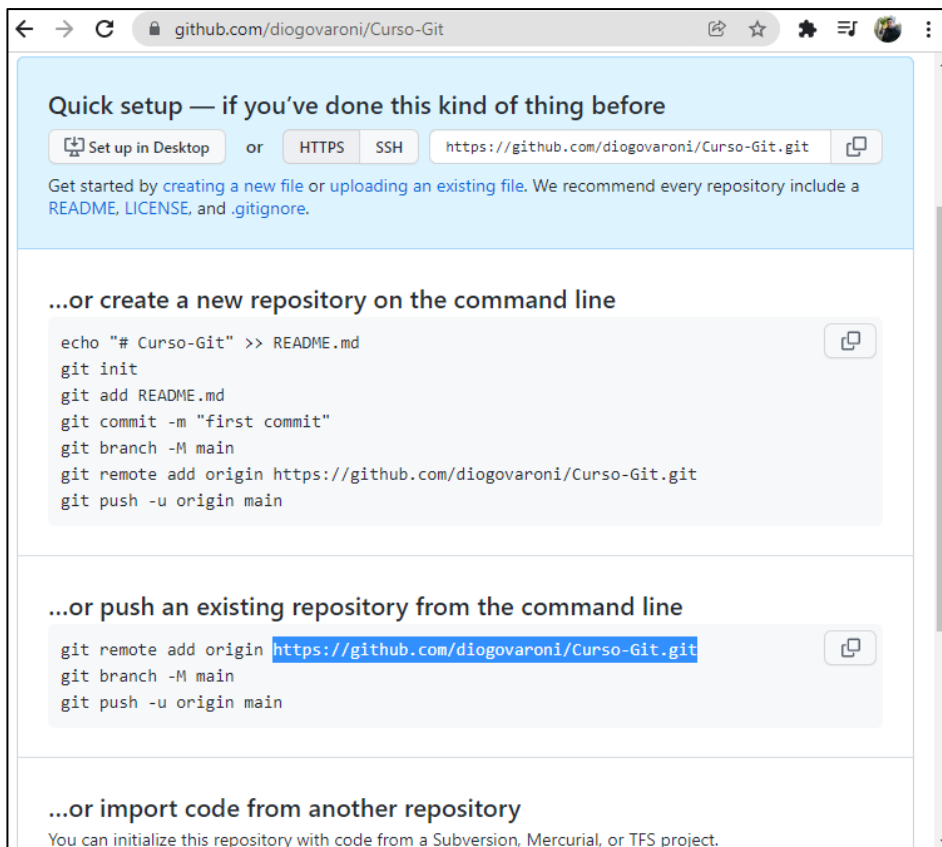
```
MINGW64:/c/Users/varon/Documents/workspace/Curso-Git-Dio/livro-recei...
varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ git status
On branch master
nothing to commit, working tree clean

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$ git push origin master
Enumerating objects: 16, done.
Counting objects: 100% (16/16), done.
Delta compression using up to 8 threads
Compressing objects: 100% (13/13), done.
Writing objects: 100% (16/16), 122.50 KiB | 24.50 MiB/s, done.
Total 16 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), done.
To https://github.com/diogovaroni/Curso-Git.git
 * [new branch]      master -> master

varon@DESKTOP-H6GU6N4 MINGW64 ~/Documents/workspace/Curso-Git-Dio/livro-receitas
(master)
$
```

Perceba na imagem que o Git informa o endereço do repositório remoto que recebeu o nosso repositório local. Agora vamos atualizar a página do GitHub para verificar as alterações.

- Antes de empurrar o repositório local.



The screenshot shows the GitHub repository page for 'Curso-Git' by 'diogovaroni'. The page is titled 'Quick setup — if you've done this kind of thing before'. It provides three options for setting up the repository: 'Set up in Desktop', 'HTTPS', and 'SSH'. The 'HTTPS' option is selected, showing the URL 'https://github.com/diogovaroni/Curso-Git.git'. Below this, there is a section titled '...or create a new repository on the command line' with a list of commands to initialize a new repository and push it to GitHub. Another section titled '...or push an existing repository from the command line' shows commands to add a remote origin and push the local repository. A final section titled '...or import code from another repository' mentions that the repository can be initialized with code from a Subversion, Mercurial, or TFS project.

Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH <https://github.com/diogovaroni/Curso-Git.git>

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# Curso-Git" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/diogovaroni/Curso-Git.git
git push -u origin main
```

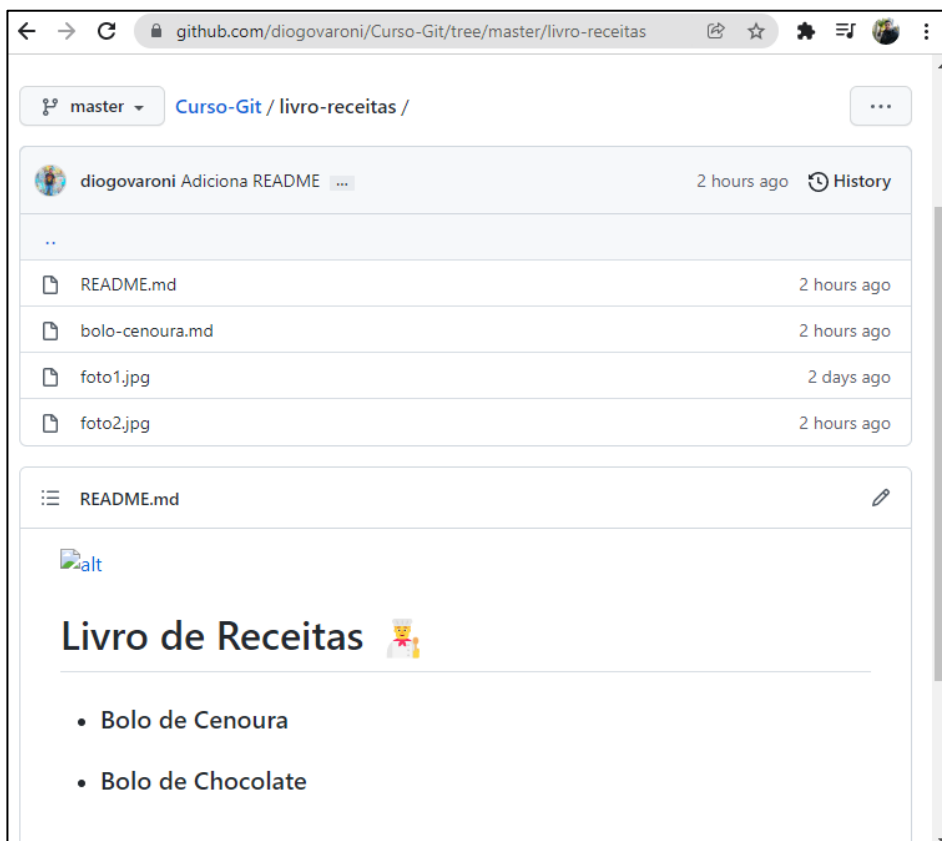
...or push an existing repository from the command line

```
git remote add origin https://github.com/diogovaroni/Curso-Git.git
git branch -M main
git push -u origin main
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

- Depois de empurrar o repositório local.



The screenshot shows the GitHub repository page for 'Curso-Git' by 'diogovaroni', specifically the 'livro-receitas' directory. The page shows a list of files and folders, including 'README.md', 'bolo-cenoura.md', 'foto1.jpg', and 'foto2.jpg'. The 'README.md' file is selected, showing its content. The content of 'README.md' is titled 'Livro de Receitas' and lists two recipes: 'Bolo de Cenoura' and 'Bolo de Chocolate'.

master Curso-Git / livro-receitas /

diogovaroni Adiciona README ... 2 hours ago History

..

README.md 2 hours ago

bolo-cenoura.md 2 hours ago

foto1.jpg 2 days ago

foto2.jpg 2 hours ago

README.md

alt

Livro de Receitas 🍪

- Bolo de Cenoura
- Bolo de Chocolate

7. Resolvendo Conflitos

Como os Conflitos Acontecem no GitHub e como Resolvê-los?

Nessa aula iremos ver como resolver os conflitos que ocorrem em um sistema de versionamento de código, como o Git. Os conflitos costumam ocorrer com certa frequência em times com vários desenvolvedores trabalhando em um mesmo repositório.

Quando uma pessoa baixa um código do GitHub para a máquina local ela terá exatamente o mesmo código que está no repositório remoto. Esse é o princípio do sistema distribuído de código que o Git proporciona. Se uma segunda pessoa baixar esse código do GitHub ela também terá o mesmo código. Enquanto ninguém fizer nenhuma alteração o código estará sincronizado.

Agora vamos supor que as duas pessoas abrem o código, cada uma em sua máquina local, e fazem alterações no código. A partir desse momento, os códigos nas máquinas não estão mais em sincronia: o código que está no GitHub é um, o código que está na máquina da primeira pessoa é outro e o código que está na máquina da segunda pessoa é outro.

Se a primeira pessoa depois de finalizar as modificações subir o código para o GitHub, então o código do GitHub passa a ser o que estava na máquina dessa pessoa. Quando a segunda pessoa tentar fazer um Commit e subir esse código para o GitHub, vai ser gerado o que chamamos de conflito de *merge*, que é quando o GitHub tenta juntar duas alterações na mesma linha.

No momento em que a segunda pessoa tenta empurrar o código para o GitHub o Git vai rejeitar a modificação porque no GitHub está um repositório mais recente. Devemos então puxar (**pull**) o código que está no GitHub, rodando o comando **“git pull origin master”**, que assim o Git irá fazer a integração entre esses 2 arquivos. Porém, se a alteração estiver na mesma linha, vai dar o conflito de merge. Para resolver devemos editar o arquivo manualmente, acertando as duas alterações feitas pelas duas pessoas. Devemos abrir o arquivo e ver qual é a versão correta e fazendo as edições específicas.

OBS.: para clonar um repositório do GitHub, basta digitar o comando **“git clone”** seguido do link do repositório.