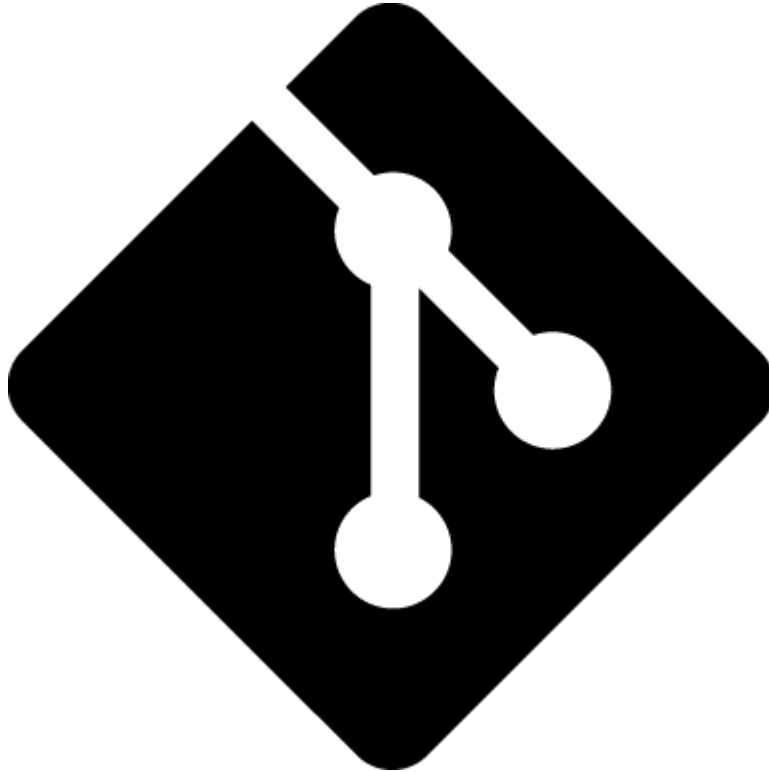


# Manual Básico de Git



Resumo de Informações para consulta

Criador: Gabriel Fanto Stundner

Versão do Manual: 2.0

Link do Repositório de Informações: <https://github.com/F4NT0/ages-online-git>

## GLOSSÁRIO

1. O que é GIT? **Página 3**
2. O que é um Sistema de Controle de Versões? **Página 4**
3. Nomenclaturas no GIT! **Página 5**
4. Comandos do GIT **Página 6**
  1. Criando um Repositório **Página 6**
  2. Criando uma nova Branch de Trabalho **Página 11**
  3. Modificando arquivos na Branch **Página 13**
  4. Adicionando Arquivos para Salvar **Página 15**
  5. Salvando todas as Modificações **Página 17**
  6. Enviando as Modificações Remotamente **Página 19**
  7. Verificando Remotamente as Modificações **Página 21**
  8. Editando arquivos Remotamente **Página 23**
  9. Baixando modificações feitas para o computador **Página 24**
  10. Como voltar um Commit feito **Página 25**
  11. Unindo duas Branchs fazendo Merge **Página 30**
  12. Utilizando o Git Remote **Página 32**
5. Interagindo com o Github **Página 39**
6. Entendendo Estrutura de Texto de Repositórios Remotos **Página 44**
7. Dicas de Organização de Equipe **Página 46**
8. Resumo de Comandos GIT **Página 47**

## O que é GIT?

O significado das Três letras GIT não tem uma resposta única, o criador Linus Torvalds já apresentou diferentes tipos de explicações do que significa, podendo ser desde **G**lobal **I**nformation **T**racker quando se está de bom humor ou **G**oddamn **I**diotic **T**ruckload of Sh\*t em caso de mau humor.

O GIT é um *Sistema de Controle de Versões Distribuidos*, ou seja, é um sistema de gerenciamento de versões de um Projeto em Código.

É o Principal Sistema de Controle utilizado pelas empresas e pelo mundo todo em Desenvolvimento de Software, Lançado inicialmente em 2005.

O GIT possui vários comandos para auxiliar o usuário a verificar e interagir com as versões do seu código, onde o usuário tem toda a liberdade de voltar atrás caso tenha acontecido algum problema em seu código.

Cada Diretório(ou Pasta) de um Projeto gerenciado no GIT se chama **Repositório**.

As principais plataformas Remotas para guardar os seus Projetos Gerenciado pelo GIT são o **GITHUB**, **GITLAB** e **BITBUCKET**.



## O que é um Sistema de Controle de Versões?

Quando desenvolvemos um projeto em código estamos trabalhando com um arquivo texto, onde ele é modificado várias vezes por um Desenvolvedor, mas, como projetos em código pode acontecer de que duas ou mais pessoas estejam mexendo no mesmo arquivo ao mesmo tempo, pode causar problemas ou desentendimentos.

Um Sistema de Controle de Versões ajuda com que quando se está mexendo em um arquivo texto você possa salvar o arquivo no estado atual dele e se houver necessidade, possa voltar no estado anterior do arquivo sem precisar apagar o que já foi feito.

Além de salvar e poder voltar para a versão anterior, um Sistema de Controle de Versões ajuda também caso mais de uma pessoa estar mexendo na mesma linha do arquivo que você, onde se pode resolver problemas de conflito das versões do mesmo arquivo.

No GIT, todas as modificações feitas em um arquivo são gravados e podem ser modificados por várias pessoas separadamente e depois no final conectar todas as versões em uma só, resolvendo qualquer conflito entre as versões para se ter uma única versão final do Projeto.

## Nomenclaturas no GIT!

- **Repositório ou Repository:** Um repositório é a Pasta do Projeto que possui um diretório oculto chamado `.git`, esse diretório oculto irá fazer com que todo o diretório possa ser gerenciado com comandos GIT.
- **Branch:** Uma Branch é uma “Ramificação” do Projeto inicial, onde cada desenvolvedor pode mexer nos arquivos em seu computador sem interferir nas modificações dos outros Desenvolvedores do Projeto, a Branch principal para onde vai o Projeto inteiro final se chama *Master* mas nos sistemas mais atuais se chama *Main* devido que a palavra Master não é mais aconselhado a ser usada.
- **Commit:** Um commit é o nome dado para cada salvamento que se faz do código, toda vez que quiser salvar o que foi feito até o momento, você deve fazer um Commit das suas modificações, onde se deve colocar uma mensagem do que foi feito antes de Salvar.
- **Log:** Log é uma lista de todos os Commits feitos na sua Branch Atual, onde cada Commit possui um ID único, a Branch que o Commit foi feito e o nome do Autor do Commit, a Data que ele foi feito e a Mensagem do Commit.
- **HEAD:** Quando você ler escrito HEAD em um Commit significa que esse Commit é a versão atual do seu Código se ele não tiver sido salvo novamente, ou seja, é o ultimo salvamento feito até agora do seu Projeto na sua Branch
- **Status:** O Status é um comando GIT que mostra quais arquivos foram modificados ou criados antes de poder salvar, se nenhum arquivo foi modificado ou criado ele avisa que não tem nada para ser feito Commit, portanto nada foi modificado no Programa.
- **Merge:** quando pegamos modificações de uma Branch e queremos adicionar em outra Branch, chamamos esse ato de Merge, onde estamos conectando todas as modificações de duas versões diferentes em uma só, onde podem ocorrer conflitos

# Comandos do GIT

Agora vou apresentar como usar o GIT em seus Projetos, onde irei apresentar os comandos em sua Ordem de Desenvolvimento, além de explicar cada comando e opções que podem ser usadas.

A primeira coisa que irei apresentar é como criar um Repositório vazio para o seu Código, onde existem duas formas diferentes de se fazer isso:

## Criando um Repositório

### Primeira Forma de Criação de um Repositório

1. A primeira coisa a ser feita é criar um Diretório(pasta) em seu computador, onde dependendo do computador se cria de uma forma diferente, eu irei mostrar como criar nos sistemas Unix e no Prompt do Windows

#### LINUX

- Abra um Terminal com as teclas **Ctrl + Alt + T**
- Navegue para um Diretório onde deseja criar o Projeto usando o comando **cd**
- Crie um novo Diretório com o nome projeto-git como o comando **mkdir projeto-git**
- Acesse o Diretório projeto-git usando o comando **cd projeto-git**

#### MACOSX

- Procure pelo aplicativo terminal
- após abrir o terminal, utilize o comando **cd** para ir até o Diretório que deseja criar o novo diretório
- Crie um novo Diretório com o nome projeto-git usando o comando **mkdir projeto-git**
- Acesse o Diretório projeto-git usando o comando **cd projeto-git**

## WINDOWS

- Abra o Programa do GIT instalado, onde ele possui um Terminal Próprio que se pode usar comandos Linux para auxiliar o Trabalho pelo Windows
- Vá até o Diretório Desejado usando o comando **cd**
- crie um novo Diretório vazio com o nome projeto-git com o comando **mkdir projeto-git**
- Acesse o Diretório projeto-git usando o comando **cd projeto-git**

2. Agora que possuímos um Diretório vazio para trabalharmos com o Código que iremos desenvolver, irei mostrar o primeiro comando GIT usado nessa forma para iniciarmos o git nesse Projeto:

# git init

O comando `git init` serve para iniciarmos um Repositório GIT em um Diretório novo, onde com esse comando ele vai criar um Diretório oculto chamado `.git`.

Esse Diretório `.git` vai ser o sistema de gerenciamento do projeto, onde todos os comandos do git poderão ser feitas pelo terminal ou por algum programa que auxilie o gerenciamento de forma gráfica.

Abaixo uma imagem dos passos dados acima pelo terminal Linux:

```
M4TRIX:Desktop > mkdir teste
M4TRIX:Desktop > cd teste
M4TRIX:teste > git init
Initialized empty Git repository in /home/f4nt0/Desktop/teste/.git/
M4TRIX:teste (master #) > ls -la
total 12
drwxr-xr-x 3 f4nt0 f4nt0 4096 jul 24 14:16 .
drwxr-xr-x 4 f4nt0 f4nt0 4096 jul 24 14:16 ..
drwxr-xr-x 7 f4nt0 f4nt0 4096 jul 24 14:16 .git
M4TRIX:teste (master #) > cd .git
M4TRIX:.git (GIT_DIR!) > ls
branches  config  description  HEAD  hooks  info  objects  refs
M4TRIX:.git (GIT_DIR!) > cd ..
M4TRIX:teste (master #) > █
```

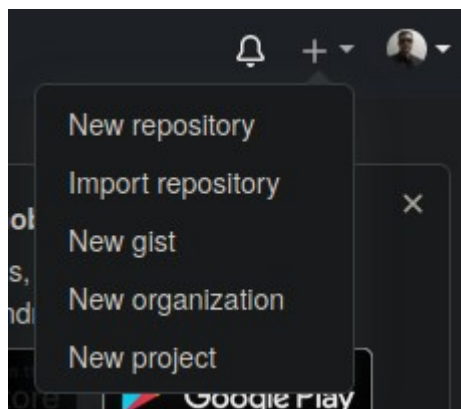
Para facilitar o entendimento e uso, o comando **cd ..** serve para voltarmos para o diretório anterior.

3. Agora você possui um repositório local em seu computador, onde ele é gerenciado pelo GIT, mas você não tem como enviar seus dados Remotamente somente assim, onde você deve pular para a parte do comando **REMOTE** nesse relatório, onde eu irei explicar isso mais tarde.

## Segunda Forma de Criação de um Repositório

Nessa segunda forma de se criar um Repositório, nós iremos criar ele de forma Remota ao nosso computador, onde iremos construir ele fora e passarmos ele localmente para o nosso computador, onde iremos seguir os seguinte passos:

1. Acesse a sua conta no Github que foi criada seguindo o tutorial de criação de conta no github.
2. Quando acessado a sua conta, na ponta direita tem um + onde existem opções de coisas para serem criadas, como abaixo:



3. Selecione a opção de **New repository** para criarmos um novo Repositório desejado
4. A próxima tela pede as informações sobre esse Repositório, onde ele vai verificar se o nome do Repositório já foi pego, a Descrição do Repositório e se deseja um README, que é um arquivo que mostra informações sobre o Repositório para outros Usuários.



# Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner \*



F4NT0 ▾

Repository name \*

/ projeto-git



Great repository names are short and memorable. Need inspiration? How about **jubilant-dollop**?

Description (optional)

Exemplo de Repositório Criado



**Public**

Anyone on the internet can see this repository. You choose who can commit.



**Private**

You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

☒ **Initialize this repository with a README**

This will let you immediately clone the repository to your computer.

Add .gitignore: None ▾

Add a license: None ▾

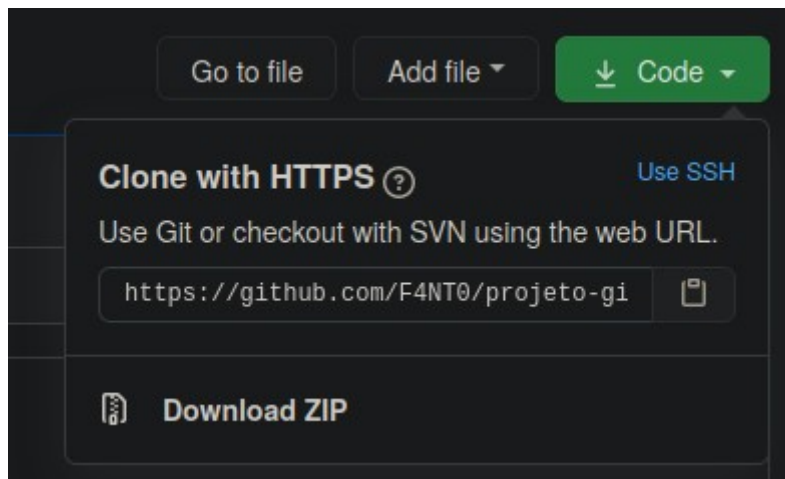



Create repository

5. Quando clicado no botão **Create repository** ele vai criar um Repositório vazio para podermos trabalhar, como esse abaixo:

The screenshot shows the GitHub repository page for 'F4NT0 / projeto-git'. The repository is public and has been initialized with a README. The page displays the repository name, owner, and a list of files including 'README.md'. The README content is visible, showing the repository name and a description. The right sidebar contains sections for 'About', 'Releases', and 'Packages', all indicating no content has been published yet.

6. Agora iremos trabalhar com um novo comando, onde iremos chamar esse Repositório que se encontra nos Servidores do Github para o nosso computador, onde iremos clicar no botão verde escrito **Code** que irá nos entregar a URL desse Repositório:



Se você clicar no botão com o simbolo  ele vai copiar a URL HTTPS da localização do seu Repositório

7. Então agora irei apresentar o novo comando GIT para ser feito em seu Terminal com GIT:

# git clone url

Nesse comando acima, a palavra url que se encontra em vermelho é onde iremos colar o URL copiado no passo anterior em seu Terminal no Diretório desejado.

Para exemplificar, vamos se dizer que eu estou no Terminal em um Diretório de nome *GIT/* onde eu coloco todos os meus Repositórios vindos Remotamente, então eu irei usar o comando acima para poder baixar o Repositório para o meu computador e guardar o Repositório dentro do Diretório *GIT/*.

```
M4TRIX:GIT > git clone https://github.com/F4NT0/projeto-git.git
Cloning into 'projeto-git'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
M4TRIX:GIT > ls
projeto-git
M4TRIX:GIT > cd projeto-git/
M4TRIX:projeto-git (master) > █
```

No exemplo acima, ele fez o clone do Repositório, pegando todas as informações e arquivos da Branch Principal, chamada de *master* e depois verifiquei quais Repositórios tinha dentro do Diretório GIT com o comando **ls** e depois acessei o nosso Repositório com o comando **cd projeto-git/**

8. Agora temos o Repositório Localmente dentro do nosso computador, para podermos mexer nele e fazer o código que quisermos.

## **Criando uma nova Branch de Trabalho**

Agora que temos o nosso Repositório criado, podemos trabalhar com o nosso Código, onde nesse momento existe uma organização de trabalho muito utilizada na indústria que é a criação de Branchs de trabalho.

A Branch principal que será entregue ao cliente final é a Branch *master* que é a primeira Branch criada pelo Sistema GIT, Não é recomendado trabalhar direto nela devido que o seu Projeto pode ter várias pessoas trabalhando ao mesmo tempo e isso iria interferir trágicamente no desenvolvimento do Projeto, onde iria ter mais de uma pessoa alterando a mesma parte do arquivo, sendo um transtorno de Problema de compatibilidade de arquivos.

Para isso podemos criar várias Branchs em um único Projeto, onde cada um pode trabalhar em uma Branch sem interferir no Trabalho do outro, onde a Branch copia todos os arquivos da Branch Original onde o Desenvolvedor pode mexer nela sem se preocupar em modificar o Projeto Original.

Exemplo: Temos agora o nosso Repositorio projeto-git na sua Branch Original *master* somente com um arquivo texto chamado README.md que é um arquivo texto em uma Linguagem de Texto chamado Markdown. Queremos então modificar esse Texto sem mexer no Arquivo Original, para isso iremos usar um comando novo do GIT para podermos criar uma Branch vinda da *master*:

# git checkout -b nome-branch

No comando *checkout* ele pode ser usado de diferentes formas mas essa forma é a mais aconselhada de se usar, onde nesse comando *checkout* tem a opção *b* que significa que iremos criar uma Branch nova com o nome *nome-branch*, que está em vermelho porque significa que você pode colocar o nome que quiser na Branch iniciada.

O comando ao total significa que você está criando uma Branch nova e indo direto para ela, se quiser, você pode criar uma Branch sem precisar acessar ela direto, como abaixo:

## OPÇÃO SECUNDÁRIA

# git branch nome-branch

Esse comando vai criar uma Branch, mas não vai acessar ela direto, onde você pode usar esse comando se quiser criar várias Branchs vindas de uma única Branch Original.

# git checkout nome-branch

Esse comando faz com que você possa acessar uma Branch qualquer que você tenha dentro de seu computador, sempre que quiser. Sempre que quiser trocar de Branch use esse comando

Abaixo um exemplo desses comando no Terminal Linux:

```
M4TRIX:projeto-git (master) > git checkout -b branch_teste
Switched to a new branch 'branch_teste'
M4TRIX:projeto-git (branch_teste) > git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
M4TRIX:projeto-git (master) > git branch nova_branch
M4TRIX:projeto-git (master) > git checkout nova_branch
Switched to branch 'nova_branch'
M4TRIX:projeto-git (nova_branch) > git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
M4TRIX:projeto-git (master) > █
```

## Modificando arquivos na Branch

Agora que temos o nosso Repositório e a nossa Branch de Trabalho, podemos finalmente trabalhar com nosso arquivos, como irei mostrar com o Arquivo README.md que existe dentro do nosso Projeto de Teste.

Fiz uma modificação qualquer no arquivo, onde apaguei o que tinha e coloquei novas informações quaisquer, mas primeiro devo verificar se estou na Branch que criei onde, se não estiver escrito qual a branch que estou eu uso o seguinte comando no Terminal para saber quais Branchs eu tenho e qual delas estou atualmente:

**git branch**

Se você usar esse comando, vai aparecer um Asterisco (\*) na Branch que você está atualmente, assim como vai apresentar as Branchs que você possui localmente.

Se você quiser ver as Branchs Remotas, utilizamos o seguinte comando para vermos quais Branchs existem fora do nosso computador(se o seu Repositório estiver bem atualizado)

**git branch -r**

Agora abaixo um exemplo dos dois comandos:

```
M4TRIX:projeto-git (master) > git branch
branch_teste
* master
nova_branch
M4TRIX:projeto-git (master) > git branch -r
origin/HEAD -> origin/master
origin/master
M4TRIX:projeto-git (master) > █
```

**origin** significa que a Branch que está apresentando está nos Servidores do Site onde se encontra o Repositório de forma Remota.

Agora que sabemos que estamos na Branch correta para trabalhar, podemos modificar o arquivo README.md que foi criado junto com o Repositório.

Pode modificar o arquivo em qualquer Programa de Edição de Texto de sua escolha, mas o mais tranquilo de mexer e que pode ser usado tranquilamente e é gratuito é o **VSCODE**.

Após modificado o arquivo, podemos ver quais arquivos foram modificados no Projeto usando esse próximo comando GIT apresentado:

## git status

Com o comando git status ele vai mostrar todas as modificações feitas em todos os arquivos que foram modificados, além disso irá mostrar os novos arquivos criados que ainda não existem salvos nas versões anteriores.

Abaixo o exemplo que eu falei acima:

```
M4TRIX:projeto-git (branch_teste *) > git status
On branch branch_teste
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
M4TRIX:projeto-git (branch_teste *) > █
```

quando se usa o comando git status ele auxilia ao desenvolvedor quais os Próximos passos que devem ser feitos para salvar as modificações, mas nós já chegaremos lá.

Irá aparecer em vermelho os arquivos que foram modificados e arquivos novos criados irão aparecer em uma área separada.

Irei criar um arquivo novo para mostrar a diferença:

```
M4TRIX:projeto-git (branch_teste *) > touch teste.txt
M4TRIX:projeto-git (branch_teste *) > git status
On branch branch_teste
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README.md

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

        teste.txt

no changes added to commit (use "git add" and/or "git commit -a")
M4TRIX:projeto-git (branch_teste *) > █
```

touch é um comando Linux para criar um novo arquivo vazio, somente para exemplo.

Como é um arquivo novo ele aparece mais abaixo na área de **Untracked files**, ou seja, ele ainda não foi adicionado aos arquivos já verificados pelo git, que são arquivos que já foram salvos anteriormente.

## Adicionando Arquivos para Salvar

Agora fizemos as modificações que queríamos fazer nos nossos arquivos, onde criamos e modificamos arquivos a vontade, agora, esse é o momento que queremos salvar as nossas modificações, onde iremos utilizar um comando que pode ser usado de diferentes formas, onde o comando principalmente é:

# git add file

O comando add ele vai adicionar os arquivos que desejamos salvar, onde podemos definir como desejamos salvar esses arquivos, onde irei apresentar alguns exemplos:

- **git add --all**: se queremos salvar todos os novos arquivos criados.



- **git add .** : irá salvar todos os novos arquivos dentro do Diretório atual que estamos mexendo
- **git add \*.extensão**: irá salvar todos os arquivos que possuem uma extensão específica em seu projeto
- **git add file.extensão**: irá salvar somente o arquivo file.extensão que foi chamado
- **git add file1.extensão file2.extensão**: podemos chamar cada um dos arquivos que queremos salvar, somente chamando nome a nome como acima, não tendo um limite de arquivos para adicionar

Para facilitar o entendimento, vamos ver no exemplo que fizemos anteriormente:

Eu vou salvar primeiro somente o novo arquivo teste que eu criei, portanto eu escrevo o seguinte comando: *git add teste.txt*

Depois irei verificar como anda o Status das Modificações, onde se ele tiver sido adicionado ele deve aparecer em verde no terminal, como abaixo:

```
M4TRIX:projeto-git (branch_teste *) > git add teste.txt
M4TRIX:projeto-git (branch_teste *) > git status
On branch branch_teste
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   teste.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README.md

M4TRIX:projeto-git (branch_teste *) > █
```

Se quiséssemos, poderíamos já fazer um Commit, porque já adicionamos um arquivo para ser salvo, mas temos outro que foi modificado e ainda não salvamos as modificações feitos nele.

Vou salvar o Arquivo usando outro tipo de comando, onde desejo que todos os arquivos com extensão *.md* sejam adicionados:



```
M4TRIX:projeto-git (branch_teste +) > git add *.md
M4TRIX:projeto-git (branch_teste +) > git status
On branch branch_teste
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.md
        new file:   teste.txt

M4TRIX:projeto-git (branch_teste +) > █
```

Foram pegos todos os arquivos com extensão *.md* no nosso Repositório e foram adicionados para poder serem salvos, onde agora como mostrado acima não temos mais arquivos que não estão prontos para serem salvos, todos foram adicionados.

## Salvando todas as Modificações

No Sistema GIT, quando dizemos que iremos salvar queremos dizer que iremos fazer um **Commit**.

Um Commit significa que tudo que foi modificado no arquivo está salvo no Repositório, ou seja, não irá perder o que foi modificado.

Um Commit além disso serve para vermos o que cada pessoa fez no Projeto, onde um Commit possui o nome do Desenvolvedor que fez aquelas modificações.

Podemos fazer quantos Commits quisermos em nossa Branch da forma que quisermos, mas existem algumas opções que devem ser as principais formas de se fazer um Commit, como esse abaixo:

**git commit -m "message"**

Esta estrutura de Commit é a mais usada de todas as Possíveis usadas, onde colocamos uma mensagem do que fizemos nos Arquivos, a tag -m é para podermos colocar uma mensagem que ficam entre aspas ("").

Existe mais duas formas de se podermos fazer um Commit, são elas:

- **git commit:** se simplesmente colocarmos *git commit* sem uma mensagem, vai ser aberto um editor de texto que vai pedir que escrevamos essa mensagem, por isso é mais rapido usarmos o *git commit -m* para simplificar e agilizar o trabalho
- **git commit -am "message":** esse comando é utilizado quando somente deletamos ou modificamos arquivos já existentes, sem termos criados novos arquivos, onde esse comando vai adicionar as modificações e fazer o commit em um único comando.

Então vamos ver o exemplo que estamos fazendo, agora que já adicionei quais arquivos desejo salvar, podemos fazer um Commit, como mostrado abaixo:

```
M4TRIX:projeto-git (branch_teste +) > git status
On branch branch_teste
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.md
        new file:   teste.txt

M4TRIX:projeto-git (branch_teste +) > git commit -m "Primeiro Commit Feito"
[branch_teste 8265469] Primeiro Commit Feito
 2 files changed, 1 insertion(+), 1 deletion(-)
 create mode 100644 teste.txt
M4TRIX:projeto-git (branch_teste) > █
```

Toda vez que você fizer um Commit ele vai mostrar um Código, esse código acima (8265469) são os primeiros 7 dígitos do Código único do nosso Commit, onde eu irei explicar mais a frente para o que ele serve.

Além do Código, temos quantos arquivos foram modificados(2 *files changed*), quantas linhas foram adicionadas(1 *insertion(+)*) e quantas linhas foram deletadas (1 *deletion(-)*).

Então com isso fizemos o primeiro Commit feito por um Desenvolvedor, onde quando criamos o Repositório no Github ele faz o Primeiro Commit automaticamente para adicionar o arquivo README.md.

## Enviando as Modificações Remotamente

No passo anterior, fizemos o nosso Commit se salvamos as modificações em nosso computador, mas somente nós podemos mexer nas modificações que foram feitas e nosso colegas Desenvolvedores não podem mexer no que você fez, por isso devemos enviar as modificações que fizemos para o Servidor Remoto onde o Repositório está vinculado, no nosso caso para o Github.

Para enviar Remotamente as nossas modificações, iremos usar um novo comando de GIT que vai pegar os nossos commit e atualizar a nossa Branch Remotamente. Se a Branch foi criada em seu computador e você nunca enviou nada por essa Branch, esse comando vai criar a Branch Remotamente com as suas Modificações.

Então o nosso novo comando é:

**git push origin branch**

O comando push nesse formato está dizendo que queremos enviar a branch específica para o servidor Remoto.

O origin, como dito anteriormente, significa que estamos querendo enviar a branch para a “origem”, ou seja, para o Servidor Remoto que está o nosso projeto.

Existem algumas formas extras de se usar o comando push, como esses abaixo:

- **git push:** dependendo da versão do GIT, esse comando pode ou enviar as modificações na sua Branch atual(versão 2.x) ou enviar todas as modificações em todas as Branchs(versão 1.x) por isso não é recomendado utilizar esse comando assim.
- **git push --set-upstream origin master:** a tag --set-upstream faz com que se essa for a primeira vez que você for enviar modificações de um Repositório que antes não tinha Remotamente possa vincular ela melhor, mas isso será apresentado melhor quando eu explicar o que é **REMOTE**.

Vamos voltar ao nosso exemplo, agora que fizemos o commit na branch de teste chamada *branch\_teste*, iremos criar essa branch Remotamente e enviar as modificações feito nelas:

```
M4TRIX:projeto-git (branch_teste) > git push origin branch_teste
Username for 'https://github.com': F4NT0
Password for 'https://F4NT0@github.com':
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 346 bytes | 346.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'branch_teste' on GitHub by visiting:
remote:   https://github.com/F4NT0/projeto-git/pull/new/branch_teste
remote:
To https://github.com/F4NT0/projeto-git.git
 * [new branch]      branch_teste -> branch_teste
M4TRIX:projeto-git (branch_teste) > █
```

No meu caso, toda vez que eu for enviar algo Remotamente eu devo colocar meu usuário do Github e minha senha do Github, como uma forma de segurança para evitar que alguém Remoto sem permissão possa enviar alguma modificação, mas tem casos que isso não precisa, onde pode se usar os comando de *global* para salvar suas informações e ele enviar automático.

O sistema irá verificar o que foi modificado, irá carregar e enviar as modificações feitas na minha Branch Remotamente, onde no final ele avisa que criou a branch nova Remotamente (*[new branch]*)

Além disso ele fala que se pode abrir um **Pull Request** onde eu irei explicar isso mais a frente.

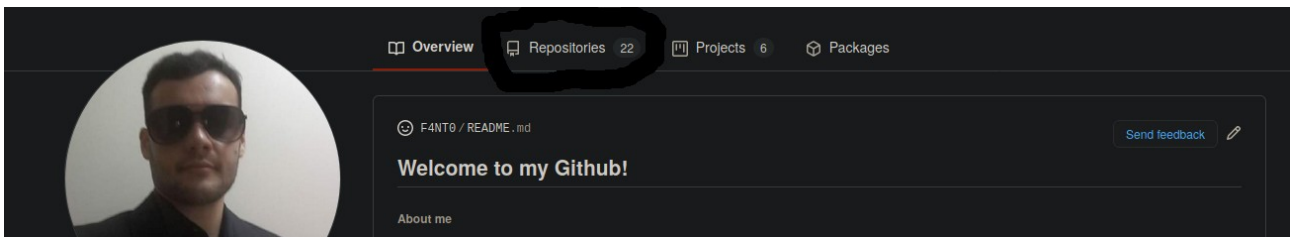
Dessa forma agora temos as modificações que fizemos de forma Remota, onde os Desenvolvedores podem ver o que fizemos.

## Verificando Remotamente as modificações

Agora que enviamos as modificações, vamos verificar se elas estão realmente no servidor do Github, onde iremos acessar o nosso Repositório Remoto.

Para isso existem os seguintes passos:

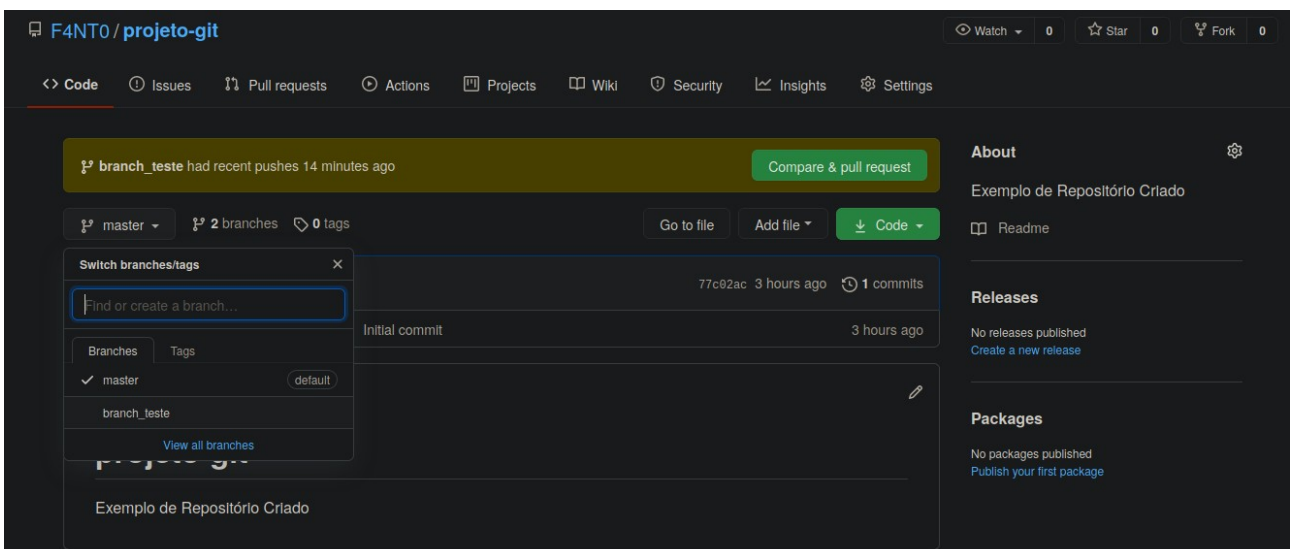
1. Acesse seu Github e vá ao seu Perfil
2. Em seu Perfil tem a aba **Repositories** que possui todos os seus Repositórios já criados:



3. Clique no nome do Repositório que estavamos trabalhando:

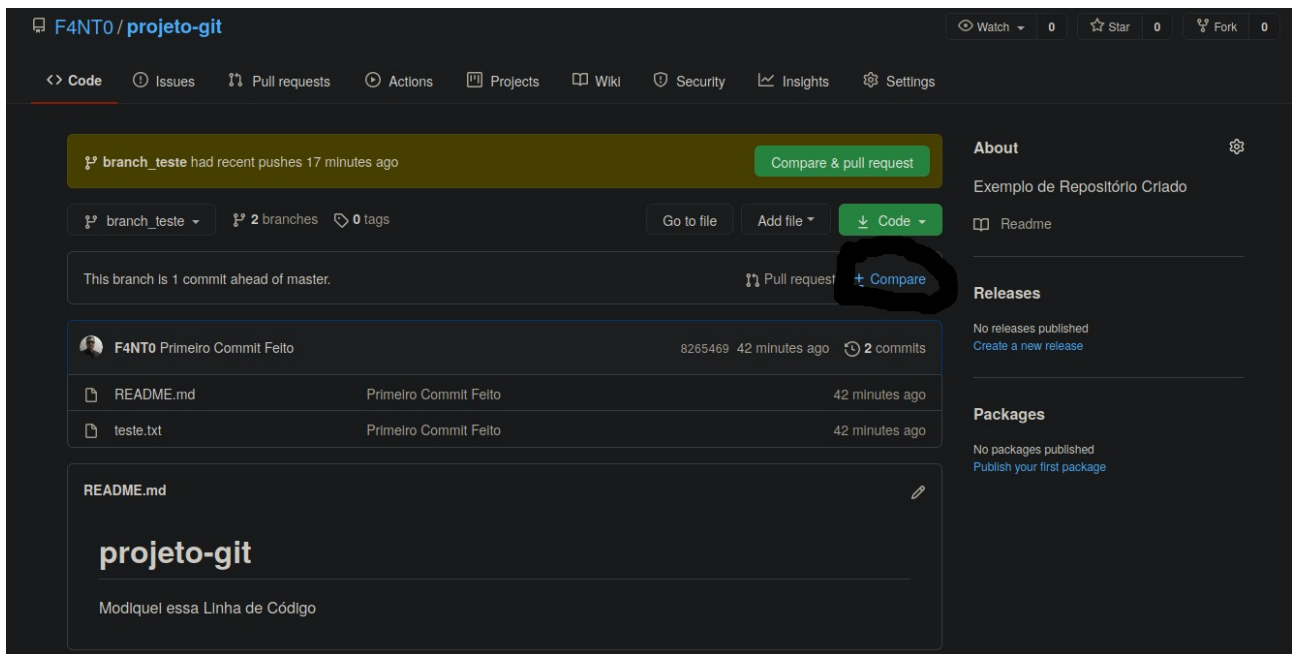


4. Agora iremos selecionar das Branchs Disponiveis a Branch que estávamos mexendo:



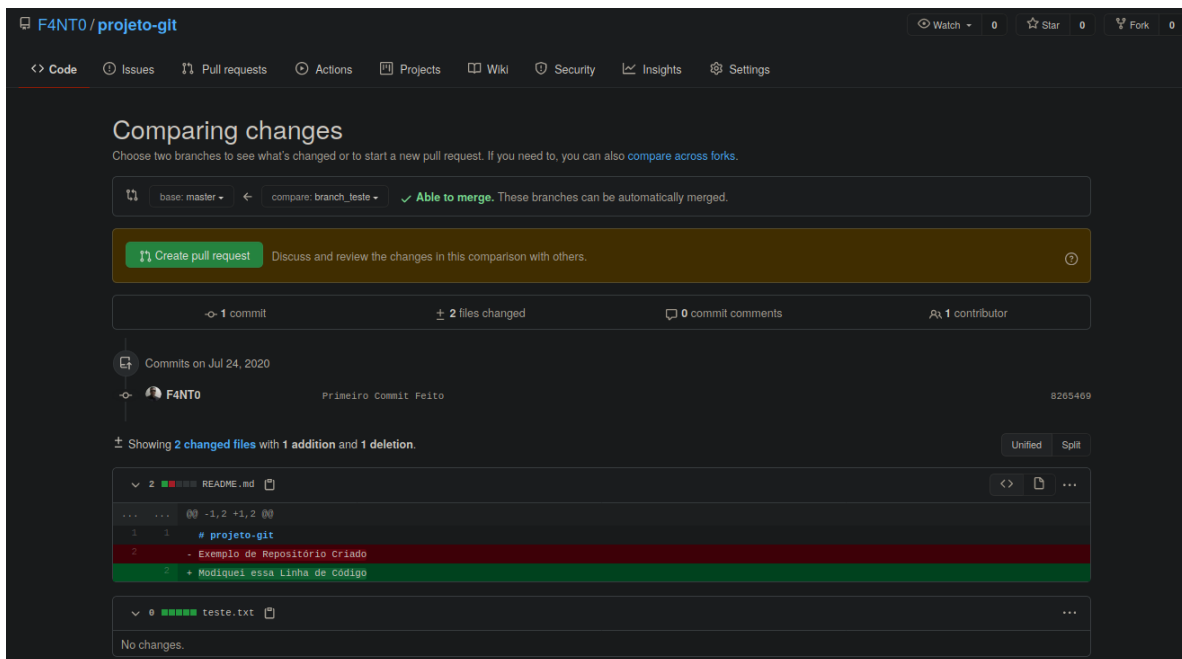
Podemos escolher a branch somente clicando no nome dela das opções de branches na extrema esquerda da tela, como mostrado acima.

5. Agora que acessamos a Branch que mexemos, podemos ver que possui o último Commit que fizemos, onde agora iremos clicar em **Compare** no canto direito para vermos as modificações feitas nessa Branch:



6. Na tela de modificações feitas, irá mostrar em vermelho o código deletado e em verde o código que foi colocado no lugar se o arquivo foi modificado e se o arquivo foi recém adicionado ele vai mostrar o nome do arquivo mas não tem nenhuma modificação feita nele.

Aparecem também no topo que ele verifica se a Branch atual pode ser enviada para a Branch master, onde isso se chama **merge** que eu irei explicar em seguida.

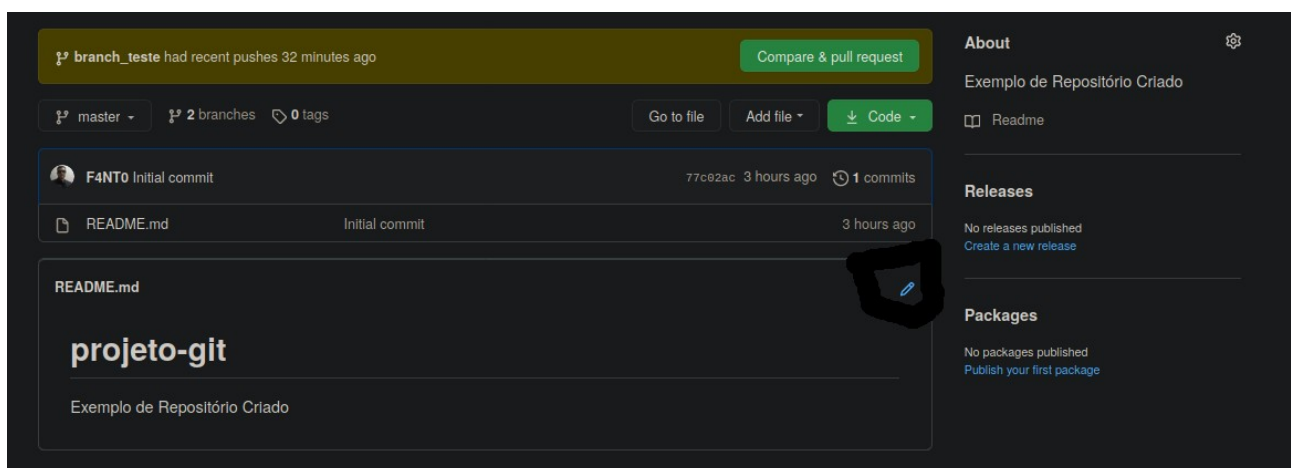


## Editando arquivos Remotamente

Nos servidores Remotos como Github, Gitlab e Bitbucket podemos modificar os arquivos do nosso Repositório de forma bem simples, onde podemos adicionar códigos ou adicionar novos arquivos de texto.

Como exemplo, irei modificar de novo o arquivo README.md pelo Github, da seguinte forma:

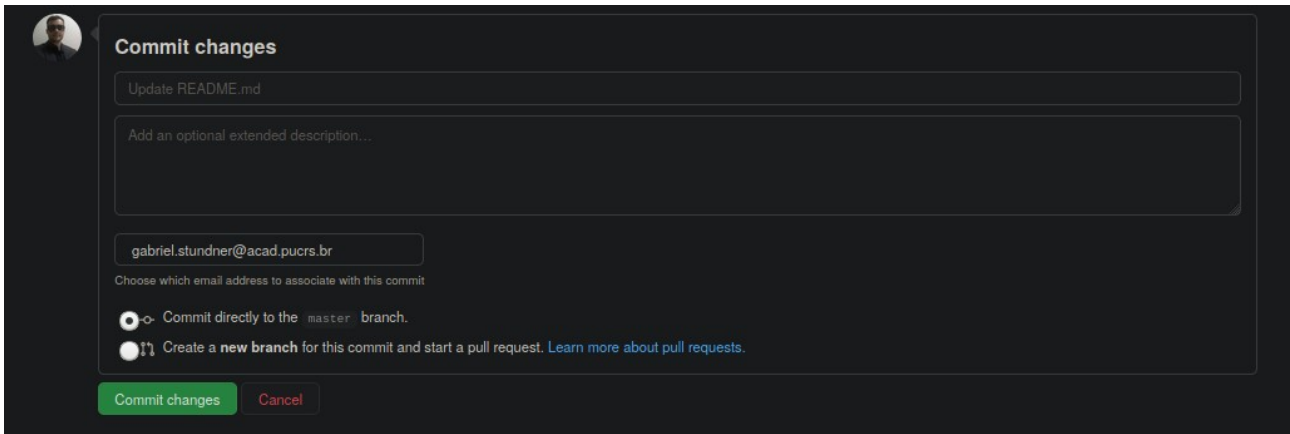
1. Vou até a página inicial do nosso Repositório
2. Irei clicar no símbolo do Lápis, que aparece no canto direito do README para poder acessar a página de edição dentro do Github:



3. Agora, com a tela de edição aberta, posso escrever o que eu quiser e depois de finalizado, eu somente clico em **Commit Changes**,



onde dessa forma é feito um Commit pelo Github com as modificações que eu fiz, onde dessa vez eu irei enviar direto na *master*.



Dessa forma então, fizemos uma modificação no Arquivo e portanto agora o nosso projeto possui um Commit salvo Remotamente, mas esse Commit não existe em nosso computador.

## Baixando modificações feitas para o computador

Agora possuímos duas Versões Diferentes do Projeto, uma Remotamente e uma Localmente, então se formos mexer na *master* no nosso computador e tentar enviar não iremos conseguir, como mostra o exemplo abaixo, fazendo todos os passos já apresentados:

```
M4TRIX:projeto-git (branch_teste) > git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
M4TRIX:projeto-git (master) > git push origin master
Username for 'https://github.com': F4NT0
Password for 'https://F4NT0@github.com':
To https://github.com/F4NT0/projeto-git.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/F4NT0/projeto-git.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
M4TRIX:projeto-git (master) > █
```

A mensagem que está mostrando abaixo do *[rejected]* é uma dica explicando que existem modificações Remotas por isso não conseguimos enviar qualquer modificação dessa Branch para o Servidor Remoto, portanto devemos usar um comando novo que serve para pegarmos as modificações que estão lá no Servidor Remoto:



# git pull

O comando `git pull` irá pegar todas as modificações feitas no *origin*, que é o nosso Repositório Remoto, e vai atualizar a nossa Branch que estamos trabalhando:

```
M4TRIX:projeto-git (master) > git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/F4NT0/projeto-git
   77c02ac..cc3e779  master    -> origin/master
Updating 77c02ac..cc3e779
Fast-forward
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
M4TRIX:projeto-git (master) > █
```

Mostra as modificações feitas e o código do Commit do HEADER novo

Agora então temos a nossa Branch atualizada com a Branch Remota no Servidor do Github.

## Como voltar um Commit feito

Nós voltamos um Commit caso alguma coisa deu de errado e queremos voltar para uma versão do Programa que o código funcionava, mas primeiro devemos verificar todos os commits feitos na Branch, usando o seguinte comando:

# git log

Esse comando vai fazer com que a gente possa ver todos os commits feitos e possamos descobrir qual que queremos voltar, como no exemplo abaixo:

```
M4TRIX:projeto-git (master) > git log
commit f13e9d1b60acfcc4a62532f7aecfae224fa6701e (HEAD -> master, origin/master, origin/HEAD)
Author: Gabriel Fanto Stundner <gabriel.stundner@acad.pucrs.br>
Date: Mon Jul 27 12:15:50 2020 -0300

    Initial commit
M4TRIX:projeto-git (master) > █
```

Onde está escrito *commit f13e9d1b60acfcc4a62532f7aecfae224fa6701e* é o id desse commit em específico, onde iremos precisar somente dos primeiro 6 números para voltarmos um Commit

Vamos dizer que criamos um Commit com modificações que deram problema no código e queremos voltar ao Commit anterior, como no exemplo abaixo:

```
M4TRIX:projeto-git (master) > git log
commit 5772b6248da7ddf5cc761d809a07e74008829468 (HEAD -> master)
Author: F4NT0 <gabriel.stundner@acad.pucrs.br>
Date: Mon Jul 27 13:39:25 2020 -0300

    Commit ruim

commit f13e9d1b60acfcc4a62532f7aecfae224fa6701e (origin/master, origin/HEAD)
Author: Gabriel Fanto Stundner <gabriel.stundner@acad.pucrs.br>
Date: Mon Jul 27 12:15:50 2020 -0300

    Initial commit
M4TRIX:projeto-git (master) > █
```

Podemos

ver que o Commit que queremos desfazer tem como id simplificado *5772b62* e é com esse código que iremos usar o novo comando chamado *revert*:

## git revert id-commit

Então como no exemplo trabalhado vamos reverter o commit *572b62*, onde esse comando vai criar um Commit novo com a versão do código anterior, o que é melhor que sumir o commit, podendo ser útil num futuro, portanto nunca é bom apagar commits criados

Se você utilizar a tag **--no-edit** de vez dele abrir uma tela do editor de texto, ele vai já criar direto o commit com as modificações do commit anterior, como abaixo:

```
M4TRIX:projeto-git (master) > git revert --no-edit 5772b62
[master 2695bca] Revert "Commit ruim"
Date: Mon Jul 27 13:49:10 2020 -0300
1 file changed, 1 insertion(+), 1 deletion(-)
M4TRIX:projeto-git (master) > git log
commit 2695bca3cb17c8c05a9e0664efb3e8189275c862 (HEAD -> master)
Author: F4NT0 <gabriel.stundner@acad.pucrs.br>
Date: Mon Jul 27 13:49:10 2020 -0300

    Revert "Commit ruim"

    This reverts commit 5772b6248da7ddf5cc761d809a07e74008829468.

commit 5772b6248da7ddf5cc761d809a07e74008829468
Author: F4NT0 <gabriel.stundner@acad.pucrs.br>
Date: Mon Jul 27 13:39:25 2020 -0300

    Commit ruim

commit f13e9d1b60acfcc4a62532f7aecfae224fa6701e (origin/master, origin/HEAD)
Author: Gabriel Fanto Stundner <gabriel.stundner@acad.pucrs.br>
Date: Mon Jul 27 12:15:50 2020 -0300

    Initial commit
```

Se deseja saber qual commit voltar, pode usar o comando checkout para voltar a um commit desejado, como no comando abaixo:

```

M4TRIX:projeto-git (master) > git log
commit 2695bca3cb17c8c05a9e0664efb3e8189275c862 (HEAD -> master)
Author: F4NT0 <gabriel.stundner@acad.pucrs.br>
Date:   Mon Jul 27 13:49:10 2020 -0300

    Revert "Commit ruim"

    This reverts commit 5772b6248da7ddf5cc761d809a07e74008829468.

commit 5772b6248da7ddf5cc761d809a07e74008829468
Author: F4NT0 <gabriel.stundner@acad.pucrs.br>
Date:   Mon Jul 27 13:39:25 2020 -0300

    Commit ruim

commit f13e9d1b60acfcc4a62532f7aecfae224fa6701e (origin/master, origin/HEAD)
Author: Gabriel Fanto Stundner <gabriel.stundner@acad.pucrs.br>
Date:   Mon Jul 27 12:15:50 2020 -0300

    Initial commit
M4TRIX:projeto-git (master) > git checkout f13e9d1
Note: checking out 'f13e9d1'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

    git checkout -b <new-branch-name>

HEAD is now at f13e9d1 Initial commit
M4TRIX:projeto-git ((f13e9d1...)) > █

```

Dessa forma, estamos em outro commit de forma temporária, onde podemos ver o que fizemos de modificação e se esse commit é o commit que precisamos voltar.

Para voltar a Branch no Commit original somente use o comando checkout normal: **git checkout nome-branch**

Se desejamos voltar vários commits, devemos saber quantos commits serão retornados, onde podemos usar o comando revert como abaixo para retornar n números de commits:

**git revert HEAD~n**

Irei mostrar por exemplo, voltar 2 commit, desde o HEAD, que é o ultimo commit feito, então o comando fica: **git revert HEAD~1**

Esse comando vai criar um commit novo onde ele vai voltar os Status dos Arquivos do commit HEAD atual mais um commit anterior, como no exemplo abaixo:

```
M4TRIX:projeto-git (master) > git log
commit 4725df62a8523fcb1dc8a803e6fe19ccff0dea18 (HEAD -> master)
Author: F4NT0 <gabriel.stundner@acad.pucrs.br>
Date:   Mon Jul 27 14:42:10 2020 -0300

    modificação 2

commit ed0f9226432608097f17d35e9d311efea48a87c6
Author: F4NT0 <gabriel.stundner@acad.pucrs.br>
Date:   Mon Jul 27 14:41:50 2020 -0300

    modificação 1

commit f13e9d1b60acfcc4a62532f7aecfae224fa6701e (origin/master, origin/HEAD)
Author: Gabriel Fanto Stundner <gabriel.stundner@acad.pucrs.br>
Date:   Mon Jul 27 12:15:50 2020 -0300

    Initial commit
M4TRIX:projeto-git (master) > git revert HEAD~1
error: could not revert ed0f922... modificação 1
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
M4TRIX:projeto-git (master *|REVERTING) > git add .
M4TRIX:projeto-git (master *|REVERTING) > git commit -m "voltando ao inicio"
[master 1f5c5f6] voltando ao inicio
 1 file changed, 1 insertion(+), 1 deletion(-)
M4TRIX:projeto-git (master) > █
```

Os conflitos que ocorrerem devem ser lidados em um Editor de Texto de sua Preferência, mas o VSCODE que eu uso é bem simples de mexer. Dessa forma, voltamos as modificações para o primeiro commit feito no sistema.

## Unindo duas Branchs fazendo Merge

Merge é algo que iremos fazer muito durante todo o tempo de desenvolvimento, sendo algo que se não tiver as ferramentas certas pode ser uma baita dor de cabeça.

Merge é a união de duas Versões de Arquivos em Branchs diferentes que precisamos unir em uma única versão, onde o GIT vai verificar as duas versões e verificar se tem algum **Conflito**

**Conflito** acontece quando nas duas Versões do Arquivo foi modificado a mesma linha de código, onde o Desenvolvedor deve verificar manualmente qual das Versões irá ser a versão oficial da linha modificada.

Para se fazer um Merge, primeiro devemos seguir alguns passos:

1. Se você estiver na Branch com as modificações que quer enviar para a oficial, utilize o comando *checkout* para ir para a Branch que deseja enviar essas modificações, por exemplo, se você estiver na Branch Teste-1 e deseja enviar as modificações para a Master, faça *git checkout master* para ir á Branch onde será feito o Merge
2. Agora que estamos na Branch desejada, podemos fazer o merge pegando a Branch Teste-1 com o seguinte comando:

**git merge branch**

**branch** vai ser o nome da nossa Branch: Teste-1

3. Se tiver algum arquivo que foi modificado a mesma linha nas duas Branchs, vai ocorrer um conflito, mas para resolvermos ele iremos abrir um Editor de Texto que nos auxilie para vermos as modificações, no meu caso irei usar o VSCODE.

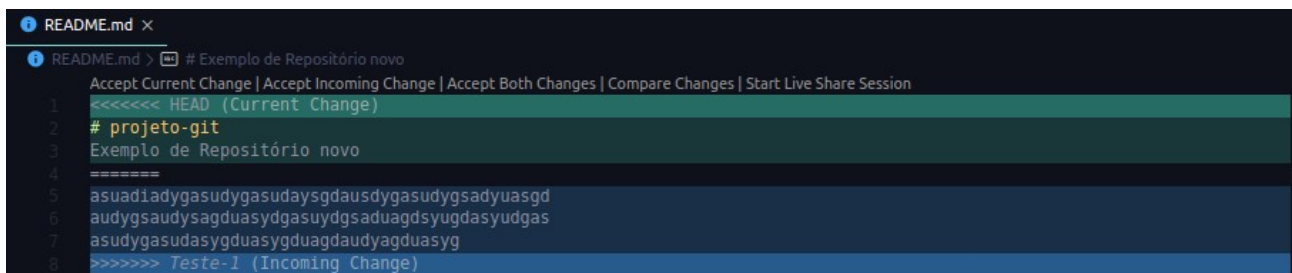
Vamos ver o caso 1 e 2 no Terminal:

```

M4TRIX:projeto-git (master) > git checkout -b Teste-1
Switched to a new branch 'Teste-1'
M4TRIX:projeto-git (Teste-1) > code .
M4TRIX:projeto-git (Teste-1) > git add .
M4TRIX:projeto-git (Teste-1) > git commit -m "modif no Teste-1"
[Teste-1 59f36f3] modif no Teste-1
 1 file changed, 3 insertions(+), 2 deletions(-)
M4TRIX:projeto-git (Teste-1) > git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
M4TRIX:projeto-git (master) > git add .
M4TRIX:projeto-git (master) > git commit -m "teste"
[master b261648] teste
 1 file changed, 1 insertion(+), 1 deletion(-)
M4TRIX:projeto-git (master) > git merge Teste-1
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
M4TRIX:projeto-git (master *+|MERGING) > █

```

Aconteceu um conflito porque modifiquei as Duas Branchs na mesma Linha, onde no VSCODE vai aparecer as opções para lidar com o conflito:



Dai o Desenvolvedor deve escolher qual das Versões é a Oficial, onde tem as seguintes opções bem acima:

- **Accept Current Change:** Vai ser aceito a Versão da Branch Atual(no caso em master)
- **Accept Incoming Change:** Vai ser aceito a Versão da Branch que está enviando as modificações(no caso em Teste-1)
- **Accept Both Changes:** Vai aceitar a modificação das duas Branchs

Após o Desenvolvedor escolher a opção desejada, se deve adicionar as modificações com *add* e depois fazer um *commit* novo:



## Utilizando o git remote

remote é um comando do Git que é muito importante, tanto para enviar modificações para diferentes Repositórios como para configurar para onde o seu Repositório Interno do Computador vai para o Repositório Remoto.

Se desejamos adicionar um URL para um Repositório Interno recém criado usamos o seguinte comando:

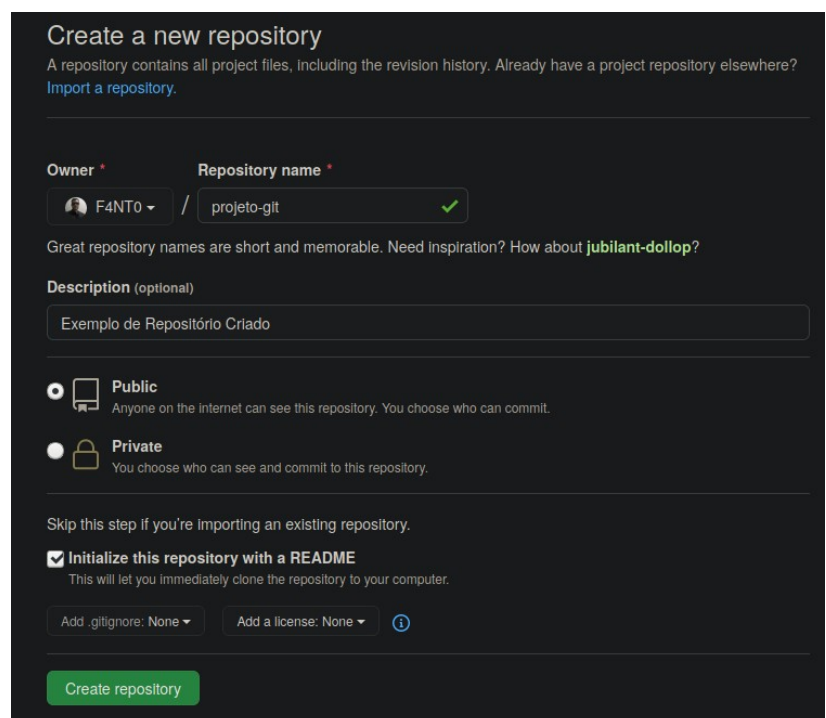
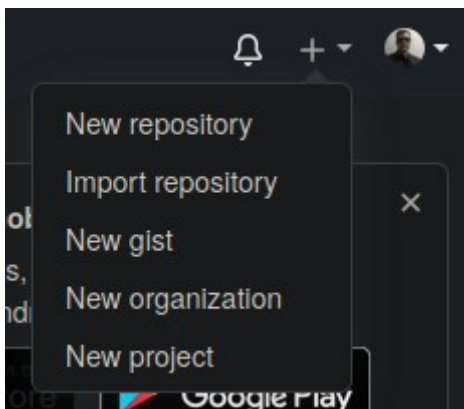
**git remote add origin url**

o comando remote add irá fazer com que o Repositório origin Remoto seja o repositório entrado pelo url.

Com esse comando podemos concluir a criação da primeira forma de criação de um Repositório, relembrando:

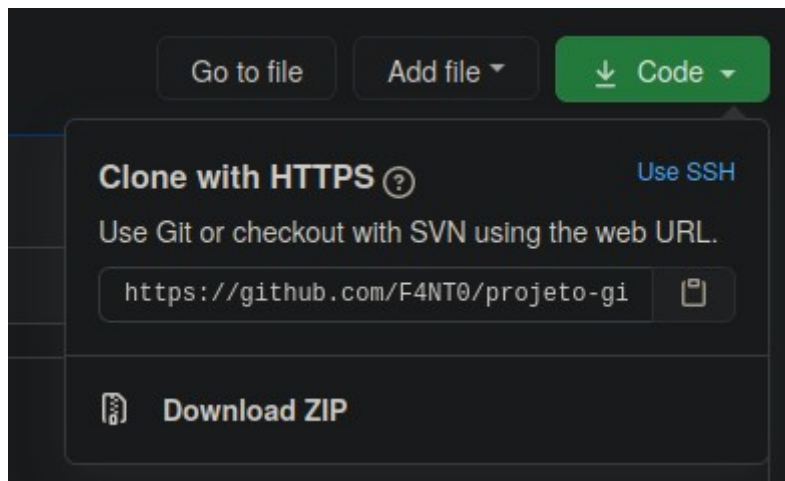
- **git init:** transforma um Diretório em um Repositório, onde com esse comando é criado o Diretório *.git* dentro do Diretório atual

Agora criamos um Repositório Remoto como na segunda forma de criar um Repositório, onde criamos um Repositório Remoto no Github:

A screenshot of the 'Create a new repository' form on the GitHub website. The form is titled 'Create a new repository' and includes a subtitle: 'A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.' The form fields include: 'Owner' (F4NT0), 'Repository name' (projeto-git), 'Description (optional)' (Exemplo de Repositório Criado), 'Public' (selected), 'Private' (unselected), 'Initialize this repository with a README' (checked), 'Add .gitignore: None', and 'Add a license: None'. A green 'Create repository' button is at the bottom.

Copiamos o URL do Repositório do Github:





Esse é o URL que iremos adicionar no comando do Remote, como no exemplo abaixo:

```
M4TRIX:GIT > cd projeto-git-2
M4TRIX:projeto-git-2 > git init
Initialized empty Git repository in /home/f4nt0/Desktop/GIT/projeto-git-2/.git/
M4TRIX:projeto-git-2 (master #) > git remote add origin https://github.com/F4NT0/projeto-git.git
M4TRIX:projeto-git-2 (master #) > git pull origin master
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/F4NT0/projeto-git
 * branch      master      -> FETCH_HEAD
 * [new branch] master      -> origin/master
M4TRIX:projeto-git-2 (master) > git log
commit f13e9d1b60acfcc4a62532f7aecfae224fa6701e (HEAD -> master, origin/master)
Author: Gabriel Fanto Stundner <gabriel.stundner@acad.pucrs.br>
Date:   Mon Jul 27 12:15:50 2020 -0300

    Initial commit
M4TRIX:projeto-git-2 (master) > █
```

No exemplo acima, criamos um Diretório vazio chamado projeto-git-2 depois iniciamos esse Diretório como um Repositório com *git init*, usamos o comando do *git remote add* para podermos fazer com que esse Repositório receba e envie informações para o repositório Remoto, que foi o que fizemos usando o comando *git pull* que pegou as modificações Remotas e adicionou nesse Repositório local que não havia nada e agora ele está atualizado com o que existe no Repositório Remoto.

Além de Adicionarmos a conexão com o Repositório Remoto, também podemos remover uma conexão usando o seguinte comando:

# git remote remove origin

O *remove* vai procurar na lista de conexões qual a conexão que possui o nome entrado no lugar do *origin* ou o origin em si, dessa forma ele vai deletar a conexão que for entrado.

Mas como saber qual conexão remover? Usamos o seguinte comando para listar todos os remotes que existem:

## git remote -v

abaixo um exemplo pelo Terminal:

```
M4TRIX:projeto-git-2 (master) > git remote -v
origin https://github.com/F4NT0/projeto-git.git (fetch)
origin https://github.com/F4NT0/projeto-git.git (push)
M4TRIX:projeto-git-2 (master) > git remote remove origin
M4TRIX:projeto-git-2 (master) > git remote -v
M4TRIX:projeto-git-2 (master) > █
```

Com remote podemos adicionar vários Repositórios juntos, onde podemos enviar todas as modificações para os dois Repositórios de uma vez.

Para enviar pela primeira vez as modificações, devemos usar um tipo especial de push:

### **git push --set-upstream origin master**

com ele, vai ser configurado que as mudanças estão vindo desse Repositório interno para o Remoto

```
M4TRIX:projeto-git-2 (master *) > git add .
M4TRIX:projeto-git-2 (master +) > git commit -m "update"
[master 9c49159] update
 1 file changed, 1 insertion(+), 2 deletions(-)
M4TRIX:projeto-git-2 (master) > git remote add origin https://github.com/F4NT0/projeto-git.git
M4TRIX:projeto-git-2 (master) > git push --set-upstream origin master
Username for 'https://github.com': F4NT0
Password for 'https://F4NT0@github.com':
Counting objects: 3, done.
Writing objects: 100% (3/3), 275 bytes | 275.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/F4NT0/projeto-git.git
   f13e9d1..9c49159  master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
M4TRIX:projeto-git-2 (master) > █
```

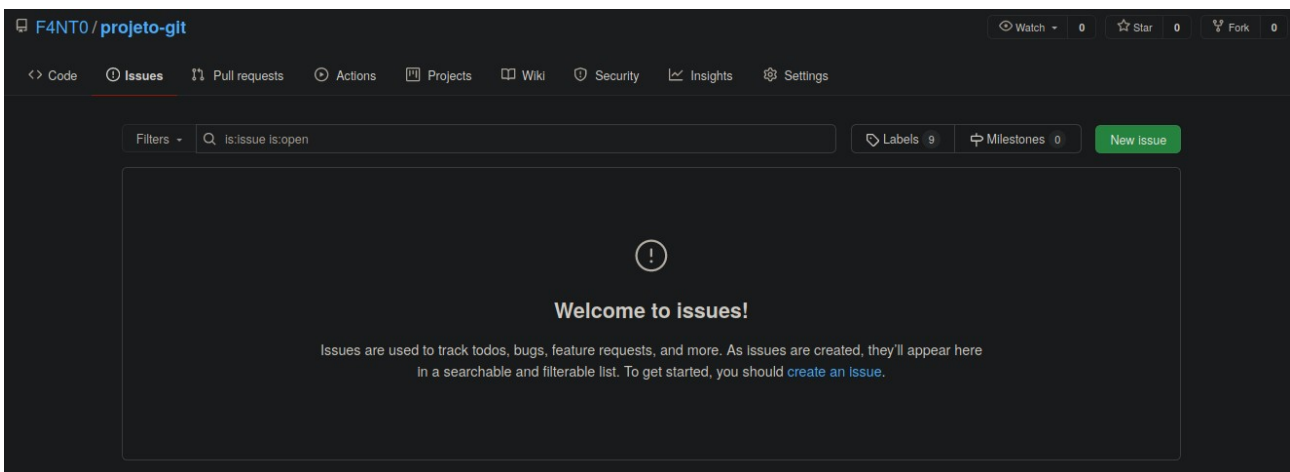
## Interagindo com o Github

Agora que sabemos mexer com o Básico de GIT, iremos agora para a Interação mais forte dentro do Github, onde primeiro devemos entender algumas coisas, como por exemplo as seguintes palavras:

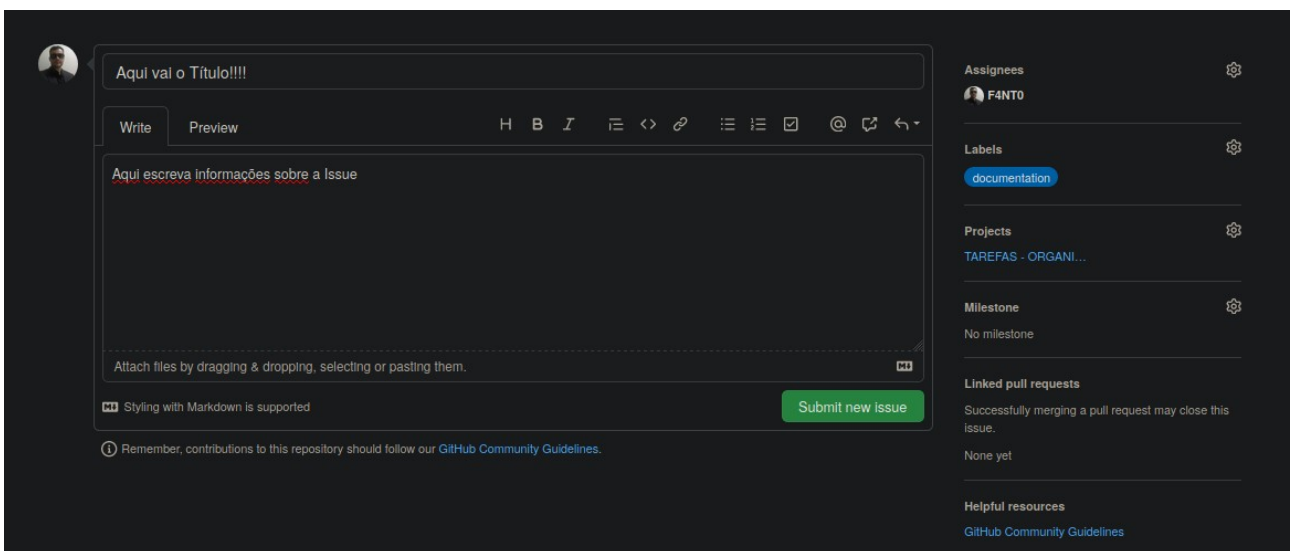
- **Issue:** Issues são uma forma de Gerenciar o que deve ser feito no Projeto, onde cada Issue é uma Tarefa a ser feita por um Desenvolvedor, qualquer um pode abrir uma Issue no Projeto, onde normalmente é o Gerente de Projetos, Usuários ou outro Desenvolvedor.

Como Abrir uma Issue no Github:

1. Abra o Repositório que deseja abrir uma Issue, depois clique na Aba onde está escrito **Issue:**



2. Para criar uma Issue, clique no Botão verde escrito **New Issue:**



Como pode ser visto acima, para criar uma nova Issue tem algumas informações que devem ser dadas:

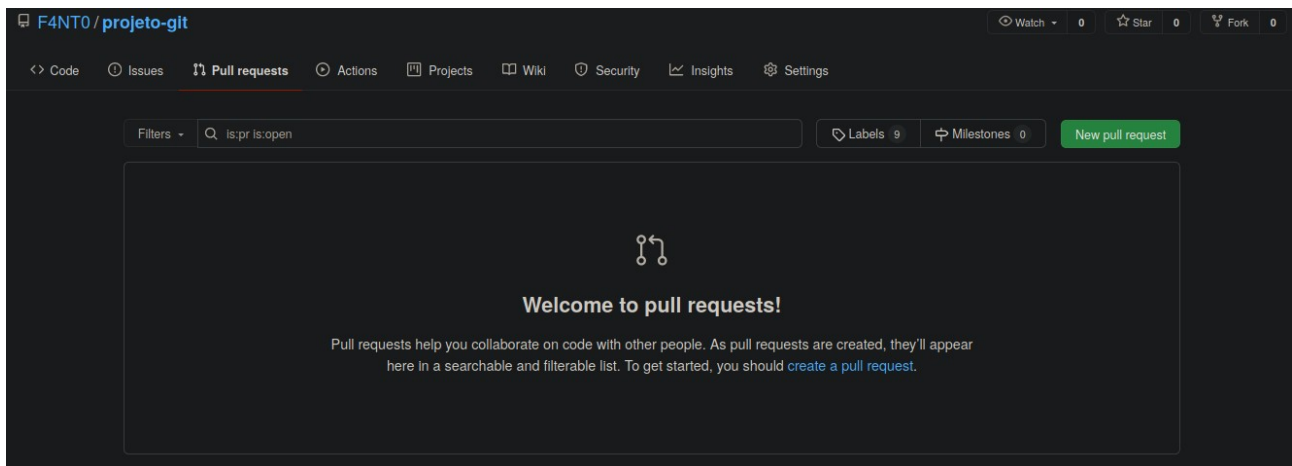
- Título da Issue, que é o nome da Tarefa
- Descrição do que deve ser feito
- **Assignees:** A pessoa ou Pessoas que irão fazer essa Issue
- **Labels:** Podemos definir o que essa Issue Representa no Projeto
- **Projects:** Qual Projeto está vinculado essa Issue
- **Milestone:** podemos organizar um Milestone, que significa que podemos definir uma Milestone como uma Sprint, onde todas as Issues e Pull Request podem estar vinculados nessa Milestone.
- **Linked Pull Request:** Podemos definir a qual Pull Request cada Issue está vinculada, para que quando for finalizado o Pull Request todas as Issues Vinculadas serão encerradas Junto

Cada Issue Possui um único ID que a estrutura é **#n**, esse ID é muito importante porque sempre que usarmos essa estrutura nos Textos ou informações, ele vai vincular um Link direto para a Issue, mas irei explicar isso mais Tarde.

- **Pull Request:** Um Pull Request é um Auxílio para poder fazer Merge de uma Branch para outra, onde os site de Repositório Remotos como Github, Gitlab, Bitbucket entre outros auxilia nessa Tarefa Ardua. Um Pull Request vai pegar as informações das Branchs, origem e destino, e vai verificar se não vai ocorrer algum conflito e avisar se pode ou não fazer Merge.

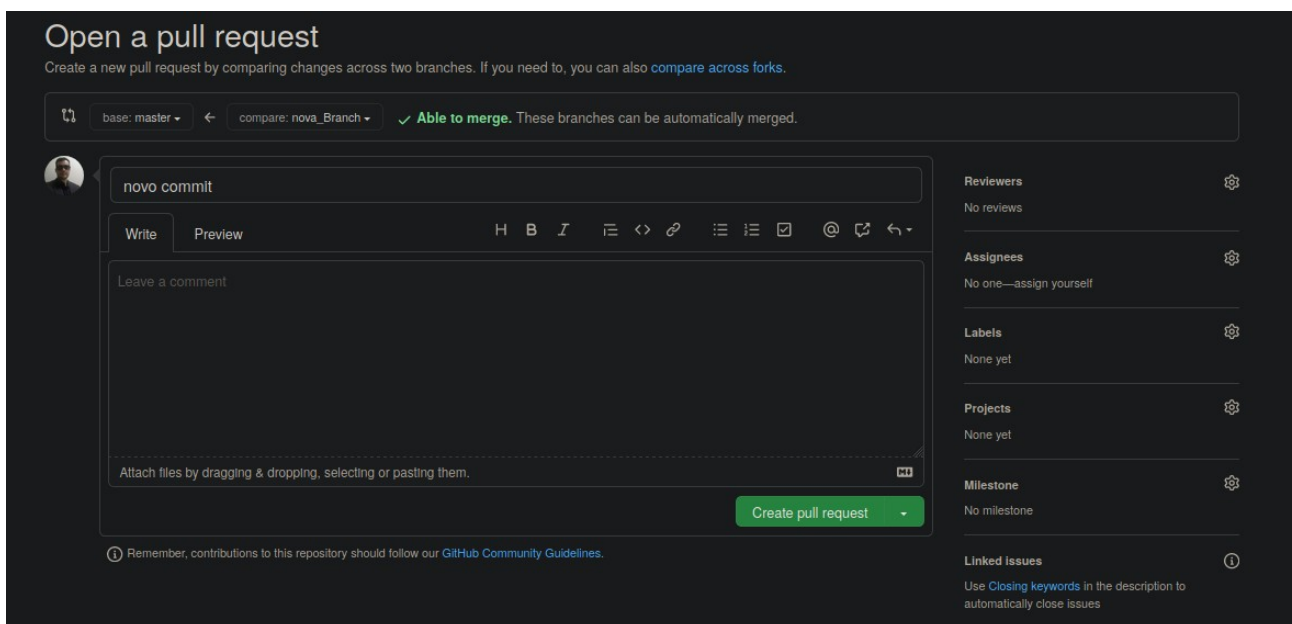
Na maioria das organizações de Desenvolvimento, quando um Desenvolvedor abre um Pull Request, outros dois Desenvolvedores devem verificar se as modificações feitas na Branch Origem são importantes para enviar para a Branch Destino.

Um Pull Request pode ser aberto indo na Aba de Pull Request do seu Repositório:



Se deseja abrir um Pull Request, clique no Botão Verde **New pull request**.

Ele vai mostrar a seguinte Página:



Bem encima, pode ver a Branch Origem, na parte **compare** e a Branch Destino na parte **base** onde possui uma seta mostrando de qual Branch vai para qual Branch.

Como não ocorreu conflitos, aparece uma mensagem dizendo **Able to merge**, avisando aos Desenvolvedores que se pode fazer o Pull Request sem ter que lidar com algum conflito.

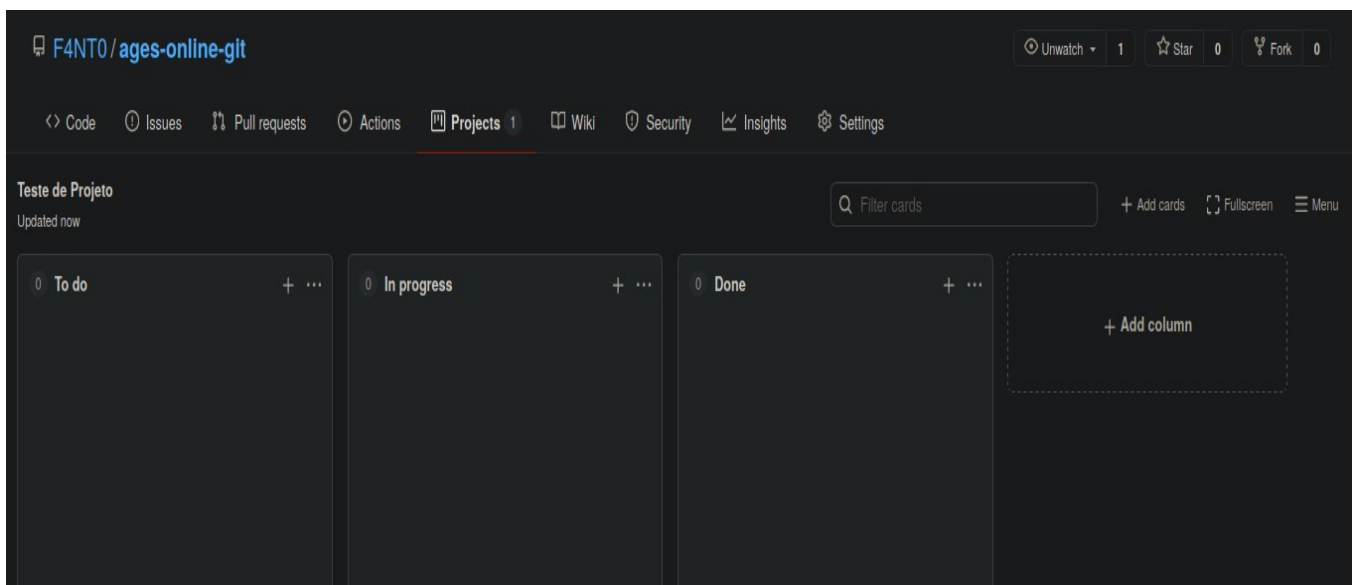
Assim como nas Issues, se deve:

- Colocar um Título para o Pull Request
- Colocar uma Descrição do que está sendo enviado para a Branch Destino

- **Reviewers:** Podemos avisar outros Desenvolvedores que criamos um Pull Request e precisamos que eles verifiquem todas as Modificações e deem seu aval para permitir o Pull Request.
- **Assignees:** Uma boa Prática de desenvolvimento é que quando foi avaliado e permitido o Pull Request, quem deve fazer o Pull Request é a pessoa que o criou, portanto o Assignees é a pessoa que abriu o Pull Request
- **Labels:** Podemos criar Labels também para os Pull Request se desejar.
- **Projects:** Qual projeto se encontra as Issues que foram resolvidos nesse Pull Request
- **Milestone:** Organização da Sprint ou alguma outra forma de Organização

Após os Reviewers derem seu aval, o Pull Request pode ser resolvido e por fim conectado tudo da Branch Origem na Branch Destino.

- **Projects:** Não só no Github mas em outros Sites de Armazenamento Remoto de Repositórios podemos organizar nossas Issues e Pull Requests em uma ordem Ágil, conhecido como Kanban, onde possui 3 tabelas que podemos colocar nossas Issues, onde as 3 tabelas tem esses nomes:
  - **To do:** Todas as Issues que devem ser feitas
  - **In Progress:** Todas as Issues que estão sendo Trabalhadas
  - **Done:** Todas as Issues que foram resolvidas e fechadas

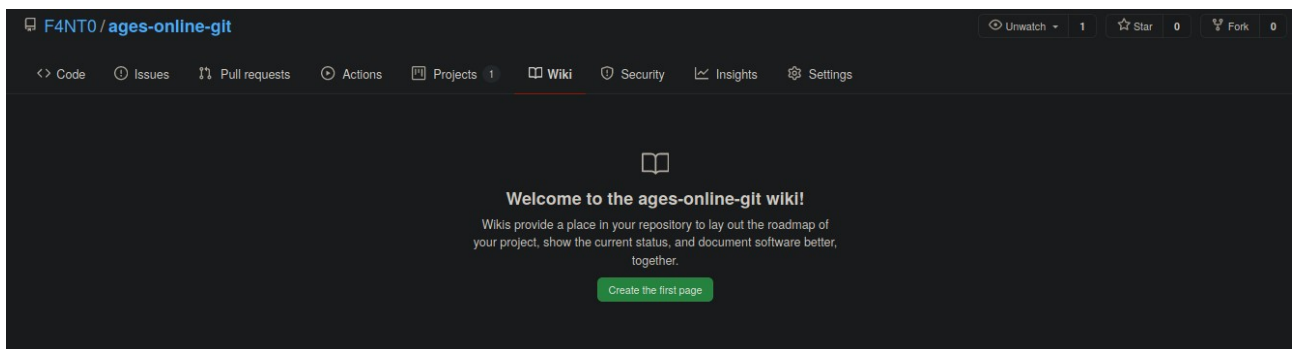


## Entendendo Estrutura de Texto de Repositórios Remotos

Toda vez que criamos um Repositório Remoto ele possui um Arquivo chamado **README.md**, esse Arquivo ele utiliza uma Linguagem de Marcação Chamada **Markdown**.

Essa Linguagem também é Utilizada em todas as Páginas de uma **Wiki** de um Repositório.

Uma Wiki é um Repositório Remoto dentro do Repositório Remoto no Github, onde podemos acessar ela pela sua aba especial:



Vou criar com vocês uma Página na Wiki para exemplo, clicando no Botão verde **Create the First Page**.

### Create new page

Home

Write Preview

h1 h2 h3 **B** *i* <> Edit mode: Markdown

Welcome to the ages-online-git wiki!

Edit message

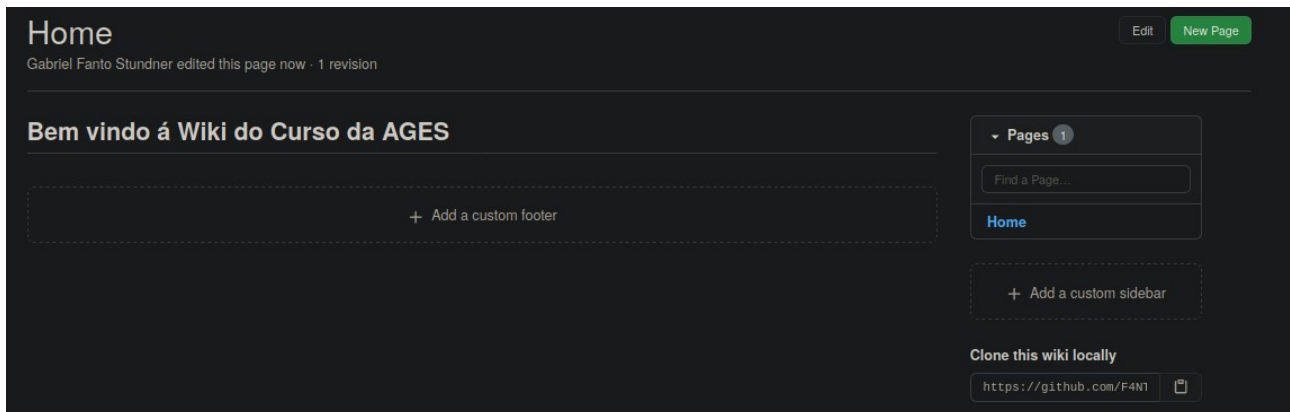
Initial Home page

Save Page

Nessa página, colocamos o Título da Página desejada e depois o Texto escrito em Markdown, onde irei apresentar como no próximo capítulo.

Criei uma Simples Página, onde para salvar as Modificações, clique em **Save Page**.

Agora que temos a Página, ela fica assim:



Como vocês podem ver, tem uma opção mais abaixo no lado direito onde está escrito **Clone this wiki locally**, é nesse momento que podemos Clonar a Wiki como um Repositório e a modificar em um Editor de Texto ou IDE, auxiliando muito na construção de uma boa wiki, explicando todas as informações importantes do projeto para que os Usuários ou outros Desenvolvedores possam saber como anda o Projeto e informações importantes.

O **README.md** serve para dar abertura para o Projeto, onde podemos colocar informações importante de como utilizar o Projeto ou como instalar, fica aberto para os desenvolvedores explicarem o que precisa ser feito no Sistema

Para saber como Escrever em Markdown, acesse este Link com todos os comandos e como escrever eles:

[https://f4nt0.github.io/PR0GR4M1NG/pages/tut\\_pages/basic.html](https://f4nt0.github.io/PR0GR4M1NG/pages/tut_pages/basic.html)



## Dicas de Organização de Equipe

1. Sempre crie um Repositório Remoto e faça clone dele para trabalhar localmente
2. Todo Projeto deve ter uma **Wiki** que deve sempre ser atualizada com informações do Projeto e organizações da equipe
3. Toda Branch deve possuir uma estrutura onde tenha o nome do Desenvolvedor, a tarefa que vai ser desenvolvida e a Sprint que está sendo desenvolvido

- **fanto\_telaInicial\_sprint01**

4. Antes de Desenvolver uma Tarefa primeiro crie uma Issue dessa Tarefa e pegue o ID dela

- ex: **#19 Criar Tela Inicial**

5. Quando for fazer um Commit na Branch da Tarefa sempre coloque o ID antes do que foi feito naquele Commit:

- **git commit -m "#19 finalizado texto"**

6. Sempre vá colocando na Wiki o que foi sendo feito em cada Sprint, utilizando Tabelas em Markdown
7. Sempre tela atualizado os **Projects**, organizando as Issues Finalizadas em **DONE**, se tiver trabalhando em uma Tarefa coloque a Issue em **IN PROGRESS** e se tiver criado uma Issue nova coloque em **TO DO**
8. Não deixe em **IN PROGRESS** uma tarefa que você parou de trabalhar, coloque somente nessa Coluna se estiver trabalhando nela.
9. Faça Dailys(reunião de equipe) pelo menos uma vez por semana para saberem o que deve ser feito e como anda o Desenvolvimento
10. Sempre estejam em contado um com outro durante o Desenvolvimento, se tiver algum impedimento entre em contato com seu Gerente para discutir formas de se resolver o Problema ou passar essa tarefa para outra pessoa.

## Resumo de Comandos GIT

AÇÃO	COMANDOS
Criar um Repositório Local e Vincular um Repositório Remoto e atualizar o Repositório Local e depois enviar novas modificações	1. git init 2. git remote add origin url 3. git pull origin master 4. git add . 5. git commit -m "message" 6. git push --set-upstream origin master
Atualizar seu Repositório Local	git pull
Enviar Modificações para o Repositório Remoto	git push origin nome-branch
Adicionar e Salvar Modificações em um Commit	1. git add . (salva tudo do Diretório) 2. git commit -m "message"
Reverter um Commit feito	1. git checkout id-commit(para ver para qual commit voltar) 2. git revert --no-edit id-commit (cria novo commit)
Fazer merge Localmente de duas Branchs	1. git checkout branch-destino(sair da Branch que iremos enviar modificações) 2. git merge branch-origem
Adicionando um novo url de outros Repositório Remoto	git remote add origin url
Verificando o nome e url de todos os Remotes do Repositório	git remote -v
Removendo um Remote do Repositório Local	git remote remove nome-remote
Verificando todos os Commits feitos na Branch	git log
Verificando todas as Branchs do Repositório Local	git branch
Verificando todas as Branchs do Repositório Remoto	git branch -r
Trocando de Branch Localmente	git checkout nome-branch
Criando nova Branch e acessando ela direto	git checkout -b nome-branch
Criando nova Branch sem acessar	git branch nome-branch