GIT e GITHUB

Nesse **documento** iremos iniciar nossos trabalhos com as ferramentas de **versionamento de código** para que possamos trabalhar todo o **historio** e a **evolução** de nossos **projetos**. Conheceremos e aprenderemos a utilizar as principais ferramentas que são usadas no dia a dia de qualquer **profissional** na área de **desenvolvimento** de **softwares**, tanto de forma **individual**, quanto em **equipe**.

Iremos aprender a como criar repositórios, trabalhar o versionamento de nossos códigos, trabalhar em **equipes** de forma **síncrona** e sem alternar drasticamente o código principal, e trabalhar com o repositório online. As ferramentas que iremos utilizar são o **git** e o **github**.

ANEXO 1: GIT E GITHUB LOGO
LINK: https://alyssonmach.github.io/Minicurso-Git-e-GitHub/jup-not/AprendendoGitHub.html

Repositório

Imagine que você foi contratado para trabalhar em um projeto de jogo no qual você irá **trabalhar** em todo o projeto **só**. O projeto é bem **simples** e com **poucos recursos**, e para isso você cria uma **pasta simples** no seu **pc** com o nome do projeto e começou trabalhar nas **mecânicas** do jogo. Ao concluir as **mecânicas**, você salvou o projeto em uma nova pasta assim tendo duas pastas:

ANEXO 2: PASTAS DO JOGO

Ao final, você tem duas pastas, e **toda nova atualização** do projeto, você decide **criar uma pasta** com cada atualização para poder ter um histórico de versões e tendo cada vez mais pastas, ao **final** do projeto **termos várias pastas** que dará muito trabalho para analisar cada uma, caso precise voltar a um versão do código que não tenha um problema que talvez possa ocorrer.

ANEXO 3: PASTAS DO JOGO FINALIZADO

Agora imane se você trabalha em **equipe** e cada um da equipe usa esse tipo de método, ficaria **inviável** o **controle** dessas **informações**. Para podermos ter todos esse trabalho de controle de uma maneira mais simples, utilizamos os softwares de versionamento, no nosso caso iremos utilizar o **Git**.

1. O que é um repositório?

Um **repositório**, no contexto de controle de versão e desenvolvimento de software, é um **local** onde os **arquivos** de um **projeto** são **armazenados** e **gerenciados**. Ele contém todo o **código-fonte**, bem como o **histórico** de **todas** as **alterações** feitas no código ao **longo** do **tempo**. Repositórios podem ser **locais**, armazenados no **computador** do desenvolvedor, ou **remotos**, armazenados em servidores que permitem colaboração entre múltiplos desenvolvedores.

ANEXO 4: BANCO DE DADOS (SIMBOLIZANDO UM REPOSITÓRIO).
LINK: https://horadecodar.com.br/caracteristicas-e-tipos-de-banco-de-dados/

Componentes de um Repositório:

- **Arquivos e Diretórios:** Contém o código-fonte, documentação, arquivos de configuração, entre outros.
- **Histórico de Commits:** Registro de todas as mudanças feitas no repositório, permitindo rastrear quem fez o quê e quando.
- **Branches:** Linhas de desenvolvimento independentes que permitem que múltiplas funcionalidades ou versões sejam desenvolvidas simultaneamente.
- **Tags:** Marcadores usados para identificar versões específicas do código (por exemplo, versões de lançamento).
- **Staging Area (Índice):** Área intermediária onde as mudanças são preparadas antes de serem commitadas.

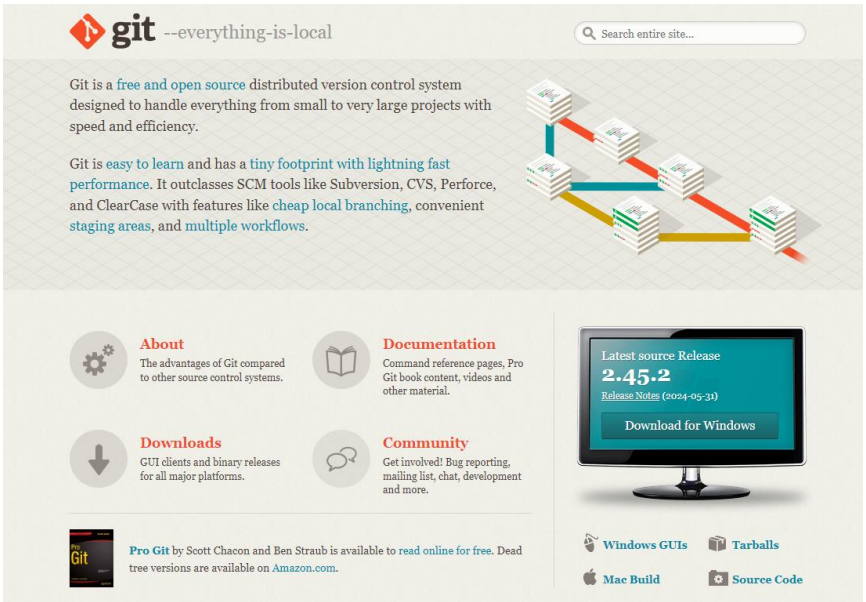
2. O que é o Git?

O **Git** é um **sistema** de controle de versão distribuído amplamente utilizado para **rastrear mudanças** no código-fonte durante o **desenvolvimento de software**. Ele permite que vários **desenvolvedores** trabalhem em um projeto de forma **simultânea sem interferir** uns nos **trabalhos** dos outros, mantendo um **histórico detalhado** de todas as **modificações** feitas no projeto.

ANEXO 5: GIT
LINK: https://git-scm.com

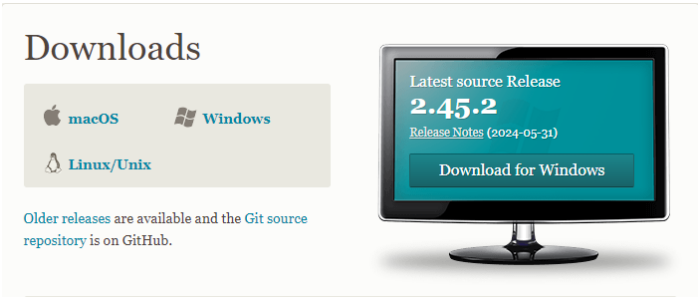
3. Como instalar o Git?

Para instalar o git me nossa máquina, é bem simples, basta acessarmos o site oficial do git clicando aqui.



ANEXO 5: SITE GIT
LINK: https://git-scm.com

Ao chegar na página inicial, você verá 4 links, **About (sobre)** que explicará o que é o git, **Documentation (Documentação)** que é onde mostra como usar o git, **Community (Comunidade)** que são as ferramentas de comunicação entre os desenvolvedores da ferramenta e outro desenvolvedores e por último **Download** que é onde iremos clicar. Escolha seu sistema operacional e execute o Download.

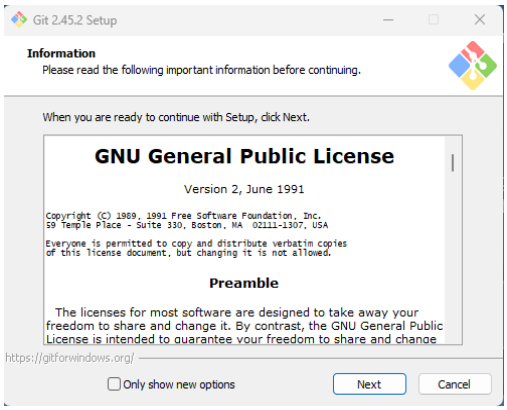


ANEXO 6: DOWNLOAD GIT
LINK: https://git-scm.com/downloads

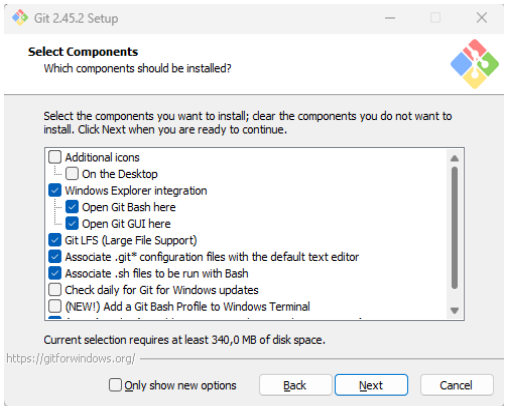
Siga o passo a passo abaixo para fazer a instalação correta do git:

1. Abra o executável que foi baixado:

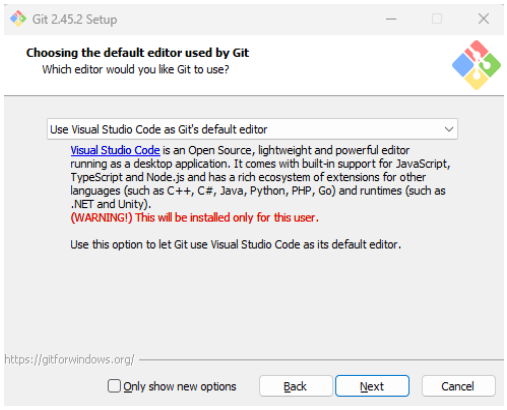
2. Desmarque a opção **“Only show new Options”** e clique em **“Next”**:



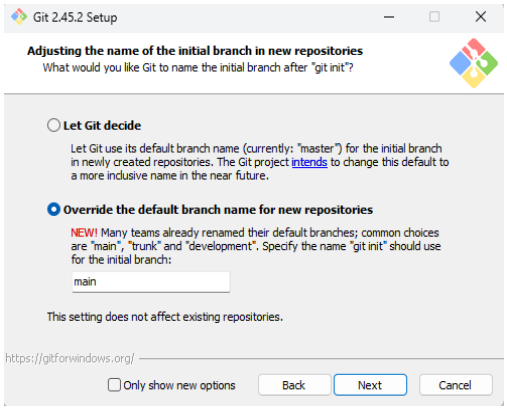
3. Verifique se todas as caixas abaixo estão marcadas:



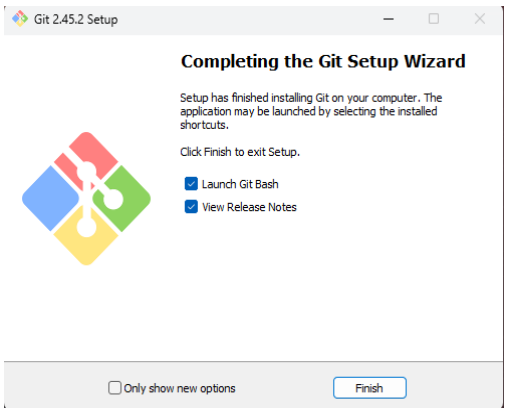
4. Mude o editor para **“Use Visual Studio Code as Git’s defalts editor”**:



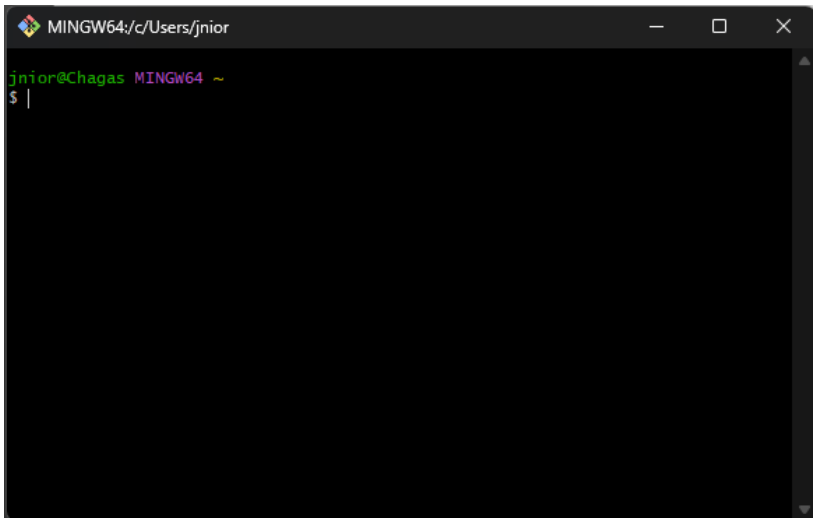
5. Deixe a opção **“Override the default branch name for new repositores”** marcada e com o nome de “main”:



6. Depois é so da **“Next”** até o **“Install”**:
7. Depois de instalado, marque as duas caixas e de uma **“Finish”**:



8. Irá abrir o Git Bash, onde enviaremos nossos código para poder trabalhar nossos repositórios:



Tudo certo para começarmos nossos trabalhos com o **git** para podemos criar as versões dos nossos projetos.

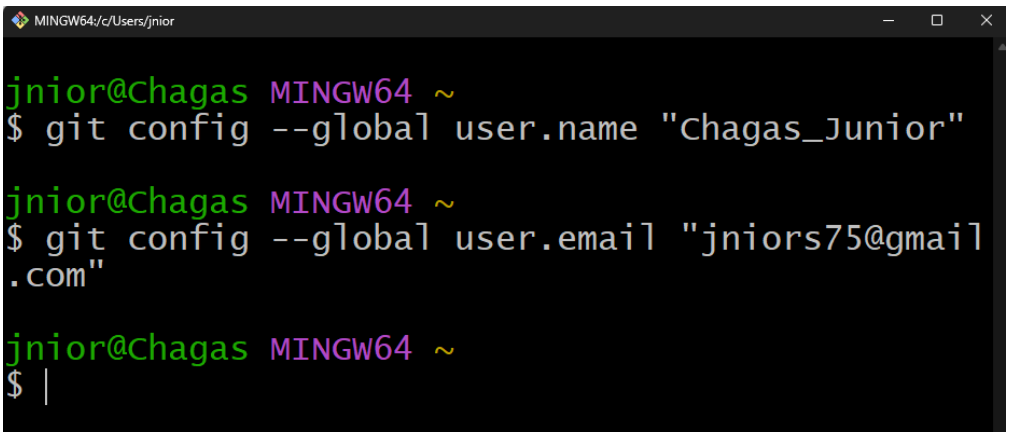
4. Configurando o Git

Antes de iniciar as atividades da criação de nosso repositório, será necessário configurar a nossa área de trabalho git com nossas informações para que ao começarmos a trabalhar em nossos repositório possamos criar históricos de trabalho, assim facilitando o rastreio das informações.

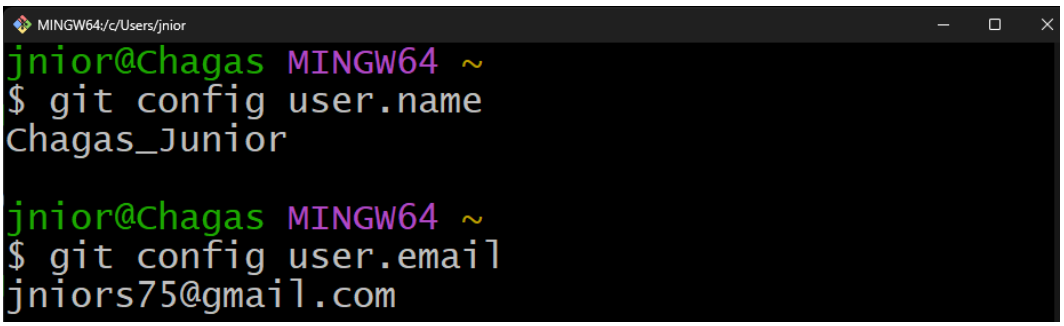
Essa configuração fazemos apenas uma vez (na sua máquina pessoal, nas máquinas pública é recomendado que insira ao chegar e apague quando sair) que será suas credenciais de uso, seu nome de usuário e um Email.

Abra o **git bash**:

1. insira o comando **git config --global user.name “SeuNomeSemEspaço”**:
2. insira depois o comando **git config --global user. email “SeuEmail”:**



3. Para verificar se deu certo a configuração do **nome** e do **email** use os comandos **git config user.name** e **git config user.email:**

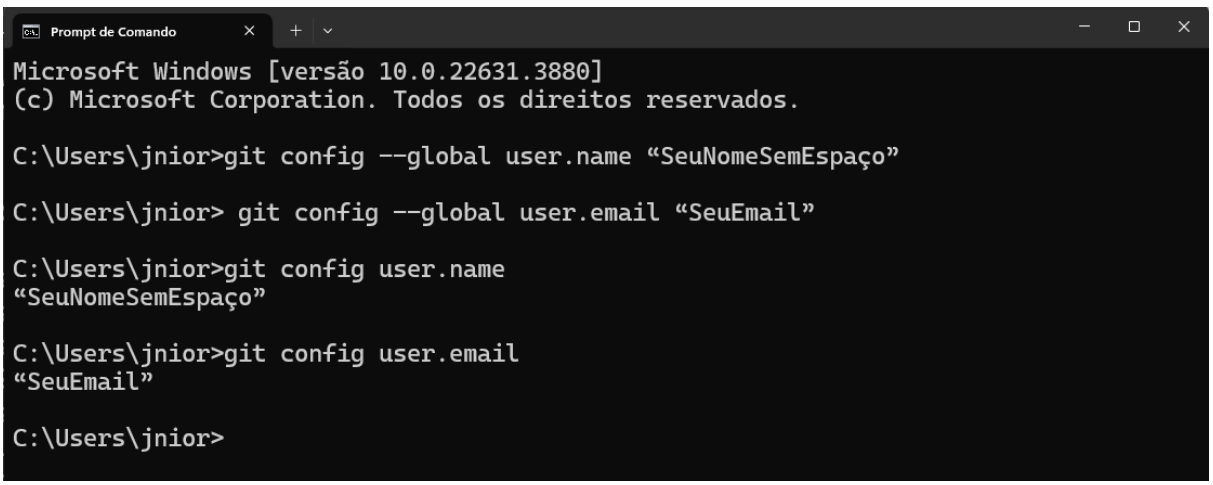


Obs.: Lembre-se de que se estiver em um computador público, antes de desligar o computador, passe as credenciais vazias:

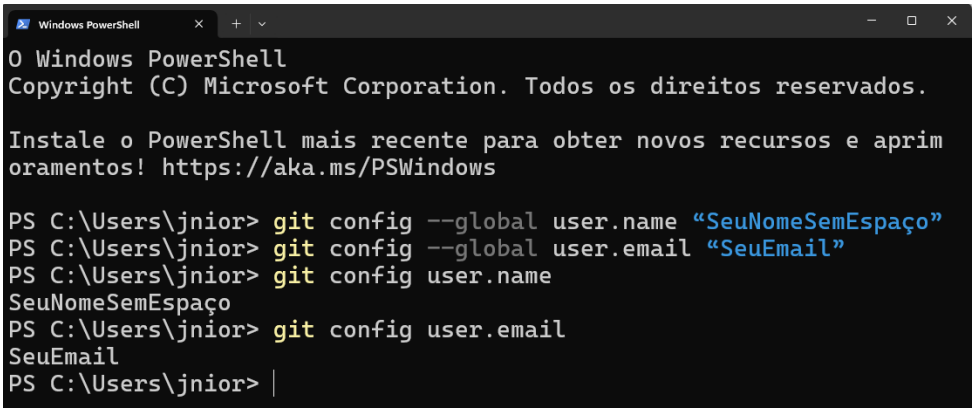
- ***git config --global user.name “”***
- ***git config --global user.email “”***

****Obs.: Também podemos usar o Prompt ou o PowerShell para mandar os comandos git (são até melhores):***

- ***Prompt:***



- ***Windows PowerShell:***



Tudo certo, git instalado e configurado, agora podemos trabalhar com ele em nossos projetos.

Comandos Git

Agora iremos aprender os principais comandos git para podemos começar a trabalhar em nosso primeiro repositório. Primeiro iremos ver os comandos e depois um passo a passo da criação, inspeção e implementação do nosso repositório local.

1. Comando Git Básicos

Aqui se encontram os comando iniciais para poder trabalhar com o **git**. Iremos aprender a como cria, verificar, adicionar arquivos e salvar as modificações.

- ***git init:*** Inicia um novo repositório na pasta que foi aberta.
- ***git status:*** Verifica o status do repositório em relação ao último commit;
- ***git add <arquivo>:*** Adiciona um ou mais arquivos (passe o "." no lugar do nome do arquivo para inseri todos os arquivos que não estão **trackeados**) para ser **trackeados** (rastreados);
- ***git commit -m "Título" -m "Descrição":*** Adiciona ao histórico as modificações feitas no repositório com a descrição das ações e alterações.

2. Comandos de Inspeção Git

Aqui se encontram os comando de inspeção e monitoramento do **git**. Aqui são os comandos que usaremos para rastrear e verificar as alterações feitas no repositório.

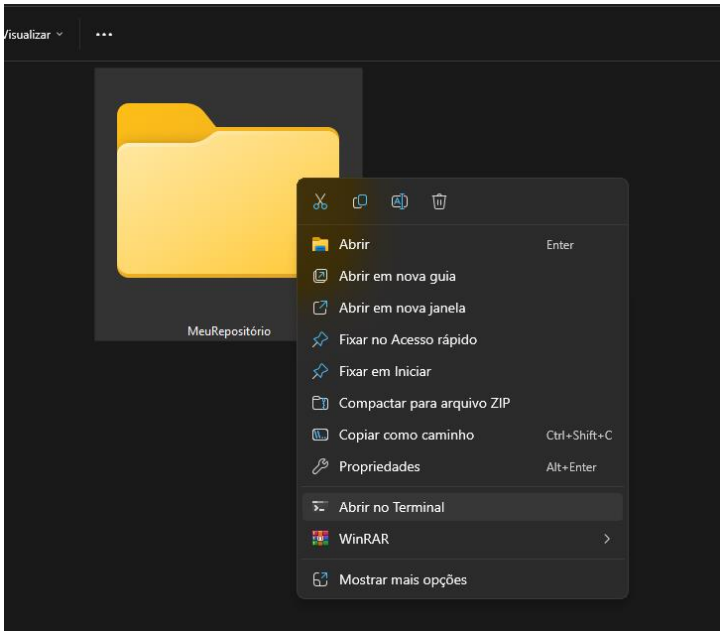
- ***git log:*** Mostra a lista de **commits**, quem fez, e horário que foi feito;
- ***git shortlog:*** Mostra a quantidade e os **commits** feitos (título apenas);
- ***git reflog:*** Mostra um histórico de todas as ações (referências) feitas em seu repositório local.
- ***git diff:*** Mostra a diferença entre os arquivos do repositório local e o remoto.
- ***git show:*** Mostra todas as informações presentes no commit.

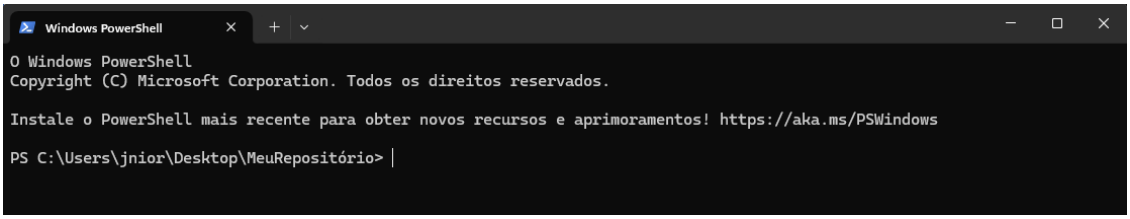
3. Praticando...

Para que possamos entender melhor cada comando, devemos praticar, então nessa etapa iremos criar nosso primeiro repositório local. Para isso vá até sua pasta, lá em **Documentos**, e:

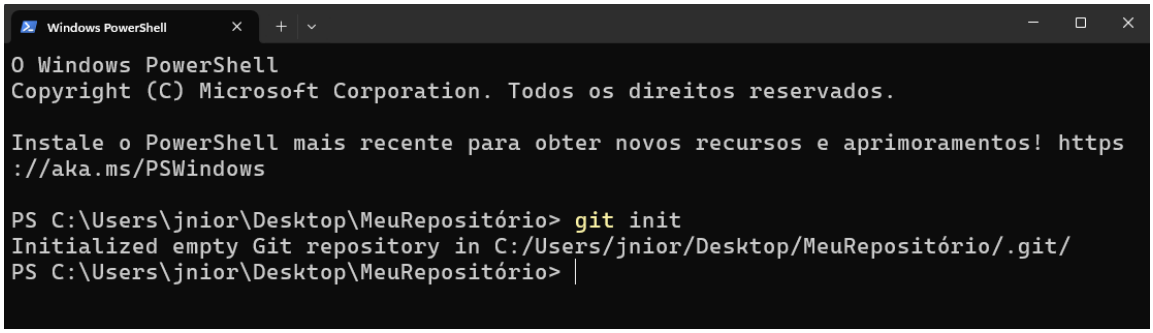
1. Crie uma **pasta** chamada de *"MeuRepositório"*:

2. Depois de criar a pasta, clique com o botão direito sobre ela e peça para abri com o **terminal**. Irá abrir o Prompt ou o Power Shell:



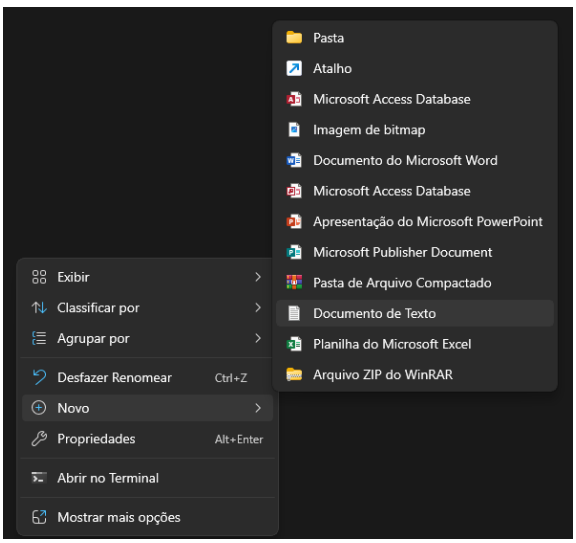


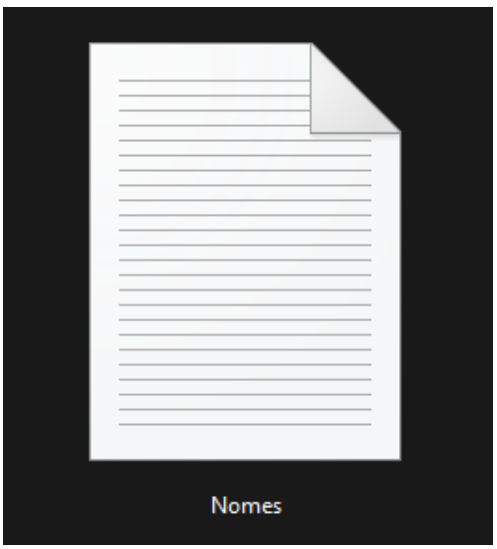
3. Agora iremos passar nosso primeiro comando. Iniciar o repositório com o **git init**.



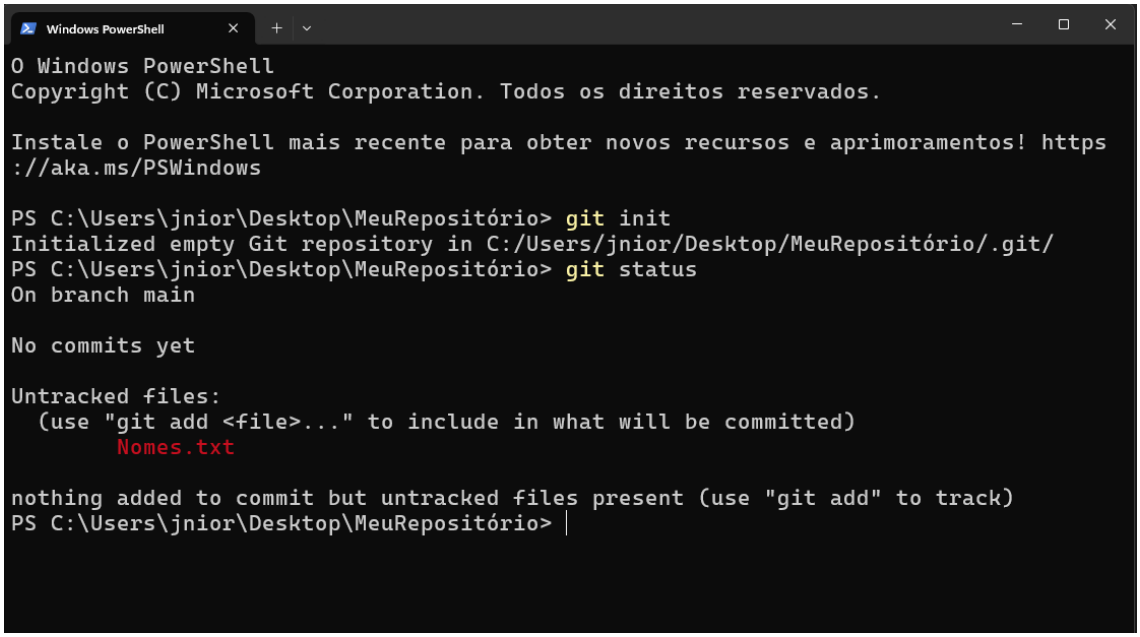
Com ele será criado uma pasta, "**.git**", dentro da pasta, mas esse arquivo não precisa ser modificado ou visto, por isso ele é oculto:

4. Agora iremos criar um arquivo .txt dentro da pasta para podermos ter um novo arquivo no "repo". Botão Direito >> Novo >> Documento de Texto >> Dê o nome de ***Nomes.txt***:

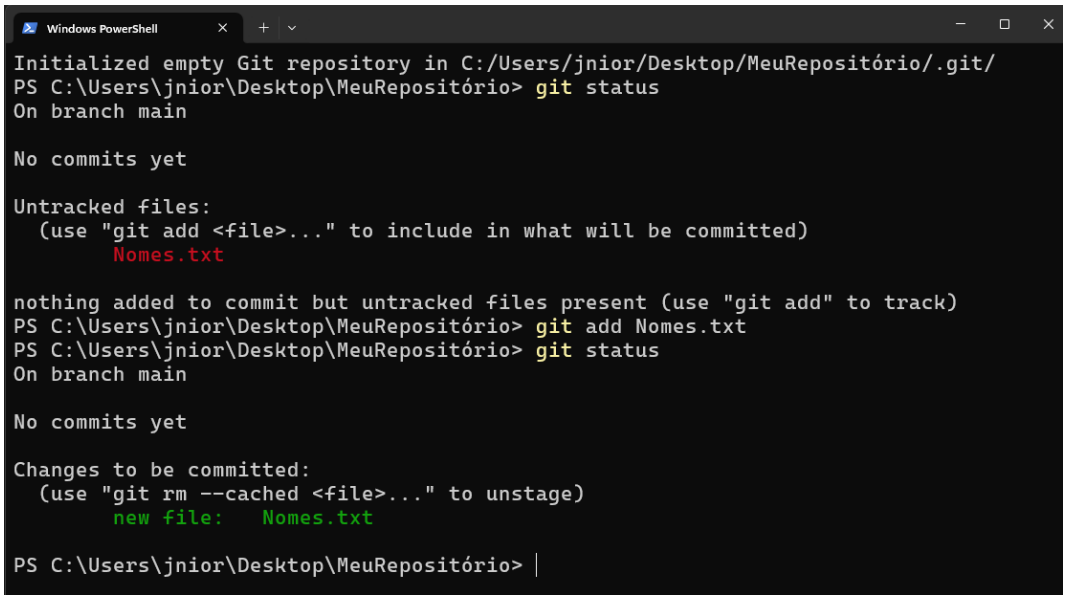




5. Ainda sem fazer modificações no arquivo, vá no prompt e passe o comando **git status**. Ele irá verificar como está o repositório:



6. Você receberá uma mensagem informando que o arquivo ***Nomes.txt*** está **Untracked** (não rastreado), para adicionar usamos o comando ***git add Nome.txt.*** Após adicionar, use novamente o **git status**. Agora o arquivo está sendo **Trackeado**:



Obs: Caso o seus **prompt** fique com muita **informação**, **poluindo** a tela, pode passar o comando **cls** que irá **limpar** todo o **console**. Você pode usar as **setas** para **cima** e para **baixo** para **navegar** nos comandos já passados e caso tenha **copiado** algum comando use o **botão direito** do **mouse** para **colar** assim como **selecionado** um comando e clicando com o **botão direito** ele **também cópia**.

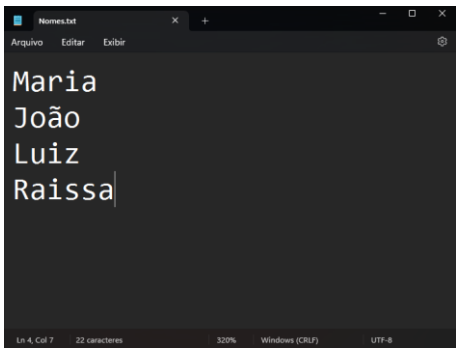
7. Agora que já temos nosso arquivo Nomes.txt trackeado, podemos commitar ele no repositório, para isso utilize o comando **git commit** (o comando commit necessita de uma flag, -m, para passar a mensagem. Se você usar o -m “” uma vez, você só passa um título para o commit, agora se usa duas vezes, -m “Título” -m “Descrição”, você consegue passar uma descrição detalhada do que foi feito nesse commit, nesse caso usaremos o título e a descrição ***git commit -m “Criando o Nomes.txt” -m “Foi executado apenas a criação do arquivo de texto com o nome de Nomes, mas ainda não foi adicionando nenhum conteúdo a esse arquivo.”:***

```
Windows PowerShell
PS C:\Users\jnior\Desktop\MeuRepositório> git commit -m “Criando o Nomes.txt” -m “F
oi executado apenas a criação do arquivo de texto com o nome de Nomes, mas ainda nã
o foi adicionando nenhum conteúdo a esse arquivo.”:
[main (root-commit) da47ebf] Criando o Nomes.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 Nomes.txt
PS C:\Users\jnior\Desktop\MeuRepositório>
```

8. Agora se der um **git status**:

```
Windows PowerShell
PS C:\Users\jnior\Desktop\MeuRepositório> git status
On branch main
nothing to commit, working tree clean
PS C:\Users\jnior\Desktop\MeuRepositório>
```

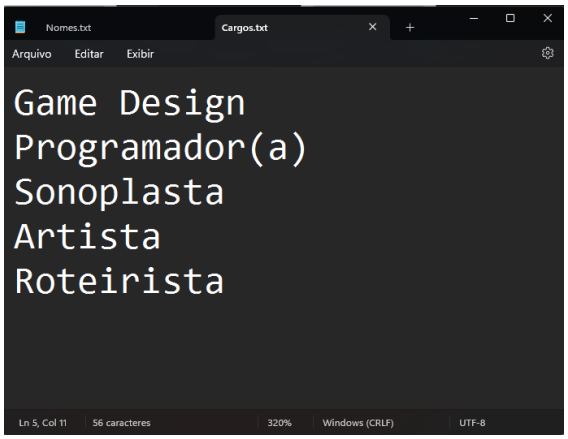
9. Depois do primeiro commit, iremos fazer alterações no arquivo **Nomes.txt**. Abra o arquivo e adicione quaisquer nomes que quiser. EX.:



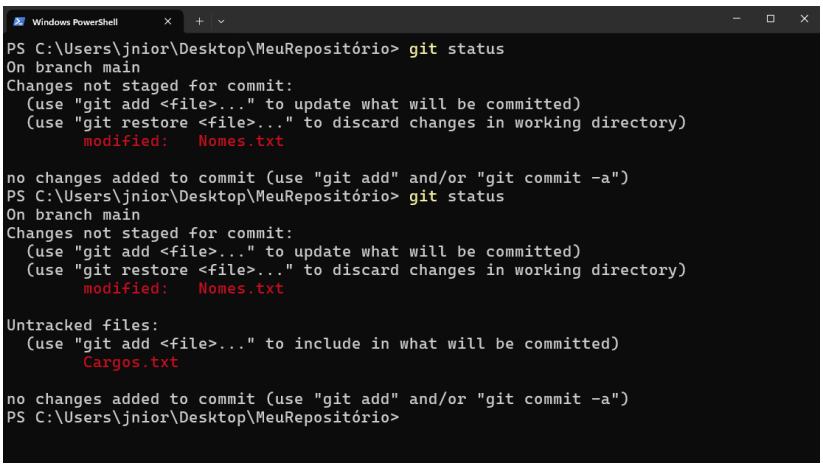
10. Salve o arquivo. Volte no **PowerShell** e de um **git status**:

```
Windows PowerShell
PS C:\Users\jnior\Desktop\MeuRepositório> git status
On branch main
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:   Nomes.txt

no changes added to commit (use "git add" and/or "git commit -a")
PS C:\Users\jnior\Desktop\MeuRepositório>
```

11. Antes de commitar essa modificação, vamos criar outro arquivo **.txt** e dê o nome de **Cargos.txt** e preencha-o também:



12. Salve o arquivo. Volte no PowerShell e de um **git status**:



13. Agora temos um arquivo **modificado** (**Nomes.txt**) e um **novo arquivo** que ainda **não** está **trackeado** (**Cargos.txt**) e precisamos **adicionar** os dois para **commitar** todo o projeto de uma **só vez**, so que para isso devemos adicionar cada nova modificação antes usando o **git add** e passando o nome do arquivo. Só que nesse caso demos dois arquivo, que ainda é pouco e dá para adicionar de forma individual, mas e se fossem mais arquivos, ter que adicionar um por um não é viável. Para isso usando o comando **"git add ."** e ele irá adicionar todas as novas modificações de uma vez. Depois de adicionado **commite** a modificação:

```
Windows PowerShell
PS C:\Users\jnior\Desktop\MeuRepositório> git add .
PS C:\Users\jnior\Desktop\MeuRepositório> git commit -m "Modificando o Nomes.txt e
Criando o Cargo.txt" -m "Foram adicionado quatro novos membros a equipe e foi criad
o um novo arquivo, Cargos.txt, para armazenar os cargos da empresa.":
[main d37473c] Modificando o Nomes.txt e Criando o Cargo.txt
 2 files changed, 9 insertions(+)
 create mode 100644 Cargos.txt
PS C:\Users\jnior\Desktop\MeuRepositório> git status
On branch main
nothing to commit, working tree clean
PS C:\Users\jnior\Desktop\MeuRepositório>
```

Finalizando nossa prática inicial, use os comando de inspeção para verificar o estado do repositório.

- ***git log;***

```
Windows PowerShell
PS C:\Users\jnior\Desktop\MeuRepositório> git log
commit d37473c6aab3d1a6ea29a4e9470509dfa18ed754 (HEAD -> main)
Author: ChagasJunior <jniors75@gmail.com>
Date:   Tue Jul 23 16:03:44 2024 -0300

    Modificando o Nomes.txt e Criando o Cargo.txt

    Foram adicionado quatro novos membros a equipe e foi criado um novo arquivo, Ca
rgos.txt, para armazenar os cargos da empresa.

commit da47ebf31d4770074ff6cbd82b02893ef75611a4
Author: ChagasJunior <jniors75@gmail.com>
Date:   Tue Jul 23 13:34:54 2024 -0300

    Criando o Nomes.txt

    Foi executado apenas a criação do arquivo de texto com o nome de Nomes, mas ain
da não foi adicionando nenhum conteúdo a esse arquivo.
PS C:\Users\jnior\Desktop\MeuRepositório>
```

- ***git shortlog;***

```
Windows PowerShell
PS C:\Users\jnior\Desktop\MeuRepositório> git shortlog
ChagasJunior (2):
      Criando o Nomes.txt
      Modificando o Nomes.txt e Criando o Cargo.txt

PS C:\Users\jnior\Desktop\MeuRepositório>
```

- ***git reflog;***

```
Windows PowerShell
PS C:\Users\jnior\Desktop\MeuRepositório> git reflog
d37473c (HEAD -> main) HEAD@{0}: commit: Modificando o Nomes.txt e Criando o Cargo.
txt
da47ebf HEAD@{1}: commit (initial): Criando o Nomes.txt
PS C:\Users\jnior\Desktop\MeuRepositório>
```

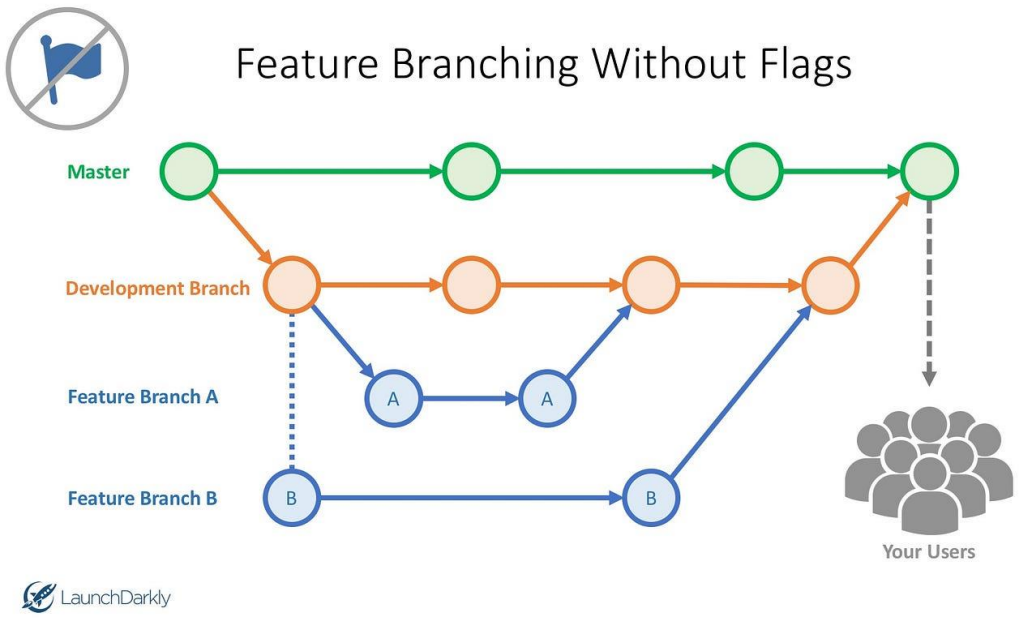
- ***git show:*** Tecle **Enter** para descer e **Q** para sair.

```
commit d37473c6aab3d1a6ea29a4e9470509dfa18ed754 (HEAD -> main)
Author: ChagasJunior <jniors75@gmail.com>
Date:   Tue Jul 23 16:03:44 2024 -0300

    Modificando o Nomes.txt e Criando o Cargo.txt

    Foram adicionado quatro novos membros a equipe e foi criado um novo arquivo, Cargos.txt, para armazenar os cargos da empresa.

diff --git a/Cargos.txt b/Cargos.txt
new file mode 100644
index 0000000..e944c89
--- /dev/null
+++ b/Cargos.txt
@@ -0,0 +1,5 @@
+Game Design
+Programador(a)
+Sonoplasta
+Artista
+Roteirista
\ No newline at end of file
diff --git a/Nomes.txt b/Nomes.txt
index e69de29..8ffbd91 100644
--- a/Nomes.txt
+++ b/Nomes.txt
@@ -0,0 +1,4 @@
+Maria
+João
+Luiz
+Raissa
\ No newline at end of file
(END)
```

Completamos o básico de **git** e já sabemos como criar um repositório, como verificar o status dos arquivos e a como salvar as alterações e criar um histórico de modificações e assim podendo rastreá-los através de um **ID** único chamado de **hash**.

Branches

Agora imagina que você precisa trabalhar em **mecânicas** mais **complexas** em seu projeto que possam **interferir negativamente** em seu projeto, caso de algum **problema** na **implementação**, isso influenciará em muito trabalho de **manutenção** no seu código principal. Para evitar esse **problema**, os **repositórios** existem uma alternativa chamada **Branch**.

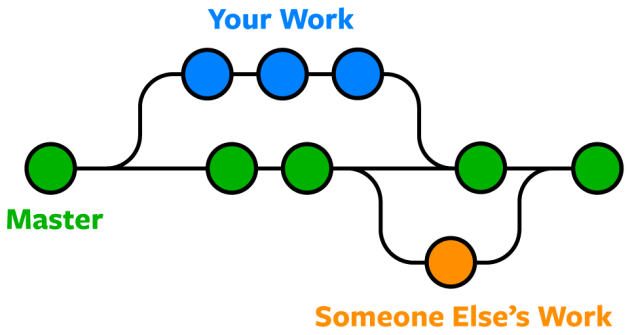


ANEXO 7: BRANCHES
LINK: https://share.atelie.software/por-que-você-deveria-diminuir-o-uso-de-branches-c2c07c1ccb07

1. O que são Branches?

Branches são ferramentas do repositórios que permitem que os desenvolvedores criem **linhas do tempo paralelas** para fazerem **modificações** com **segurança** para evitar fazer **modificações** bruscas na **linha do tempo principal**. **Branches** significa **galhos**, ou seja, na árvore principal do seu projeto, terá galhos semelhantes que possam sofrer alterações sem modificar o galho principal.

A **branch** principal do **git** é a **main**, mas você pode criar uma com o nome que quiser e deixar em historio para depois unir a **main** quando a implementação estiver completa e validada.



ANEXO 8: BRANCHES
LINK: https://www.nobledesktop.com/learn/git/git-branches

2. Comandos de Gerenciamento de Branches

Agora aprenderemos a como trabalhar com as branches. Abaixo está a lista de comandos para se usar com as branches:

- **git branch <nome_da_branch>:** Cria uma nova branch;
- **git branch:** Lista as branches do repositório em mostra qual está;
- **git checkout <nome_da_branch>:** Muda para a branch especificada;
- **git checkout -b <nome_da_branch>:** Cria uma nova branch e muda para ela;
- **git branch -d <nome_da_branch>:** Deleta a branch especificada.
- **git marge <nome_da_branch>:** Junta a branch especificada na branch atual.

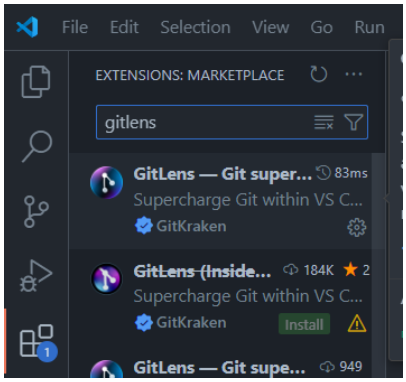
3. Praticando...

Para memorizar bem os comandos, devemos sempre praticar, para isso usaremos o exemplo anterior, **MeuRepositorio**, porém agora trabalharemos direto no Visual Studio Code, onde passaremos maior parte do tempo.

Antes de iniciamos a prática, precisamos montar o espaço de trabalho no VS Code, instalando a extensão **Gitlens**, pois ela irá nos auxiliar na utilização da ferramenta git em nosso projetos:

1. Abra o VS Code:

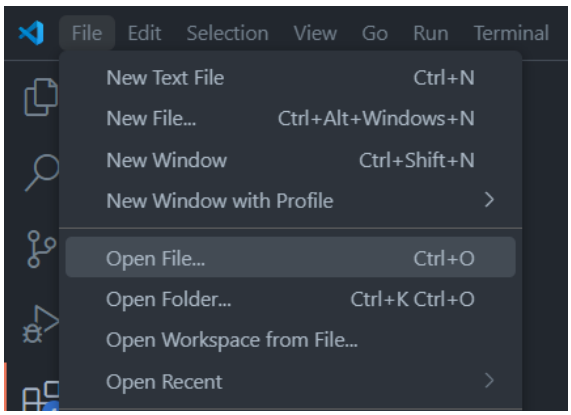
2. Vá na aba Extensões e na barra de pesquisa digite “GitLens” e baixe a extensão oficial do GitKraken:



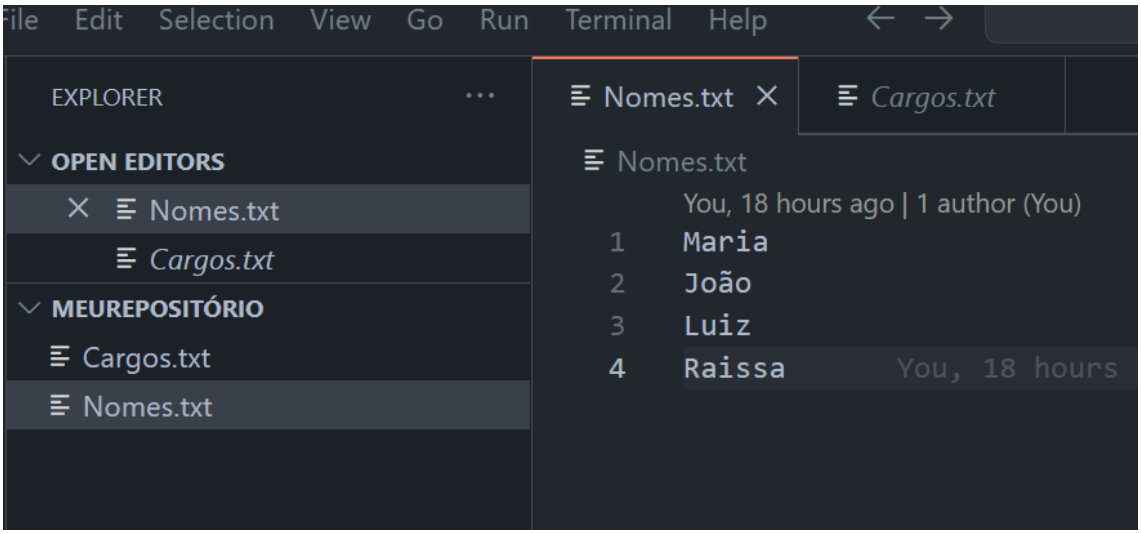
3. Tudo certo para podermos trabalhar com a ferramenta git no VS Code.

Agora que já temos o GitLens Instalado, já podemos trabalhar no repositório que criamos anteriormente:

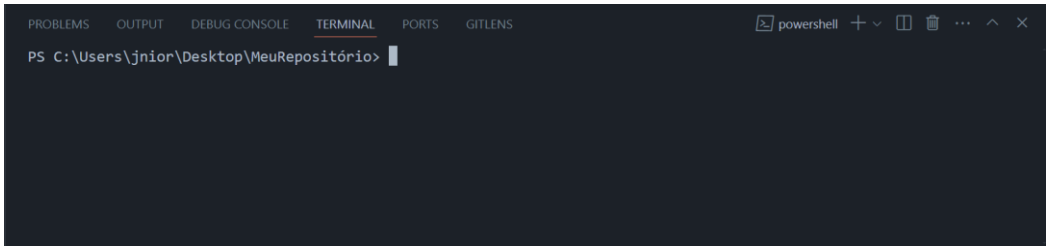
1. Abra a pasta do repositório no VS Code: File >> Open Folder >> Selecione a pasta do seu repositório:



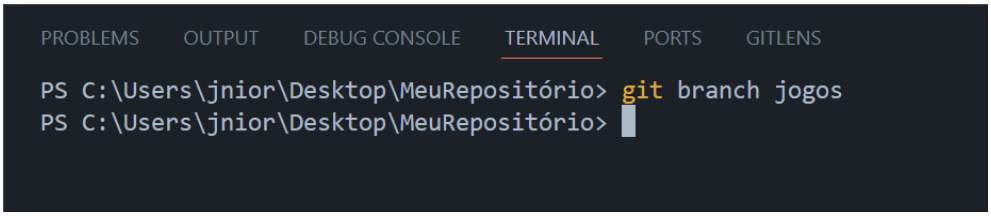
2. Depois de abrir a pasta, abra os dois arquivos, **Nomes.txt** e **Cargos.txt**:



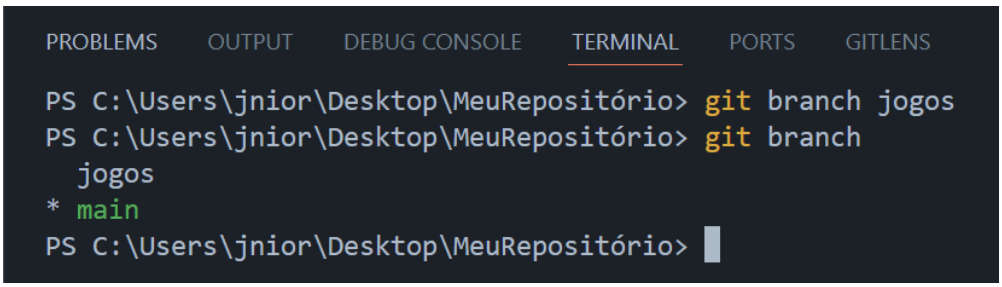
3. Para abrir o terminal do VS Code use o atalho ***Ctrl + '***:



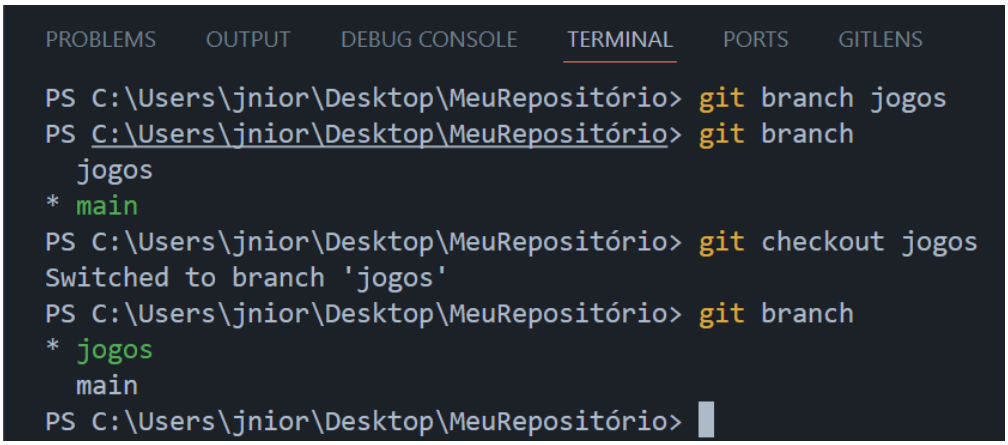
4. Agora que já temos nossa área de trabalho pronto, iremos criar uma nova branch para trabalhar com outro arquivo separado. Crie uma nova branch com o comando **git branch jogos**:



5. Para verificar se o branch foi criada, use o comando **git branch**, para listar as branches:

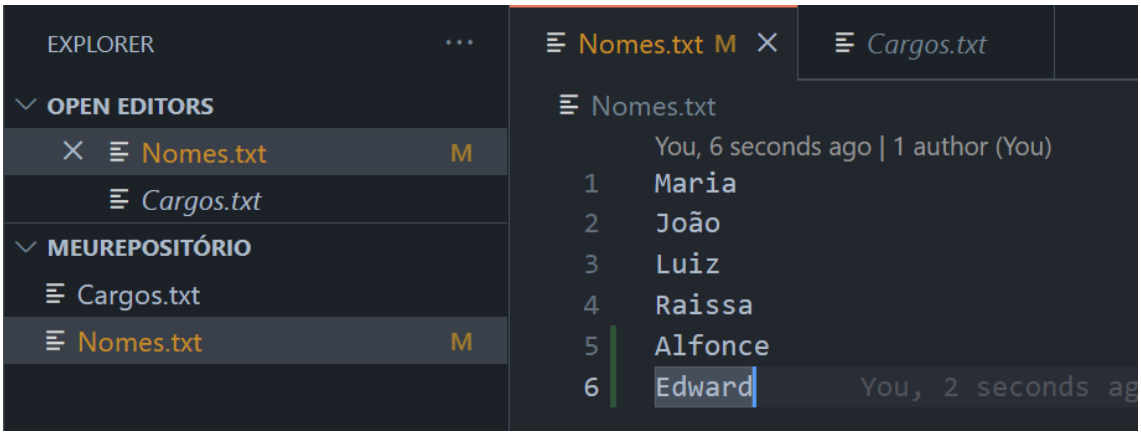


6. A branch foi criada, porém estamos ainda na branch main. Para trocar use o comando **git checkout jogos**. Depois use novamente o **git branch**:



7. Pronto, já estamos na branch **jogos**, podemos fazer nossas modificações sem alterar a branch **main**.

8. Agora iremos fazer algumas alterações no projeto. Primeiro, insira mais dois nomes no arquivo **Nomes.txt** e salve o arquivo (Verifique se a opção de auto save do VS Codo está habilitada para não precisa ficar salvando por comando):

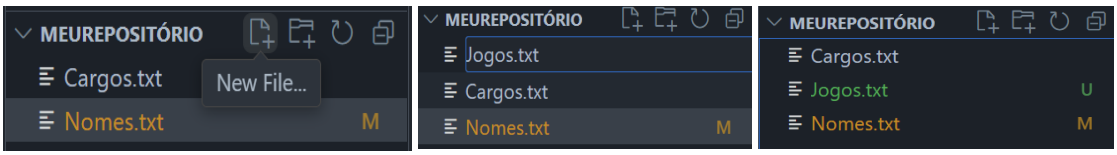


***Obs.:** Repare que quando modificamos qualquer arquivo no VS Code, ele fica marcado com um M. Isso significa que a extensão GitLens está dizendo que o arquivo ainda não sofreu modificações, então precisamos adicionar novamente usando o **git add** (mas não faremos isso agora).*

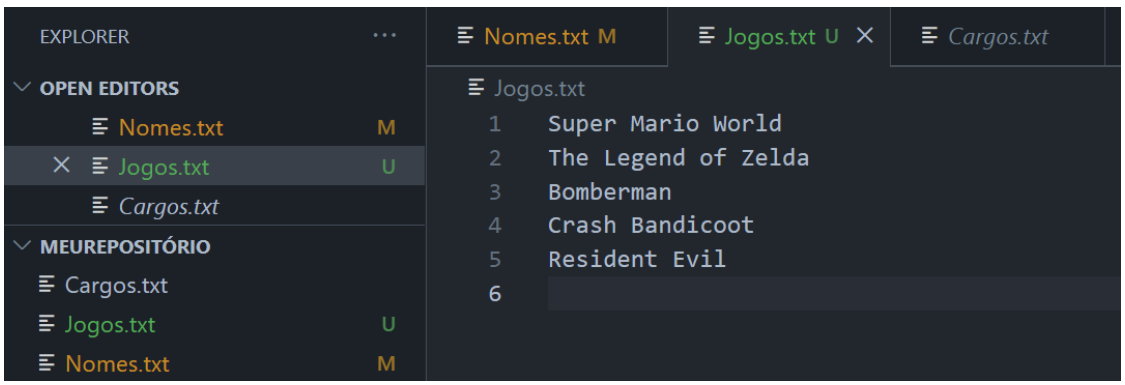
****Obs.:** As letras que podem aparecer são:*

- *U – Untracked (não rastreado, arquivo novo);*
- *A – Tracked (rastreado, adicionado);*
- *M – Modified (Modificado);*

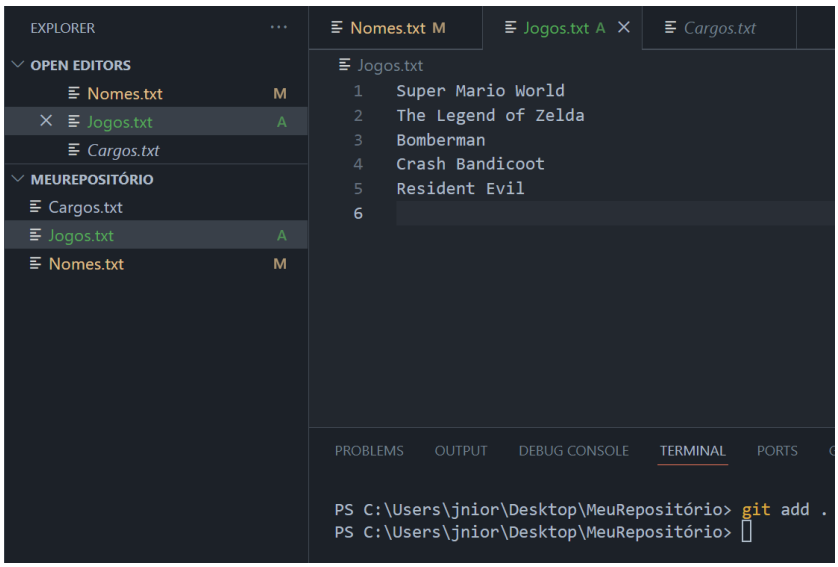
9. Agora criaremos outro arquivo chamado **Jogos.txt**. Vá em New Files >> Digite **Jogos.txt**:



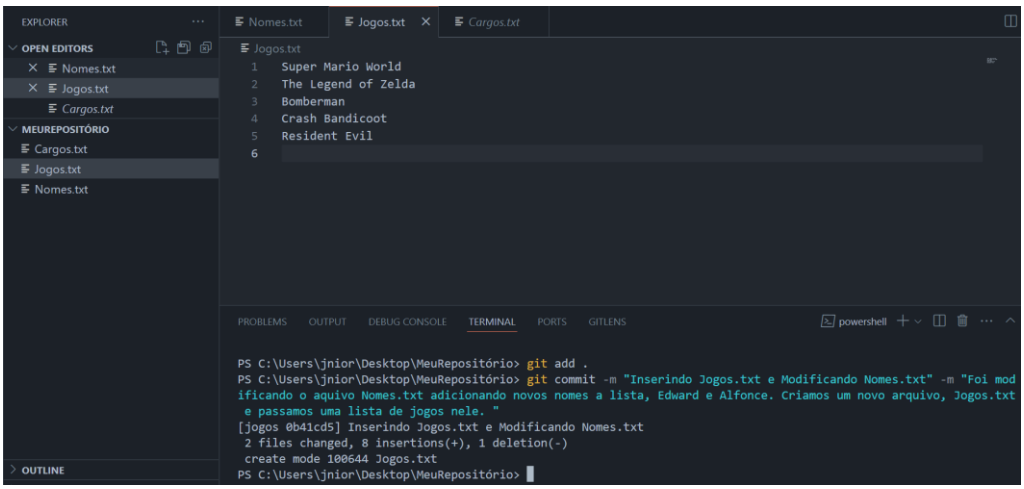
10. Abra o arquivo e passe os nomes dos jogos:



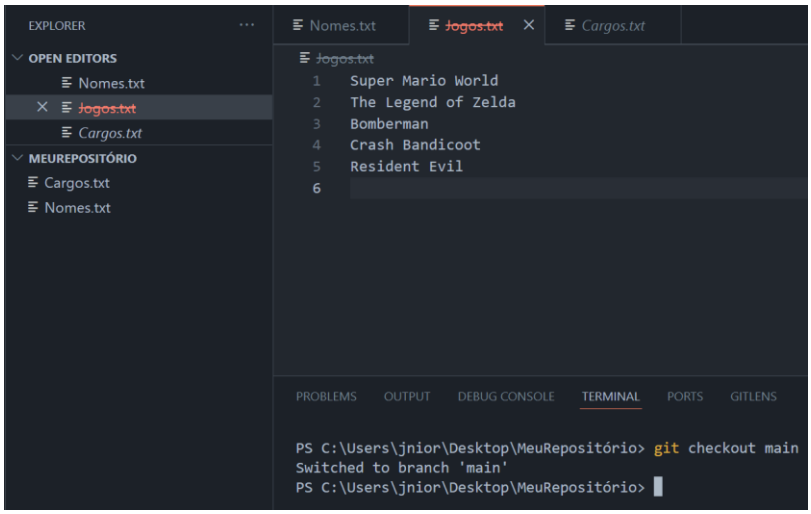
11. Agora de um ***git add .*** para traquear todos os arquivos:



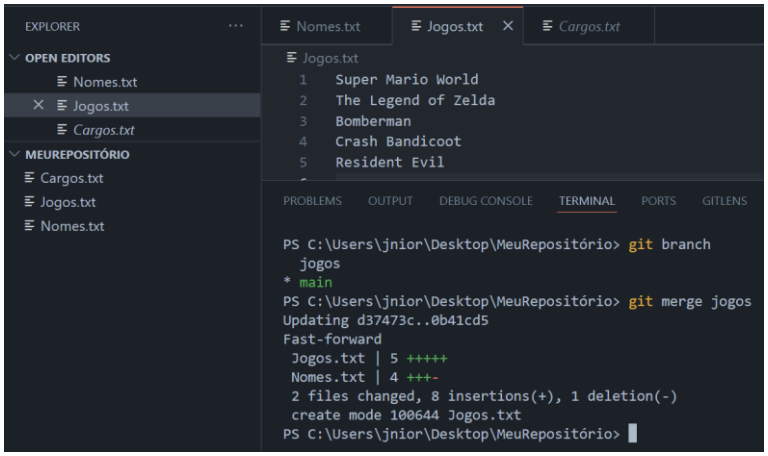
12. Agora o arquivo **Jogos.txt** está com um A, significa que já está sendo Trackeado. Dê o commit com as novas alterações:



13. Tudo certo, modificações commitadas. Agora vamos ver como está a branch **main**, dê um **git checkout main**:



14. Repare que o arquivo **Jogos.txt** foi riscado nos arquivos aberto e não existe na pasta, isso porque ele so existe na **branch jogos** que criamos. Para que essas modificações passam ir para o projeto principal, precisamos **juntar** as duas branches, usando o comando **merge**. Lembre-se de estar na branch que receberá as modificações, no caso a main. Use o **git branch** para saber se realmente está na main e depois use o **git merge jogos:**



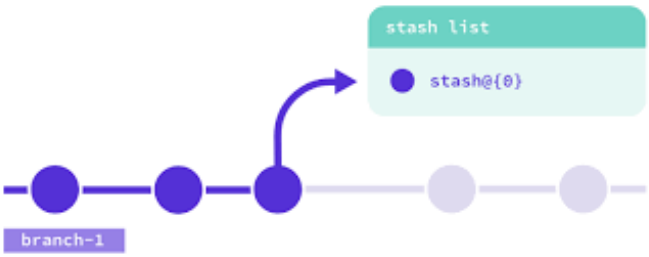
15. Pronto, todas as alterações e adiciones foram feitas na branch main, agora as duas tem os mesmos arquivos.

***Obs.:** Lembre-se de que se estiver trabalhando em mais de uma funcionalidade, use **branches diferentes**.

****Obs.**: Caso haja **duas branches** sendo trabalhadas **simultâneas** da **main**, e uma **der merge** na **main**, a outra precisa **receber** o **merge** da **main** para que o projeto esteja sempre **sincronizado**.

Stash

Imagine que você está fazendo uma **funcionalidade**, em uma **branch**, mas precisou trocar para a **branch main**, pois ocorreu um **bug** e ele precisa ser **corrigido** o mais **rápido** possível, porém você **não consegue** trocar de branch **sem commitar** as alterações antes. Para isso utilizamos os **Stash**



ANEXO 9: GIT STASH
LINK: https://css-irl.info/how-git-stash-can-help-you-juggle-multiple-branches/

1. O que é uma Stash?

Uma stash é basicamente uma funcionalidade do git que permite que possamos salvar modificações sem precisa commitar. Ao cria uma stash, ele volta ao último commit criado, permitindo que possa ser feitos nossas mudanças e caso queiro voltar ao que tinha feito antes, basta desfazer a stash através do seu identificador.

2. Comandas Git Stash

Agora que aprendemos o que são stash, vamos ver os seus comandos abaixo:

- ***git stash:*** Salvar mudanças não commitadas (modificações e arquivos rastreados).
- ***git stash save "Mensagem descritiva":*** Salvar mudanças com uma mensagem descritiva.
- ***git stash -u:*** Incluir arquivos não rastreados (novos arquivos não adicionados ao índice).
- ***git stash list*****:** Ver lista de todas as entradas de stash.
- ***git stash show -p*****:** Mostras as todas as modificações salvas no stash mais recente ao diretório.
- ***git stash show -p stash@{0}:*** Mostras as todas as modificações salvas no stash específico usando o índice
- ***git stash apply:*** Aplicar o stash mais recente ao diretório de trabalho.
- ***git stash apply stash@{0}:*** Aplicar um stash específico usando o índice.
- ***git stash pop:*** Aplicar o stash mais recente e o remove da lista.
- ***git stash pop stash@{0}:*** Aplicar um stash específico e o remove da lista.
- ***git stash drop stash@{0}:*** Remover um stash específico sem aplicá-lo.
- ***git stash clear:*** Remover todos os stashes.

3. Praticando...

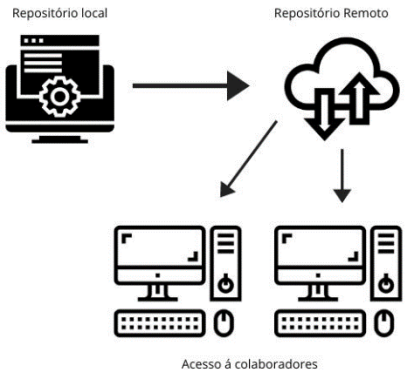
As aplicações com o stash são muito específicas, pois esse comando irá salvar informações não commitadas a fim de deixar um historio de modificação coso o desenvolvedor se arrendar ou precise voltar para um trecho que via como melhor alternativas. Tome cuidado ao usar os stash, pois eles podem conflitar com algum código que você tenha commitado antes, pois ele aloca os código nas linhas especificadas no momento da criação do stash.

Um bom exemplo da utilização do stash seria: Você está desenvolvendo uma feature em uma branch e precisa resolver um bug urgente em outra, contudo o que você já vez ainda não é suficiente para poder commitar ao projeto e poder ir para a outra branch, para isso use o stash, arrume o bug na outra branch, commite e volte a trabalhar na feature usando o **git stash apply ou git stash pop**.

***Obs.:** Cuidado para não fazer alterações que possam da conflitos ao reverter o stash.

GitHub

Aprendemos a utilização do repositório de forma local, no qual apena o desenvolvedor, como o "repo" em sua máquina consegue visualizar e modificar os código. Mas para trabalhos em equipe, ou até mesmo como forma de portifólio, essa não é a melhor maneira de trabalhar e é aí que entra o **github.**



ANEXO 11: GITHUB
LINK: https://blog.betrybe.com/git/git-push/

1. O que é o GitHub?

O GitHub é uma plataforma de hospedagem de código-fonte que utiliza o sistema de controle de versão Git. Ele permite que desenvolvedores colaborem em projetos, compartilhem código, gerenciem versões e acompanhem o histórico de mudanças. Além de hospedar repositórios públicos e privados, o GitHub oferece ferramentas para colaboração, como pull requests, revisões de código, e issues para rastreamento de tarefas e bugs.

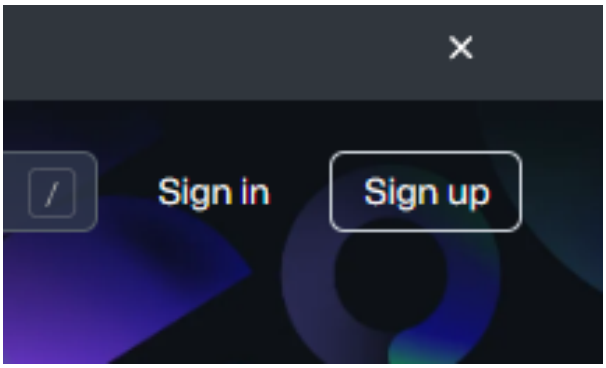
É amplamente utilizado na comunidade de desenvolvedores para projetos open source e por empresas para o desenvolvimento e gerenciamento de software. É uma ótima ferramenta para se usar como portifólio para quem atua na área de desenvolvimento de softwares, pois pode ser visto por diversas pessoas da área.

ANEXO 11: GITHUB
LINK: https://github.blog

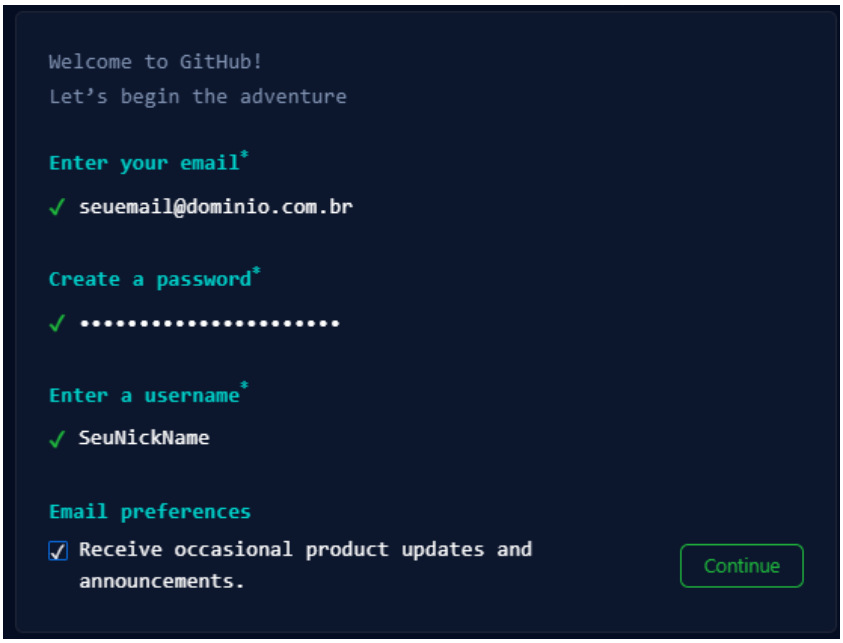
2. Criando uma conta no GitHub

Como o **GitHub** é uma plataforma online, precisamos fazer uma conta para podermos utilizar seus serviços. O **github** permite que possamos usar uma versão **gratuita** (que já atente a maioria dos públicos) e uma versão **paga** (que geralmente é utilizado por empresas).

1. Primeiro, vamos acessar o https://github.com e clique em **Sing Up**:

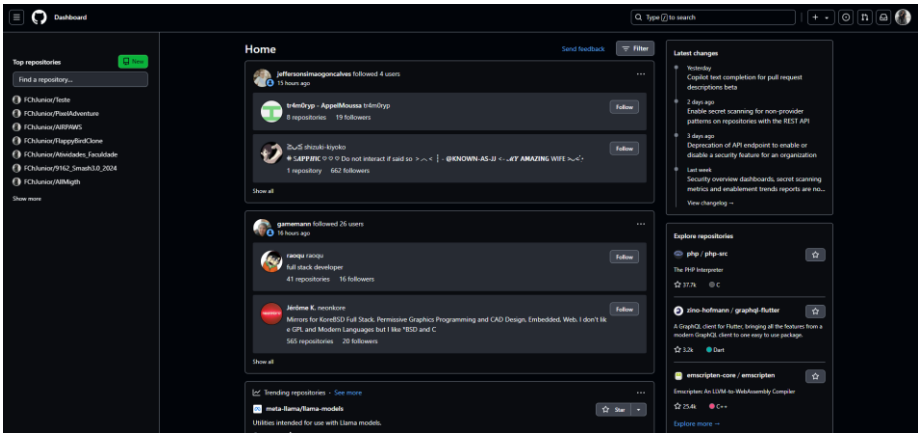


2. Insira um email valido e uma senha e um nome de usuário a sua escolha:



3. Faça a verificação:

4. Tudo certo, entre na sua conta agora. E essa é a página inicial do GitHub:

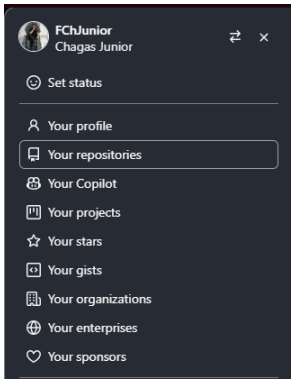


Tudo certo, agora já podemos trabalhar com nossos repositórios de forma remato e tendo a possibilidade de trabalhar em equipe e podendo expor nosso trabalhar.

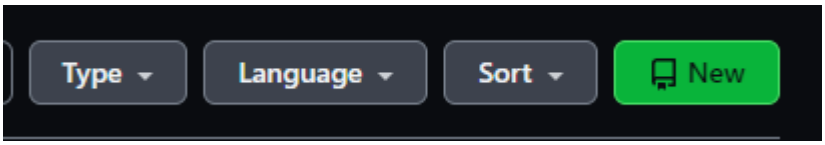
3. Como criar um repositório Remoto?

Vamos criar nosso primeiro repositório remoto para que possamos subir o local que tínhamos feito anteriormente. É bem simples e fácil, vamos seguir os seguintes passos:

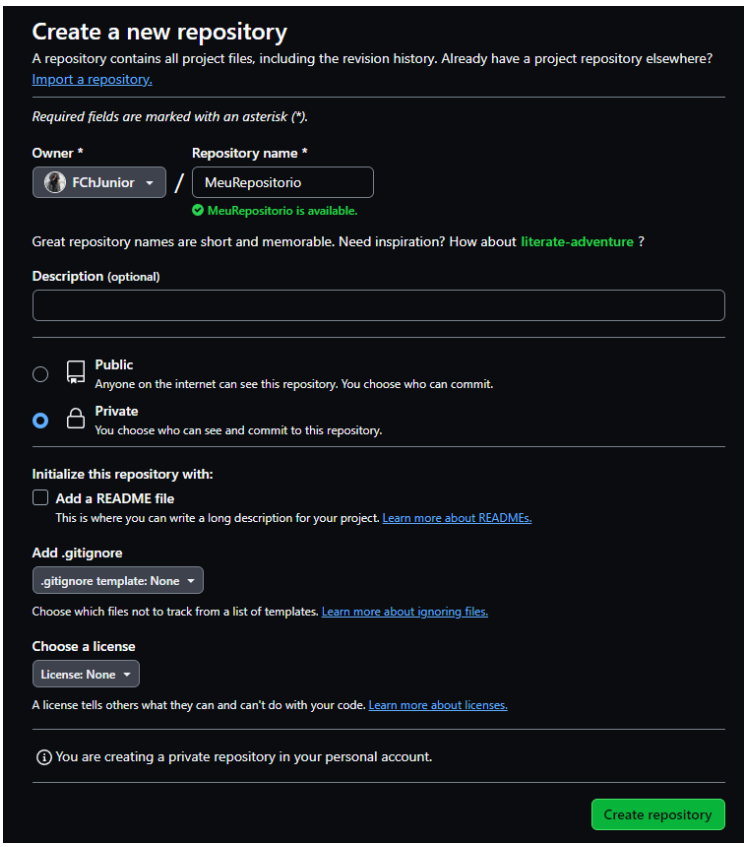
1. Vá no ícone do seu perfil e procure por **Your Repositories**:



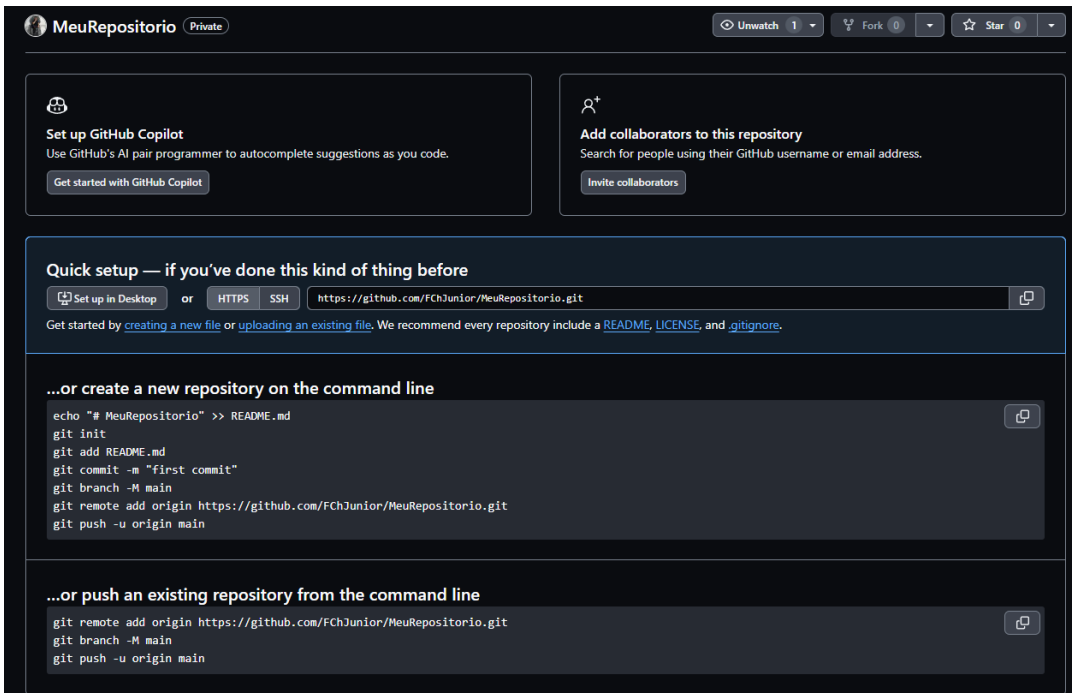
2. Clique no botão verde **NEW**:



3. Insira um nome para o seu repositório. Pode ser o mesmo nome que criamos do nosso local. Para agora, não precisa colocar descrição, deixe privado (podemos trocar a visibilidade depois), veremos mais tarde o que é o gitignore, README e as licenças. Depois só clicar em **Create Repository**:



4. Quando criamos um repositório novo, ele já vem com os comandos iniciais para podemos linkar ele com o nosso local (Essas informações so aparecem se não for criado nenhum arquivo antes, como o gitignore, readme ou licença):



5. Tudo certo, mas antes de linkar com o repo local, vamos aprender os comandos git.

4. Comandos de Controle Remoto

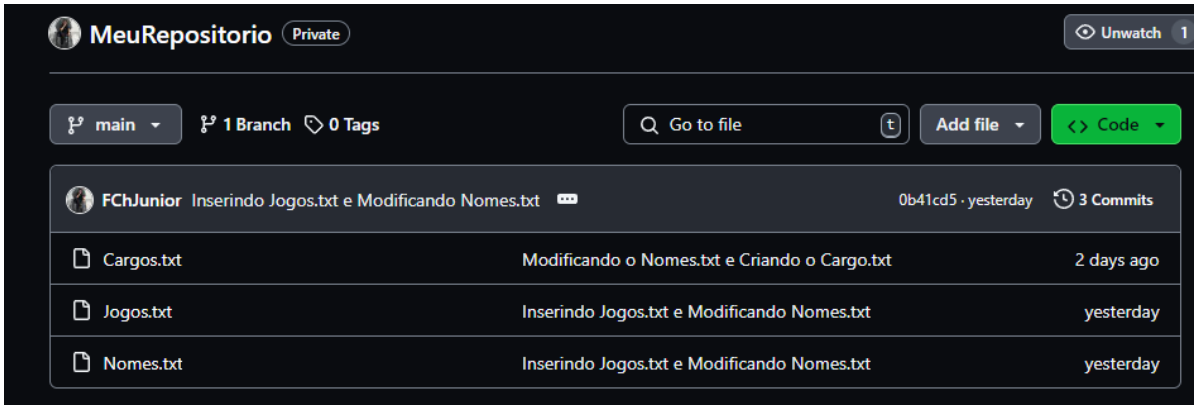
Para podemos trabalhar com o repositório local e o remoto, precisamos aprender so comando básicos para essa comunicação:

- ***git remote add <nome> <URL-do-repositório-remoto>:*** Adiciona um repositório remoto com o nome especificado e a URL.
- ***git remote rename <nome-antigo> <nome-novo>:*** Modifica o nome do repositório remoto.
- ***git clone <URL-do-repositório-remoto>:*** Clona um repositório remoto já existente (Útil para trabalhos em equipe com repositório público ou projeto Open Soure).
- ***git push <remoto> <branch>:*** Envia as alterações do repositório local para o remoto.
- ***git fetch <remoto>:*** Sincroniza os metadados do repositório remoto no local sem adição de merges.
- ***git pull <remoto> <branch>:*** Sincroniza os todos os arquivos do repositório remoto no local.

5. Praticando...

Agora iremos conectar o repositório local com o remoto. Quando criamos o repositório remoto, sem criar arquivos, ele nos da o passa a passo para poder fazer o link:

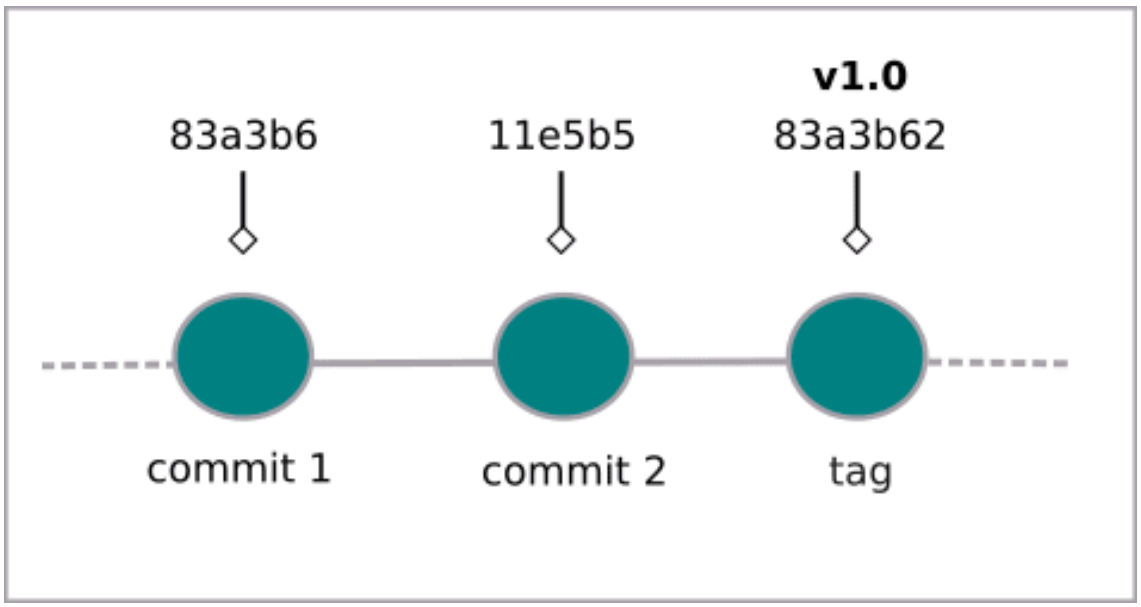
1. ***git remote add origin "URL do seu repositório";***
2. ***git branch -M main;***
3. ***git push -u origin main;***



Tudo certo, já temo nosso repositório remoto sincronizado com o local. Lembre-se que toda grande modificação no código (criação de uma nova feature) deve ser **commitado** e enviado para o remoto com o **git push**.

Releases (Tags)

Assim como as **branches** ajudam a **isolar** e desenvolver novas funcionalidades sem afetar o código principal, as **tags** no Git são usadas para **marcar** pontos **específicos** na **história** do repositório, como **versões** de lançamento de software. Imagine que você tem uma versão estável do seu projeto e deseja marcar esse ponto para referência futura. As **tags** são a solução ideal para isso.



ANEXO 11: GITHUB
LINK: https://dev.to/womakerscode/tutorial-git-etiquetas-3h1b

1. O que são Tags?

As **tags** no Git são usadas para marcar pontos específicos na história de um repositório. Elas são comumente utilizadas para marcar versões de lançamento de software (por exemplo, v1.0, v2.0). Existem dois tipos principais de **tags** no Git:

- **Tags Leves:** São simplesmente ponteiros para um commit específico, semelhantes aos branches, mas são imutáveis.
- **Tags Anotadas:** São objetos completos do Git que armazenam mais informações, como a data, o nome do autor e uma mensagem. Elas são recomendadas para versões de lançamento, pois fornecem metadados adicionais.

2. Comandos Git para Tags

- ***git tag <nome-da-tag>:*** Cria uma nova tag lincada ao último commit com a descrição desse commit.
- ***git tag:*** Lista as Tags do repositório;
- ***git tag -n:*** Lista as Tags do repositório com a descrição;
- ***git show <nome-da-tag>:*** Exibe os detalhes de uma tag.
- ***git tag -a <nome-da-tag> -m "Mensagem da tag":*** Cria Anotada.

- ***git tag <nome-da-tag> <commit-hash>:*** Cria uma tag para um commit especificado pelo seu hash;
- ***git tag -a <nome-da-tag> <commit-hash> -m "Mensagem da tag":*** Cria uma tag anotada para um commit especificado pelo seu hash;
- ***git tag -d <nome-da-tag>:*** Deleta a tag especificada;
- ***git push <remoto> <nome-da-tag>:*** Sobe a tag especificada para o repositório remoto;
- ***git push <remoto> --tags:*** Sobe todas as tags para o repositório remoto.

3. Praticando...

Imagine que você está desenvolvendo um jogo. Durante o desenvolvimento, você alcança marcos importantes, como uma versão jogável, um lançamento alfa, um lançamento beta, e finalmente a versão final do jogo. Marcar esses pontos com tags permite que você mantenha um histórico claro e facilmente acessível de cada versão significativa do jogo.

1. Criando uma Tag

Suponha que você acabou de completar a primeira versão jogável do seu jogo. Você pode criar uma tag anotada para marcar esse ponto:

- **git tag -a v1.0 -m "Primeira versão jogável do jogo"**

Esta tag (v1.0) agora aponta para o commit que representa a primeira versão jogável do seu jogo, e a mensagem de anotação ajuda a identificar o motivo dessa marcação.

2. Listando Tags

Para visualizar todas as tags no seu repositório, você pode usar o comando:

- **git tag**

Isso exibirá todas as tags criadas, permitindo que você veja rapidamente todas as versões importantes do seu jogo.

3. Visualizando Detalhes de uma Tag

Se você quiser ver mais detalhes sobre uma tag específica, como a mensagem de anotação e o commit associado, use o comando:

- **git show v1.0**

Isso exibirá informações detalhadas sobre a tag v1.0, incluindo a mensagem que você adicionou ao criar a tag.

4. Enviando Tags para o Repositório Remoto

Após criar uma tag localmente, você pode querer compartilhá-la com sua equipe ou manter um backup no repositório remoto. Para fazer isso, use:

- **git push origin v1.0**

Se você quiser empurrar todas as tags de uma vez, use:

- **git push origin –tags**

5. Utilizando Tags para Gerenciar Lançamentos

Digamos que você tenha lançado uma versão beta do seu jogo e está pronto para começar a trabalhar em novas funcionalidades ou correções de bugs. Você pode criar uma tag para essa versão beta:

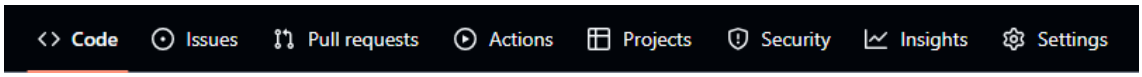
- **git tag -a v2.0-beta -m "Lançamento da versão beta"**

Isso permite que você sempre tenha um ponto de referência para a versão beta. Se precisar voltar para a versão beta para corrigir um bug específico que foi reportado, você pode facilmente verificar essa tag:

- **git checkout v2.0-beta**

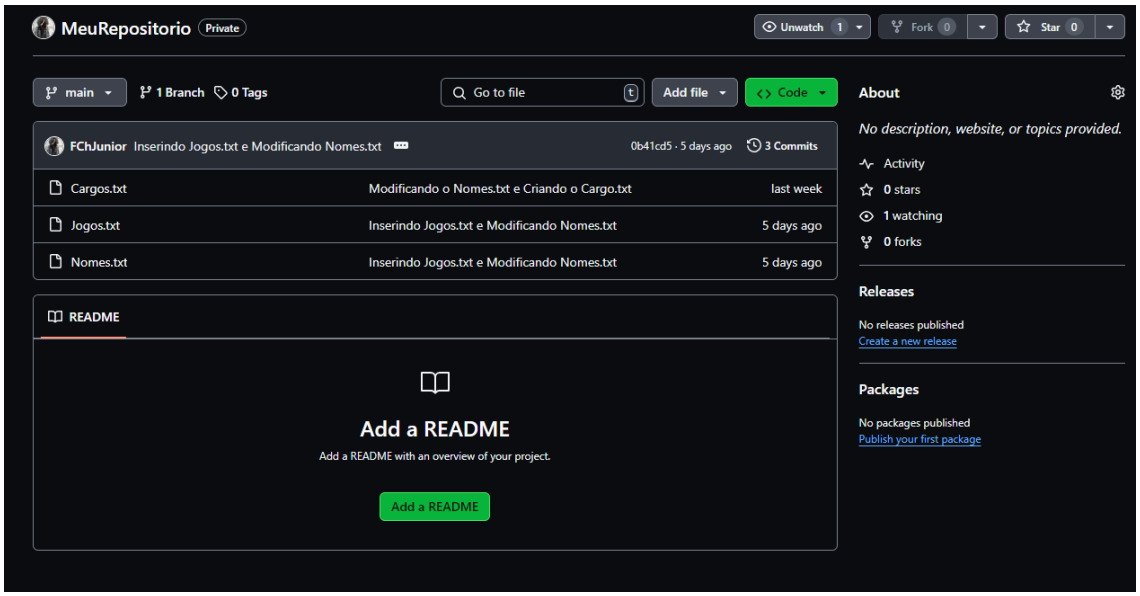
Abas de um repositório

Os repositórios no GitHub possuem várias abas que ajudam a gerenciar, colaborar e monitorar o desenvolvimento do projeto. Cada aba oferece diferentes funcionalidades e ferramentas. Vamos explorar cada uma delas em detalhe:



1. Code

A aba **Code** é a principal interface do repositório, onde o código-fonte do projeto é armazenado e gerenciado. Nessa seção, você pode visualizar os arquivos e diretórios, navegar pelas branches, ver commits, tags e muito mais. É a aba onde os desenvolvedores passam a maior parte do tempo, pois é aqui que todo o código e histórico do projeto são mantidos.



Funcionalidades:

- **Arquivos e Diretórios**: Exibe a estrutura completa do projeto, incluindo todos os arquivos e pastas. Você pode navegar pelos diretórios clicando neles e visualizar o conteúdo dos arquivos diretamente no navegador. A visualização pode ser em modo árvore, o que facilita a navegação por grandes projetos.
- **Commits**: Lista de commits recentes, cada um com uma mensagem de commit, autor, data e hora. Clicar em um commit específico permite visualizar as alterações detalhadas, incluindo diffs de arquivos modificados.
- **Branches**: Gerenciamento e visualização das diferentes branches do projeto. Você pode trocar entre branches, criar branches, e visualizar o histórico de commits de cada branch.

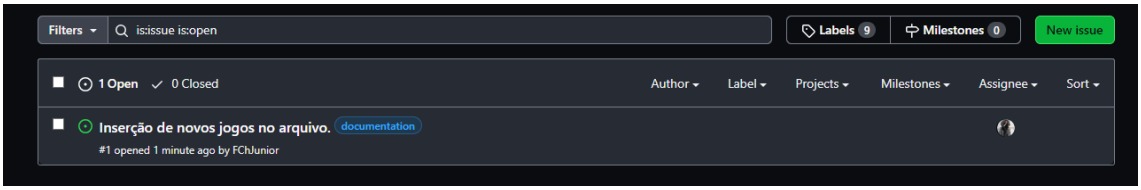
- **Tags**: Exibição e criação de tags para marcar versões específicas do projeto. Tags são usadas para marcar lançamentos ou outros pontos importantes na história do repositório.
- **Clone/Download**: Opções para clonar o repositório via HTTPS, SSH, ou baixar como arquivo ZIP. Clonar um repositório permite que você trabalhe com uma cópia local do projeto.
- **README**: Exibição do arquivo README.md na parte inferior, fornecendo uma visão geral do projeto. O README geralmente contém informações importantes como descrição do projeto, instruções de instalação, e exemplos de uso.

Como Utilizar:

- **Navegar pelo Código**: Clique nas pastas e arquivos para visualizar o conteúdo. Use a barra de navegação para se orientar na estrutura do projeto.
- **Ver Histórico de Commits**: Clique em "Commits" para ver todos os commits feitos no repositório. Cada commit mostra quais arquivos foram alterados e por quem.
- **Gerenciar Branches**: Use o menu **dropdown** para trocar entre branches ou criar. Branches permitem que você trabalhe em diferentes funcionalidades de forma isolada.
- **Criar Tags**: Utilize a opção de tags para marcar releases importantes. As tags facilitam o rastreamento de versões específicas do software.

2. Issues

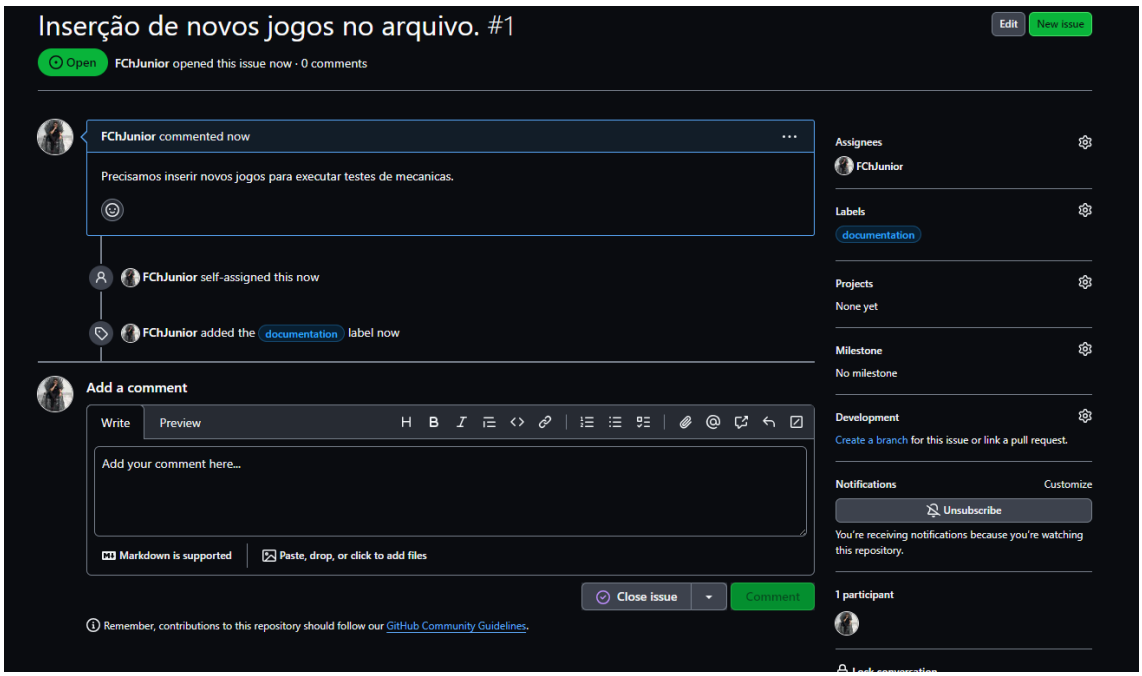
A aba **Issues** é usada para rastrear bugs, novas funcionalidades, melhorias e outras solicitações relacionadas ao projeto. É uma ferramenta crucial para a gestão de tarefas e colaboração, permitindo que os membros da equipe discutam problemas e propostas de melhorias.



Funcionalidades:

- **Criar Issue**: Botão para abrir uma nova issue, descrevendo um bug, solicitação de feature ou melhoria. Você pode adicionar rótulos, assignees, e projetos para organizar melhor as issues.

- **Listagem de Issues**: Lista todas as issues abertas e fechadas, com informações como título, rótulos, assignee e estado (aberta ou fechada). Você pode clicar em qualquer issue para ver detalhes completos e comentários.
- **Filtros e Pesquisas**: Ferramentas para filtrar issues por rótulos, assignees, projetos, milestones etc. Isso facilita encontrar issues específicas rapidamente.
- **Templates de Issue**: Modelos predefinidos para facilitar a criação de issues estruturadas. Os templates ajudam a garantir que todas as informações necessárias sejam fornecidas.
- **Discussões e Comentários**: Áreas para discutir a issue e adicionar comentários, permitindo colaboração e resolução. Os participantes podem usar markdown para formatar os comentários e anexar imagens ou links.

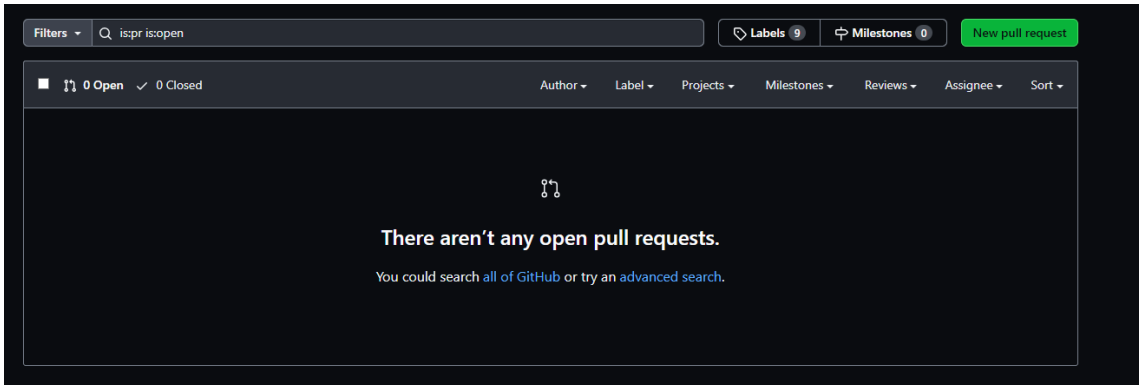


Como Utilizar:

- **Abrir uma Nova Issue**: Clique em "New issue" e preencha os detalhes necessários. Descreva o problema ou solicitação de forma clara e detalhada.
- **Filtrar e Pesquisar**: Utilize os filtros para encontrar issues específicas rapidamente. Você pode combinar vários filtros para refinar sua busca.
- **Comentar e Colaborar**: Adicione comentários em issues para discutir e solucionar problemas. Use @menções para chamar a atenção de outros colaboradores.
- **Fechar Issues**: Marque issues como resolvidas quando o problema for solucionado ou a feature implementada. Isso ajuda a manter o repositório organizado e atualizado.

3. Pull request

A aba **Pull Requests** é usada para gerenciar propostas de alterações no código do repositório. Os desenvolvedores podem enviar pull requests (PRs) para revisar e discutir mudanças antes de integrá-las ao código principal. É uma parte fundamental do fluxo de trabalho colaborativo.



Funcionalidades:

- **Criar Pull Request**: Botão para abrir um novo PR, selecionando as branches de origem e destino. Você pode adicionar uma descrição detalhada das mudanças propostas e rótulos para categorizar o PR.
- **Listagem de Pull Requests**: Lista de PRs abertos, fechados e mesclados, com informações como título, autor, estado, rótulos e comentários. Cada PR mostra o diff das mudanças propostas.
- **Discussões e Revisões**: Áreas para discutir o PR, solicitar mudanças, e aprovar as alterações. Revisores podem deixar comentários específicos em linhas de código.
- **Verificação Automática**: Integração com ferramentas de CI/CD para executar testes automatizados e verificações de código. Isso ajuda a garantir que as mudanças não introduzam bugs.
- **Merge e Fechamento**: Opções para mesclar o PR no branch de destino ou fechar o PR sem mesclar. Mesclar um PR aplica as mudanças propostas ao branch de destino.

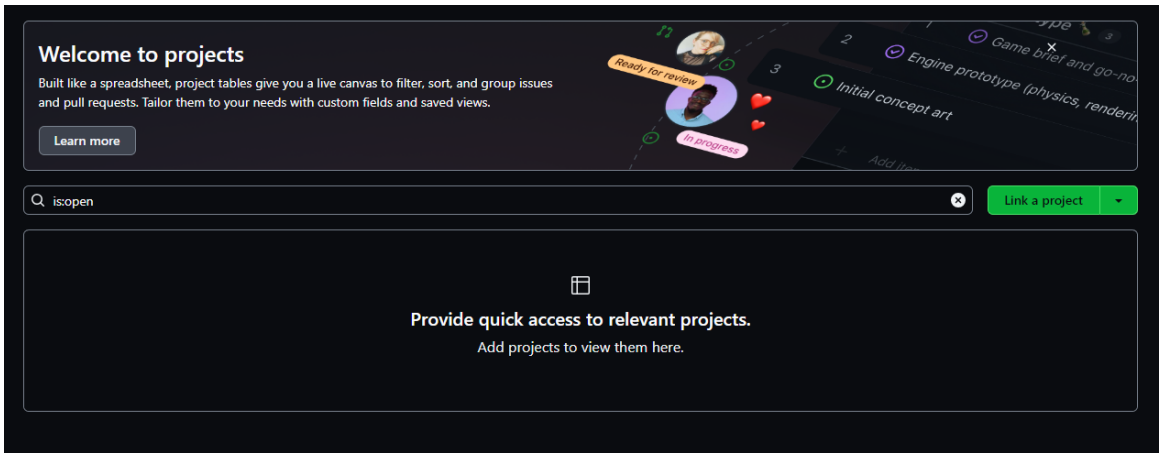
Como Utilizar:

- **Abrir um Novo PR**: Clique em "New pull request" e selecione as branches de origem e destino. Descreva as mudanças propostas e adicione rótulos e assignees.
- **Revisar PRs**: Clique em um PR para ver os detalhes das mudanças propostas. Use a área de comentários para discutir e revisar o código.

- **Aprovar e Mesclar**: Após revisar as mudanças e garantir que todos os testes passaram, você pode aprovar e mesclar o PR. Use o botão "Merge pull request" para integrar as mudanças.
- **Fechar PRs**: Se um PR não for mais necessário ou relevante, você pode fechá-lo sem mesclar as mudanças. Isso mantém o repositório limpo e organizado.

4. Projects

A aba **Projects** oferece uma maneira de organizar e gerenciar tarefas e fluxos de trabalho usando quadros Kanban. É uma ferramenta poderosa para planejar, rastrear e gerenciar o progresso do desenvolvimento.



Funcionalidades:

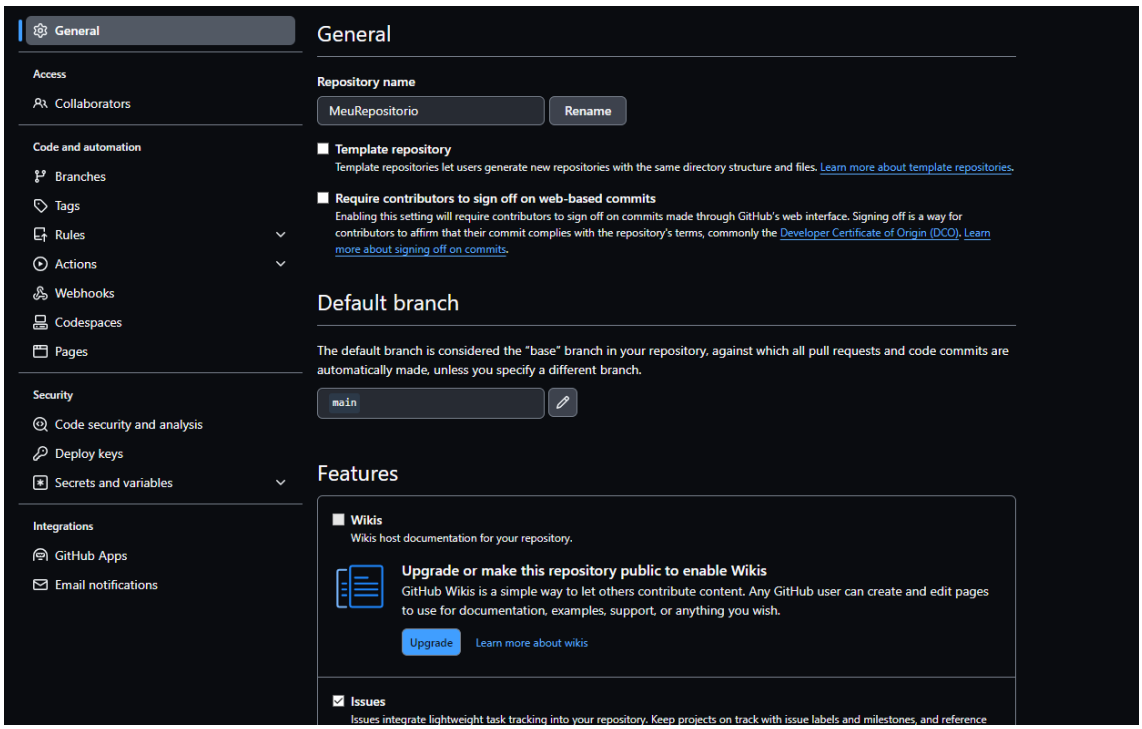
- **Quadros de Projetos**: Criação de quadros Kanban para visualizar e gerenciar tarefas. Você pode criar colunas personalizadas para representar diferentes estados de progresso (como "To Do", "In Progress", "Done").
- **Cartões de Tarefas**: Adição de cartões de tarefas para representar issues, pull requests, ou tarefas gerais. Os cartões podem ser movidos entre colunas para refletir o progresso.
- **Automação**: Configuração de regras de automação para mover cartões automaticamente entre colunas com base em eventos específicos (como fechamento de issues).
- **Filtragem e Classificação**: Ferramentas para filtrar e classificar cartões com base em critérios como assignee, rótulos e estado. Isso facilita a priorização e o acompanhamento de tarefas.
- **Visualização de Progresso**: Visualização clara do estado atual do projeto, ajudando a identificar gargalos e planejar o trabalho futuro.

Como Utilizar:

- **Criar um Projeto**: Clique em "New Project" e defina colunas que representem os diferentes estágios do seu fluxo de trabalho. Adicione cartões para issues e tarefas relevantes.
- **Mover Cartões**: Arraste e solte cartões entre colunas para atualizar o status das tarefas. Use a automação para simplificar o gerenciamento.
- **Filtrar e Classificar Cartões**: Utilize as ferramentas de filtragem e classificação para encontrar rapidamente tarefas específicas e priorizar o trabalho.
- **Monitorar Progresso**: Use a visualização do quadro Kanban para monitorar o progresso do projeto e identificar áreas que precisam de atenção.

5. Settigns

A aba **Settings** é onde você pode configurar todas as opções e preferências do repositório. Isso inclui configurações gerais, permissões, integrações, e muito mais. É a central de controle do repositório.



Funcionalidades:

- **General Settings**: Configurações básicas como renomear o repositório, alterar a descrição, e definir o site do projeto. Você também pode arquivar ou excluir o repositório.

- **Access Control**: Gerenciamento de permissões e acesso. Adicione ou remova colaboradores, defina níveis de acesso (leitura, escrita, admin), e configure equipes.
- **Branches**: Configurações de branches, como proteção de branches e políticas de merge. Defina regras para proteção de branches principais para evitar commits diretos.
- **Webhooks**: Configuração de webhooks para integrar o repositório com serviços externos. Webhooks podem disparar eventos em resposta a ações no repositório.
- **Integrations & Services**: Conectar o repositório com serviços e aplicações externas como CI/CD, notificações, e outras ferramentas. Configure integrações diretamente do GitHub Marketplace.
- **Security & Analysis**: Configurações de segurança, incluindo análise de código, monitoramento de dependências, e políticas de segurança. Ative ou desative funcionalidades de segurança conforme necessário.

Como Utilizar:

- **Configurar Opções Gerais**: Acesse General Settings para ajustar as configurações básicas do repositório, incluindo nome, descrição, e site do projeto.
- **Gerenciar Permissões**: Use Access Control para adicionar colaboradores e definir suas permissões. Configure equipes para gerenciar acesso de forma eficiente.
- **Proteger Branches**: Vá para Branches para definir regras de proteção, como exigir revisões de pull requests e proibir commits diretos em branches principais.
- **Configurar Webhooks**: Adicione webhooks para integrar com serviços externos. Defina quais eventos devem disparar os webhooks e para onde enviar os dados.
- **Integrar Ferramentas Externas**: Acesse Integrations & Services para conectar o repositório com ferramentas de CI/CD, notificações, e outras aplicações. Configure e gerencie integrações diretamente do GitHub Marketplace.
- **Ajustar Configurações de Segurança**: Use Security & Analysis para ativar ou desativar funcionalidades de segurança, como análise de código e monitoramento de dependências. Configure políticas de segurança para proteger o repositório.

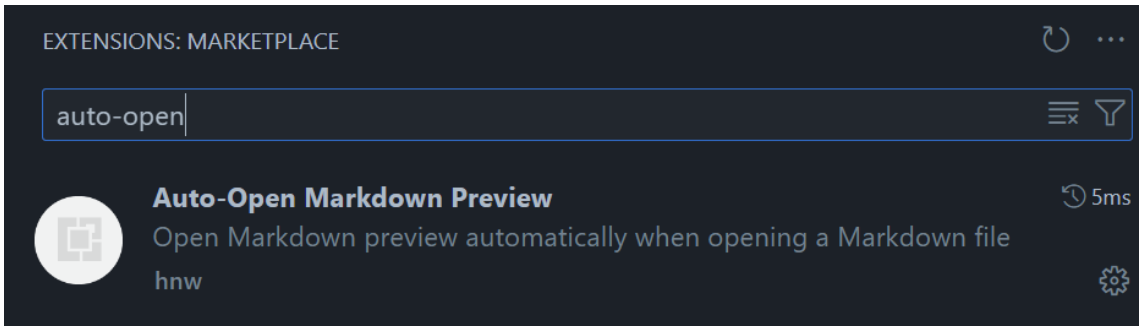
Markdown

1. O que é o Markdown?

Markdown é uma linguagem de marcação leve criada por John Gruber em 2004. É amplamente utilizada para formatar texto em muitos contextos, como arquivos README, fóruns, blogs e documentos. A principal vantagem do Markdown é que ele é simples de escrever e ler, convertendo-se facilmente em HTML ou outros formatos.

ANEXO 11: MARKDOWN
LINK: https://vaikida.online/o-que-e-markdown-e-como-ele-funciona/

Antes de começar a criar nossos arquivos **README**, iremos instalar uma extensão no VS Code para ajudar na criação e visualização dos nosso arquivos **.md**. Va na extensões do **VS Code** e procure por **Auto-Open Markdown Previw** e instale:

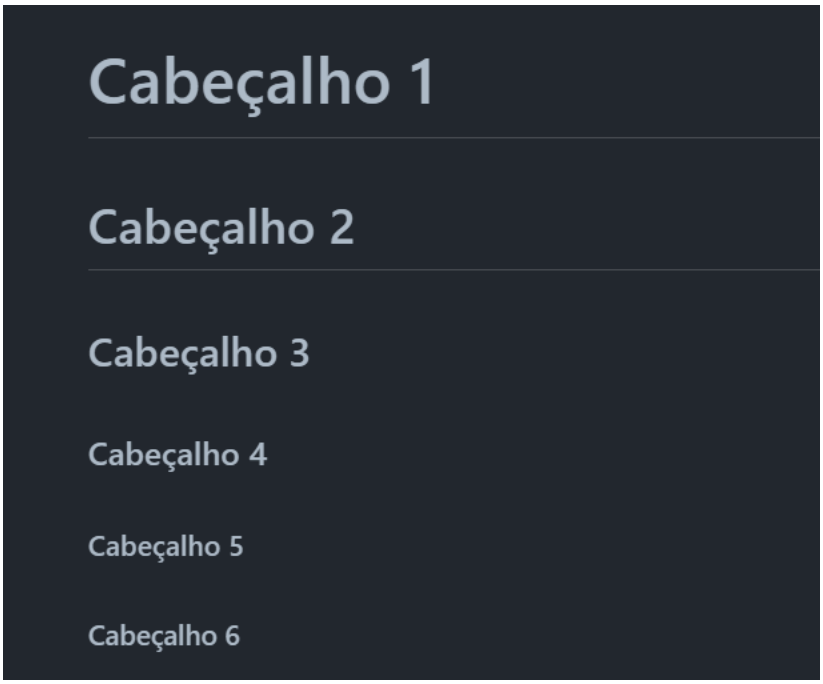


Tudo certo! Agora já podemos trabalhar com arquivos .md com a visualização instantânea sem precisar criar commits.

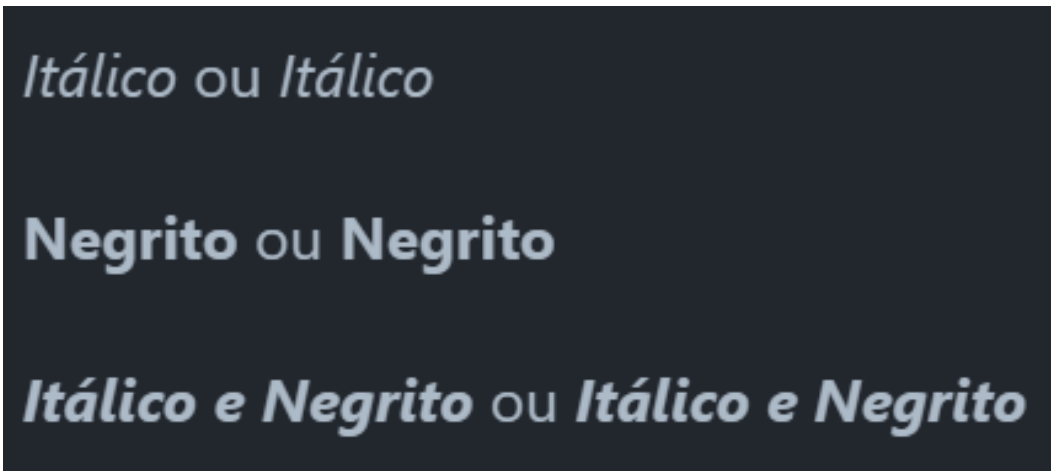
2. Comandos Markdown

O Markdown é uma linguagem de marcação semelhante ao HTML, e ele usa de caracteres para criar as marcações no texto. Veremos agora os principais caracteres de marcação para se utilizar o markdown.

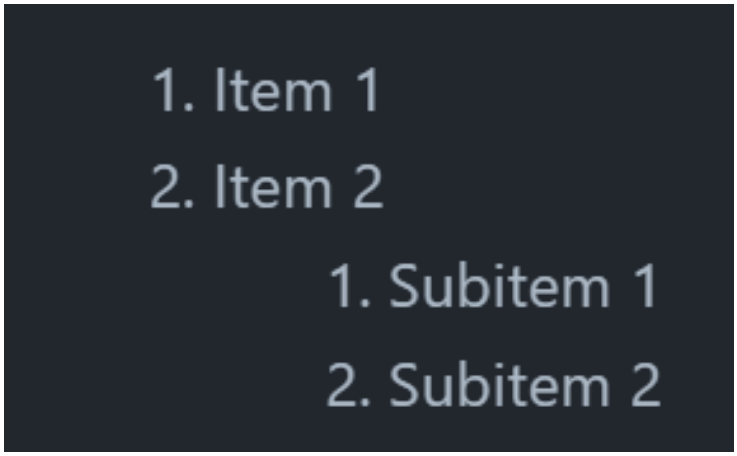
1. **Cabeçalhos:** Para criar cabeçalhos, use o símbolo # seguido de um espaço. O número de # indica o nível do cabeçalho (de 1 a 6).
   - **# Cabeçalho 1**
   - **## Cabeçalho 2**
   - **### Cabeçalho 3**
   - **#### Cabeçalho 4**
   - **##### Cabeçalho 5**
   - **###### Cabeçalho 6**



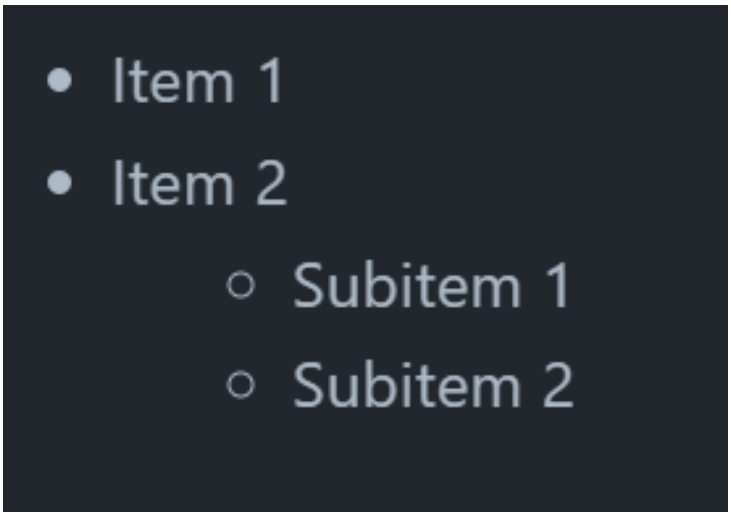
2. **Ênfase:** Para adicionar ênfase ao texto, use asteriscos ou underscores.
   - ***Itálico* ou _Itálico_**
   - ****Negrito** ou __Negrito__**
   - ****Itálico e Negrito** ou ___Itálico e Negrito___**



3. **Listas:**
   a. **Listas Ordenadas:** Use números seguidos de um ponto.
      - **1. Item 1**
      - **2. Item 2**
      - **1. Subitem 1**
      - **2. Subitem 2**

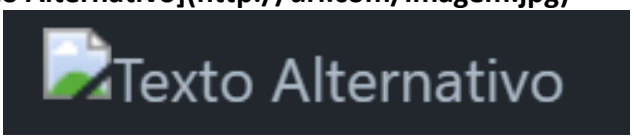


   b. **Lista não Ordenadas:** Listas Não Ordenadas: Use hífen -, asterisco *, ou sinal de mais +:
      - **- Item 1**
      - **- Item 2**
      - **- Subitem 1**
      - **- Subitem 2**

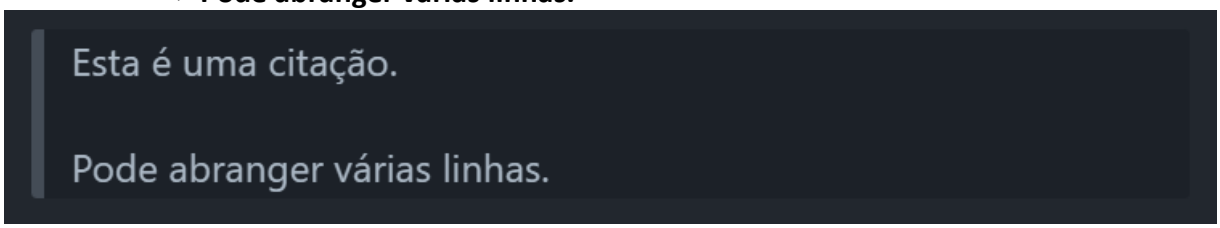


4. **Links e Imagens:** Para adicionar links e imagens, use colchetes [] e parênteses ().
   a. **Links:**
      - **[Texto do Link](http://url.com)**

b. **Imagens:** Adicione um ponto de exclamação (!) antes dos colchetes.
   - **![Texto Alternativo](http://url.com/imagem.jpg)**



5. **Citações:**
   a. **Citações Simples:** Para criar citações, use o símbolo >.
      - **> Esta é uma citação.**
      - **>**
      - **> Pode abranger várias linhas.**



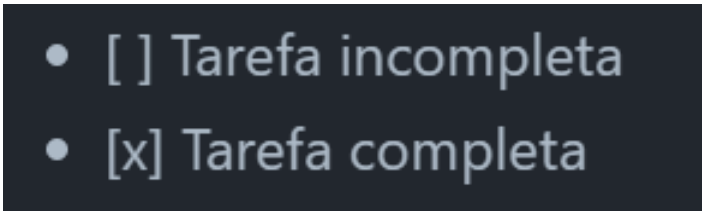
   b. **Citações Aninhadas:** Para criar blocos de citação aninhados, use > múltiplas vezes.
      - **> Primeira citação**
      - **>> Segunda citação**
      - **>>> Terceira citação**

6. **Tabelas:** Para criar tabelas, use hifens (-) para criar a linha do cabeçalho e pipes (|) para separar colunas.

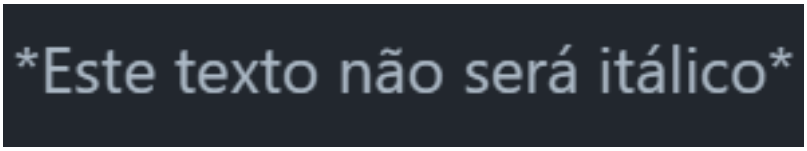
   - **| Coluna 1 | Coluna 2 | Coluna 3 |**
   - **|----------|----------|----------|**
   - **| Linha 1 | Item 1 | Item 2 |**
   - **| Linha 2 | Item 3 | Item 4 |**

Coluna 1	Coluna 2	Coluna 3
Linha 1	Item 1	Item 2
Linha 2	Item 3	Item 4

7. **Lista de Tarefas:** Para criar listas de tarefas, use colchetes [] e adicione um x dentro dos colchetes para marcar como concluído.
    - **- [ ] Tarefa incompleta**
    - **- [x] Tarefa completa**

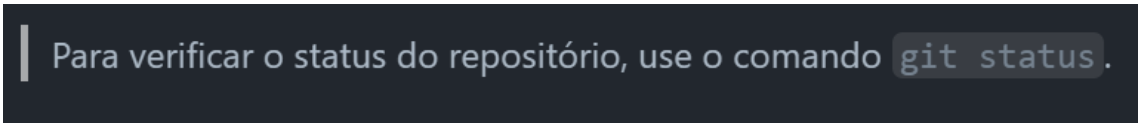


8. **Escapando Caracteres:** Para usar caracteres reservados do Markdown como texto normal, escape-os com uma barra invertida \.
    - ***Este texto não será itálico***



9. **Citações de Código:** No Markdown, você pode formatar e destacar código de várias maneiras, utilizando citações e marcação de linguagem. Isso facilita a leitura e compreensão do código em documentos e documentos técnicos.

    a. **Código em Linha:** Para incluir código em linha, use crases simples `. Isso é útil para trechos pequenos de código ou comandos.
        - **`comando` ou `trecho de código`**
        - **EX.: Para verificar o status do repositório, use o comando *`git status`*.**



b. Blocos de Código: Para criar blocos de código, que são mais longos e geralmente formatados, use três crases ```. Você pode também incluir a marcação de linguagem para destacar a sintaxe do código, o que melhora a legibilidade.

- **```linguagem**
  código aqui
  ```
- **Ex.:**

```C#
using System;
namespace Project
{
** public class Program**
** {**
**   public static void Main()**
**   {**
**   Console.WriteLine("Hello World!");**
**   }**
** }**
}
```

```C#
using System;

namespace Project
{
    public class Program
    {
        public static void Main()
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

10. Comentário: Para adicionar comentários que não aparecem na renderização final, use a sintaxe de comentário HTML.

- **<!-- Este é um comentário -->**

E aqui encerramos nossos conhecimentos em Git e GitHub. Pratique bastante utilizando esse material para fixar o conhecimento. BONS ESTUDOS!