

# GIT & GITHUB

## Do Básico ao Avançado

Domine Git e GitHub com este material completo. Com ele você aprenderá a gerenciar fluxos de trabalho de maneira simples e dinâmica.



### Git branch

Neste material nós vamos aprender sobre uma das ferramentas mais famosas operações do Git e GitHub, que consegue criar **“universos alternativos”** de uma determinada linha do tempo.

Estamos nos referindo as Branchs, e elas atuam como ramificações do código principal para que o desenvolvedor consiga fazer alterações sem a necessidade de alterar a branch principal (**“Linha do Tempo Atual”**)

Para entendermos melhor como funciona as Branchs, eu vou contar a história do **“Strongest Man”** um super-herói que deixou a fama subir a sua cabeça, e tomou caminhos errados durante a vida.



#### Momento 1)

Tudo começou quando nosso querido herói trabalhava numa construtora, e por obra do acaso, acabou descobrindo sua enorme força descomunal, quando resolveu erguer árvores e equipamentos de escavação com suas próprias mãos.



#### Momento 2)

Certo dia, em sua volta para casa, ele acabou se deparando com uma cena chocante, assaltantes estavam invadindo um banco próximo à rua onde ele morava. E foi nesse momento que nosso herói decidiu intervir e conter os assaltantes.



### Momento 3)

Depois desse dia, nosso herói saiu da construtora por conta própria e decidiu combater o crime na cidade, assumindo o nome de "Strongest Man".



### Momento 4)

Anos depois, com muita fama, dinheiro e poder, ele começou a agir de forma mesquinha, a cobrar impostos da população, e até mesmo fazer parceria com criminosos para que ele os prendesse e soltasse logo em seguida, ou seja, tudo se tornou um grande teatro.



### Momento 5)

Algum tempo depois, algo tocou o coração de nosso herói "Strongest Man", e ele se deu conta daquilo que ele se transformou, e com muita vergonha de si mesmo decidiu desaparecer do mapa, e nunca mais foi visto.

## VIDA

Infelizmente, nosso herói foi tomado pelo orgulho e pela avareza, o que fez tomar caminhos errados durante a vida.

Mas, e se ele pudesse ter acesso à joia das branch do Enfermeiro Normal (Pegou a referência? rs), voltar no tempo e arrumar tudo?

Ou seja, e se ele pudesse voltar desde o "Momento 3" e fazer com que o "Momento 4 e 5" fossem completamente diferentes?

Ora, isso é totalmente possível com a **joia da Branch!**

Com ela em mãos, nosso herói decidiu voltar no tempo e fazer tudo diferente:



### Momento 4)

Anos depois, com muita fama, dinheiro e poder, ele começou a ser uma pessoa mais humilde e inteligente, doando parte do seu dinheiro para instituições de caridade, e ajudando os mais pobres a terem oportunidades na vida.



### Momento 5)

Algum tempo depois, ele se tornou o homem exemplo, um tipo de pessoa a ser seguida, e com isso, ele pode se aposentar e dar oportunidades aos outros heróis que estavam surgindo.

Note que cada momento que foi criado acima, no universo do GIT é considerado um commit, ou seja, uma foto de um determinado estado do projeto que está sendo controlado.

E a linha do tempo do nosso herói é considerado uma Branch, cujo significado é ramo ou braço.

É como se fosse uma linha do tempo paralela, independente da linha do tempo principal.



Entendendo mais  
sobre commits

Voltando agora ao universo do Git, quando a gente inicializa um novo repositório com ele, é automaticamente criada uma única linha do tempo, ou seja, uma branch, cujo nome sempre será "Master".

É nela que fica localizado todos os commits que iremos fazer no projeto, isto é, se não criarmos outra branch logo de cara.

A cada commit que fazemos na nossa branch master, é armazenado dentro do commit algumas informações importantes, como:

**COMMIT:** um hash único que identifica o commit.

**AUTHOR:** a pessoa que realizou o commit (nome e e-mail do usuário).

**DATE:** a data e hora do commit.

**MESSAGE:** uma mensagem que digitamos ao realizar o commit.

Já quando nós realizamos um segundo commit na nossa branch master, dentro desse novo commit uma nova informação é adicionada:

**PARENT:** faz referência ao commit anterior, dizendo que esse novo commit está relacionado com o antigo, ou seja, é realizado um parentesco com o primeiro commit.

E a lógica segue quando fazemos um terceiro commit, aonde este, terá um parentesco com o segundo commit, e assim por diante :D

E é dessa forma que o Git consegue rastrear as alterações dos nossos projetos!



## BRANCHS & MERGES

**A primeira coisa que você deve entender é:** quando você está trabalhando em um projeto com muitas pessoas, é sempre bom manter a branch master intacta.

**Sempre se pergunte:** Vou precisar fazer alterações?

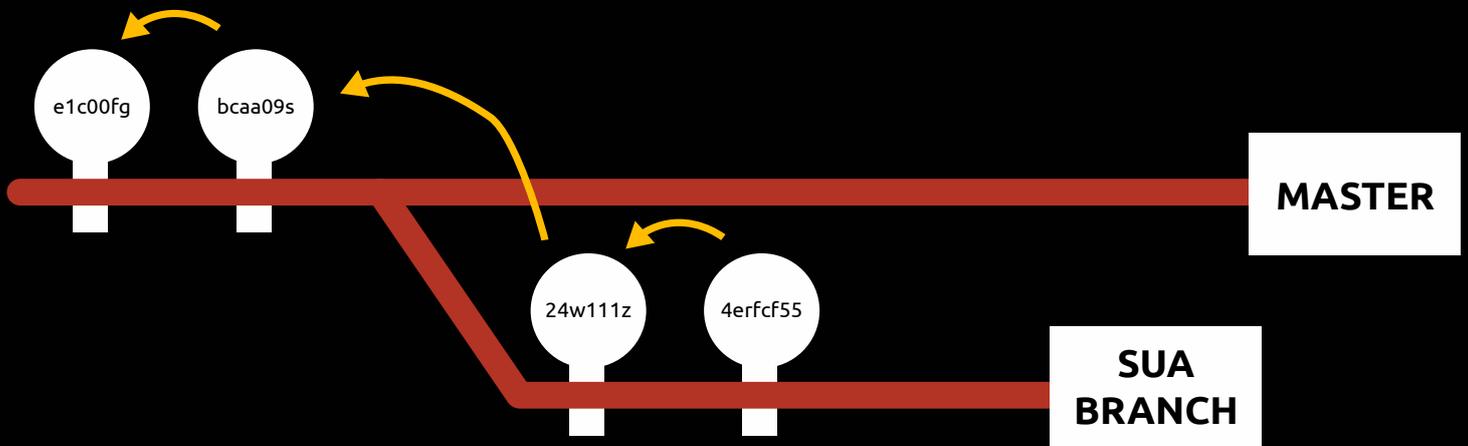
Então crie uma nova branch para você, e assim que terminar de desenvolver seu trabalho, sincronize suas alterações com a Branch Master.

Mas nunca trabalhe diretamente na **branch master**, pois se você errar alguma coisa ou fizer alguma besteira, pelo menos está fazendo isso na sua própria branch e não na master.

*“Tá, mas mesmo assim, eu consigo reverter a besteira que fiz na Branch Master recuperando commits anteriores, né?”*

Sim, você está certo, só que é muito mais fácil corrigir erros em novas branches (que possuem poucos commits), do que branches que possuem milhares de commits.

Outro fato interessante é que quando criamos uma nova branch, e fazemos commits dentro dela, cada commit terá ligação direta com a branch anterior, veja:



Para fazer com que os commits existentes na sua nova branch, vá parar na branch master, nós fazemos um Merge, que nada mais é do que a junção das suas alterações para a branch master.

**IMPORTANTE:** A branch master não fica bloqueada para novos commits depois que criamos novas branches, nesse caso, você ainda consegue fazer novos commits nela.

Existem três tipos de Merge:

**Merge Fast Forward:** Quando ninguém fez alteração na Branch Master. Esse tipo de Merge é mais tranquilo, fazendo com que o Git pegue seus commits e os anexe na branch master sem precisar que você (usuário) realize novos commits.

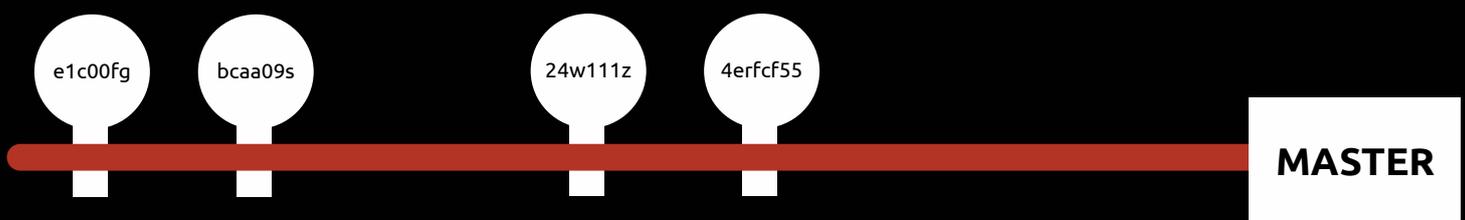
**Merge (Recursive Strategy):** Quando houver alterações na Branch Master, nesse meio tempo que você estava trabalhando na sua Branch. A Recursive Strategy acontece quando conseguimos fazer o merge sem a necessidade dos arquivos se sobreporem um ao outro, na maioria das vezes isso acontece quando os arquivos da branch master não são os mesmos que modifiquei na minha branch.

**Merge (Conflict):** Quando houver alterações na Branch Master nos mesmos arquivos que modifiquei na minha branch. Com isso o Git pede para resolvermos o conflito da seguinte forma:

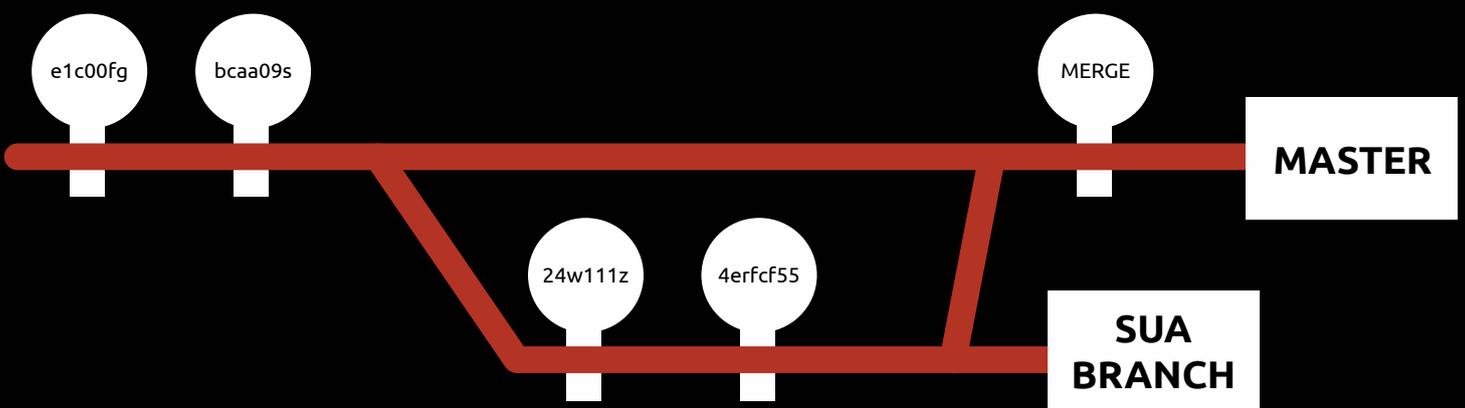
*“Olha Fulano, eu vi que esse arquivo que você está tentando fazer merge, também foi alterado recentemente nessa Branch... mais especificamente na linha 54, e então o que faço com essa linha? Eu removo a sua e fico com essa? Ou removo essa e fico com a sua? Ou ficamos com as duas?”*

Após fazer um dos três tipos de merge, você poderá tranquilamente remover a branch criada, ou seja, você não precisa mais da branch uma vez que os commits já foram anexados na master.

Exemplo quando fazemos um merge do tipo **Fast Forward**:



Exemplo do que acontece quando fazemos um merge do tipo **Recursive Strategy** ou **Conflict**:



Como podemos perceber nesse caso, o Git ele cria um único commit de merge que representa todos os Merges anteriores feitos na outra branch.



## Trabalhando com Branchs

```
git branch sua-branch
```

Para criar uma nova branch no Git, nós usamos este comando, onde no local onde está escrito 'sua-branch' você pode colocar o nome da branch que você deseja.

```
git branch
```

Para verificar se o git realmente conseguiu criar uma nova Branch nós executamos este comando.

Automaticamente o Git nos informará uma lista de todas as branchs existentes nesse repositório:

```
MINGW64: /c/Users/SeuUser/Desktop
```

```
SeuUser@DESKTOP-UHDBDBV MINGW64 ~/Desktop/Meu Projeto (master)
```

```
$ git branch
```

```
* master
```

```
* sua-branch
```

Tentei criar uma branch após o [git init] e recebi o erro *'fatal: not a valid object name: 'master' git branch'*.

É importante ressaltar que quando executamos o comando [git init] em uma pasta, não é criada uma branch chamada master.

Essa branch só será criada a partir do momento que fizermos nosso primeiro commit dentro dessa pasta.

Nesse caso, após o [git init], experimente fazer o processo de criação de um commit com um determinado arquivo: [git add .] [git commit -m 'primeiro commit'], e sem seguida crie sua branch [git branch sua-branch].

Repare que o Git nos informa por meio do **asterisco (\*)** a branch que estamos usando.

```
git checkout sua-branch
```

Para entrar na branch que acabamos de criar podemos usar o comando, onde no local onde está escrito 'sua-branch' você coloca o nome da branch que criou.

Dessa forma, todas as alterações que você realizar, a partir de agora estarão na sua nova branch.

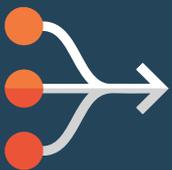
```
git checkout -b sua-branch
```

Para criar uma nova branch e entrar nela automaticamente usamos o comando.

Para voltar para a branch master, basta digitar **[git checkout master]**.

```
git branch -d sua-branch
```

Para remover uma branch já criada, usamos o comando.



## Realizando Merges

```
git merge sua-branch
```

Para fazer o Merge no Git nós usamos o comando ao lado, onde no local onde está escrito 'sua-branch' você deve colocar o nome da branch que você vai fazer o merge.

Esse comando deve ser usado dentro da branch que irá receber o Merge, que na maioria das vezes é a branch principal, que pode ser a **Master** ou alguma outra branch **superior a branch atual**.

Nesse caso, se certifique de que você está na branch master ou superior antes de fazer o **'git merge sua-branch'**, na branch que você vai fazer o merge.

END (y/n)



Merge (Fast Foward)

Para testarmos um Merge do tipo **Fast Foward**, primeiro vamos criar uma pasta chamada **'Projeto'** dentro do meu disco local:



*Projeto*

Com o **Git Bash** aberto dentro dessa pasta, executaremos o comando para inicializar nosso repositório:

```
git init
```

Em seguida nós iremos criar um arquivo de texto chamado de **'arquivo\_1.txt'**, dentro da pasta **'Projeto'** com o seguinte conteúdo:

```
1 Primeiro Arquivo Criado!
```

Após isso, com o **git bash** aberto, nós iremos fazer um novo commit:

```
git add .  
git commit -m 'commit 1'
```

Agora nós iremos criar uma nova branch para testarmos o Merge do tipo Fast Forward, no meu caso criei uma **nova branch** chamada de 'sua-branch':

```
git branch sua-branch
```

Vamos entrar nela:

```
git checkout sua-branch
```

E criar um arquivo de texto chamado '**arquivo\_2.txt**' com o seguinte conteúdo:

```
1 Segundo Arquivo Criado!
```

Após isso, com o **git bash** aberto, nós iremos fazer um novo commit:

```
git add .  
git commit -m 'commit 2'
```

Se você resolver voltar para a branch principal usando o comando **git checkout master**, você vai perceber que o **arquivo\_2.txt** sumiu, isso aconteceu, pois ele só existe dentro da branch '**sua-branch**'.

**Para fazer o Merge do tipo Fast Forward, temos que nos certificar que estamos dentro da Branch Master**, nesse caso, execute o comando:

```
git checkout master
```

E de que não fizemos nenhum commit na branch master, ok?

Com isso em mente, é só executarmos o comando:

```
git merge sua-branch
```

Esse comando vai fazer um **merge da sua-branch** com a branch que você está (que no caso é a **master**).

Com isso o **arquivo\_2.txt**, agora estará presente na branch master. E isso foi feito automaticamente como se nunca tivesse existido uma nova branch.

```
MINGW64:/c/Projeto
SeuUser@DESKTOP-UHDBDBV MINGW64 C:/Projeto (master)
$ git merge sua-branch
... Fast-Foward ...
```

Com o comando **[git log -graph]**, veremos que foi realizado o commit 2, e que o git não considera que ele veio de uma nova branch, uma vez que o processo de merge foi do tipo Fast Forward.

Por fim, basta fazer a **exclusão da branch** que criamos:

```
git branch -d sua-branch
```



**Merge (Recursive Strategy)**

Para começar, vamos manter os testes que fizemos anteriormente, e criar uma nova branch chamada de **'segunda-branch'**:

```
git checkout -b segunda-branch
```

E criar um arquivo de texto chamado **'arquivo\_3.txt'** com o seguinte conteúdo:

```
1 Terceiro Arquivo Criado!
```

Após isso, com o **git bash** aberto, nós iremos fazer um novo commit:

```
git add .  
git commit -m 'commit 3'
```

Agora vamos voltar para a nossa **branch master**:

```
git checkout master
```

E vamos criar um arquivo de texto chamado de **arquivo\_4.txt**, com seguinte conteúdo:

```
1 Quarto Arquivo Criado!
```

Após isso, com o **git bash** aberto, nós iremos fazer um novo commit:

```
git add .  
git commit -m 'commit 4'
```

Por fim, vamos um **merge** com a nossa branch que criamos (isso dentro da branch principal).

```
git merge segunda-branch
```

Observe que a mensagem que o Git nos dá, é um pouco diferente, aonde é mostrado a mensagem **Recursive Strategy**:

```
MINGW64: /c/Projeto
```

```
SeuUser@DESKTOP-UHDBDBV MINGW64 C:/Projeto (master)
```

```
$ git merge segunda-branch
```

```
... Recursive Strategy ...
```

Se realizarmos o `git log` ou `git log --graph`, veremos todos os 4 commits, e um novo, que representa o merge criado pelo **recursive strategy** (*Merge Branch segunda-branch*).

Como deu certo o **merge**, podemos agora excluir a branch que criamos:

```
git branch -d segunda-branch
```



**Merge (Conflict)**

Para começar, vamos manter os testes que fizemos anteriormente, e criar uma nova branch chamada de **'terceira-branch'**:

```
git checkout -b terceira-branch
```

Em seguida, dentro dessa branch vamos fazer algumas alterações no **arquivo\_4.txt**, como por exemplo:

```
1 BLABLA_Quarto Arquivo Criado! - @WWE
2 ERGERGER
3 ERGER345345
```

Em seguida salve este arquivo e faça o mesmo processo do commit:

```
git add .  
git commit -m 'arquivo 4'
```

Agora vamos voltar para a **branch master**, e fazer alterações no **arquivo\_4.txt**, olha as alterações que fiz:

```
1 wufhe9iurhfeuwhfr  
2 34523452345  
3 ghr
```

Em seguida salve este arquivo e faça o mesmo processo do commit:

```
git add .  
git commit -m 'arquivo 4'
```

Agora nos temos uma situação, em que na hora que falarmos para o Git fazer o Merge, ele não terá como saber quais alterações nas linhas devem continuar e quais devem ser removidas, ou quem sabe, se é para manter as duas ou não.

Para testar isso, ainda na branch master, execute o seguinte comando:

```
git merge terceira-branch
```

Um novo conflito será mostrado pelo git, indicando o nome do arquivo:

```
MINGW64: /c/Projeto
```

```
SeuUser@DESKTOP-UHDBDBV MINGW64 C:/Projeto (master|MERGING)
```

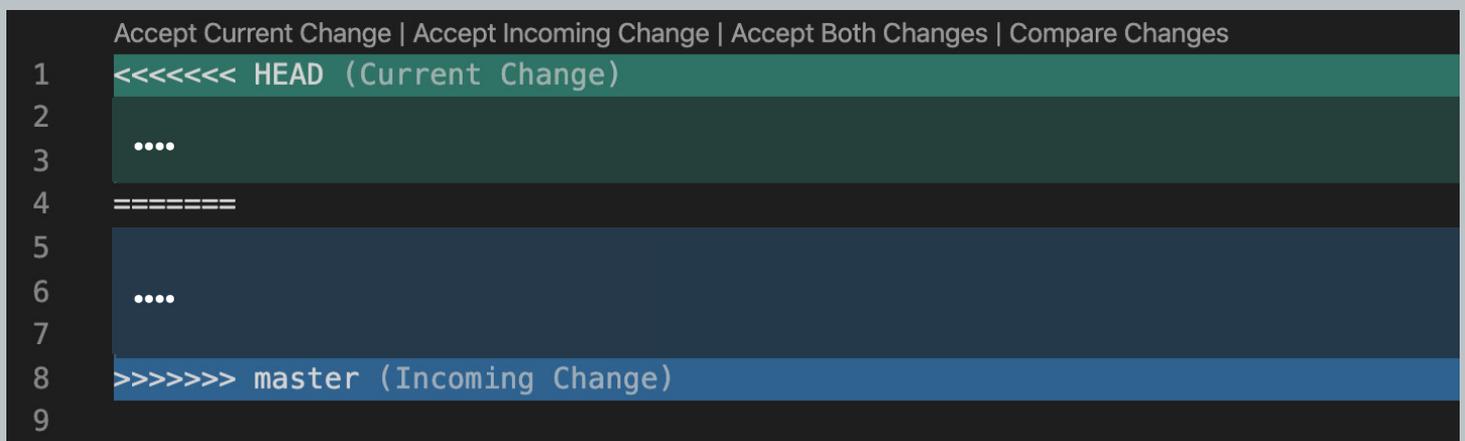
```
$ git merge terceira-branch
```

```
CONFLICT (content): .... arquivo4.txt ....
```

Mensagens de conflito acabaram de aparecer na tela, e o nome da branch ficou como **(master|Merging)** o que indica que estamos parados no processo de merge, e só iremos continuar quando resolvermos todos os conflitos!

Se executarmos o **git status**, veremos que estamos em uma situação de Merging que esta pendente para o arquivo **arquivo\_4.txt**.

Se você estiver usando o **Visual Studio Code**, e abrir o **arquivo\_4.txt**, você verá uma interface gráfica organizada que nos mostra o local onde está o conflito.



```
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
1 <<<<<<< HEAD (Current Change)
2
3 ....
4 =====
5
6 ....
7
8 >>>>>> master (Incoming Change)
9
```

Com algumas opções em inglês, como:

**Accept Current Change:** Aqui você aceita o bloco que está nomeado como 'current', quando clicamos nele, o bloco 'incoming change' é deletado.

**Accept Incomming Change:** Aqui você aceita o bloco que está nomeado como 'incoming change', quando clicamos nele, o bloco 'current' é deletado.

**Accept Both Changes:** Como a própria tradução já diz, ele aceita ambas as alterações e nenhum dos dois blocos é apagado.

**Compare Changes:** Compara as diferenças entre os dois blocos.

Lembrando que se você tiver mais conflitos em outros arquivos, você também precisa resolver.

No meu caso, eu cliquei no botão **'Accept Both Changes'**.

Para finalizar o **Merge**, precisamos fazer o commit final manualmente:

```
git add .  
git commit -m 'Merge terceira-branch'
```

A partir de agora, o nome de **'merging'** sumiu do lado do nome da branch master, e se verificamos com o git log, veremos que tudo ocorreu bem :D

Nesse caso, percebemos que no **Merge Conflict**, além de resolver os conflitos, ainda precisamos realizar um novo commit.

Diferente dos Merges do tipo **Fast Forward** e **Recursive Strategy**, que já criam commits de forma automática.

END (y/n)



## Branchs Remotas

Vamos supor que você queria **sincronizar sua branch local com seu repositório GitHub**? Como você faz isso?

Para testarmos a nova funcionalidade que iremos aprender, certifique-se de que você criou um novo repositório no GitHub, realizou um clone para a sua máquina local, e criou uma nova branch.

Após a adição de um novo arquivo, e realização de um commit, basta apenas que chamemos o comando **[git push]**, só que se fizermos isso, o git nos retornará um problema, isso aconteceu pois o push está configurado por padrão a aceitar somente envios da nossa branch master.

```
git push origin sua-branch
```

Com este comando somos capazes de alterar o origin, fazendo com que seja possível realizar o push da branch que você criou para o repositório do GitHub.

Supondo agora que você está trabalhando em equipe, e alguém da sua equipe te disse assim: "*Olha fulano, eu acabei de criar uma branch e quero que você trabalhe nela, o nome da branch é optus-maker*".

Como você já sabe, é so executarmos o comando **[git checkout optus-maker]** que entramos nessa nova branch, só que se fizermos isso, o Git nos dirá que essa branch não existe!

```
git fetch
```

Com este comando dizemos ao git para sincronizar todas as atualizações remotas. (E isso inclui trazer as branchs)

```
git branch -v -a
```

Com este comando somos capazes de alterar o origin, fazendo com que seja possível realizar o push da branch que você criou para o repositório do GitHub.

Por fim, é só executarmos o comando **[git checkout optus-maker]**, e pronto, a branch que criaram para você está pronta para uso!

## Gostou desse material?



Então não deixe de acessar a nossa seção de [Git & GitHub do básico ao avançado](#).

ACESSAR



MICILINI.COM

⟨Seu Portal de CODE⟩