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 Start?

Primeiros Passos

Antes de colocarmos a mão na massa e começarmos a versionar nossos arquivos, em primeiro lugar, precisamos preparar o nosso ambiente de desenvolvimento.

E isso inclui, primeiro, criar uma pasta em nosso computador aonde iremos simular a utilização do Git, e segundo, configurar o usuário que será usado pelo Git.

Vamos começar?

Criando a Pasta do Projeto de Testes

Em algum lugar do seu computador, pode ser na sua área de trabalho, no seu disco local, nos seus documentos, ou...

algum outro lugar que seja de fácil acesso para você.

Crie uma pasta chamada "Meu Projeto", você também pode nomeá-la para o nome que você quiser, ok?

A pasta que acabamos de criar será usada posteriormente para simularmos uma situação, em que iremos usar o Git para versionarmos os arquivos existentes dentro dela.

Criou a pasta? Legal, então já podemos avançar para a próxima etapa!

Iniciando o Git

Toda vez que for versionar alguma pasta onde armazena um projeto (estou partindo do pressuposto que você é organizado e separa seus arquivos por pastas, ok?), você é DEVE abrir um terminal do Git dentro dessa pasta, e executar um comando chamado:

git init

E é isso que vamos fazer agora 

Na da pasta que você acabou de criar, abra o terminal do Git, digite o comando git init, e dê [ENTER]:

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

SeuUser@DESKTOP-UHDBDBV MINGW64 ~/Desktop/Meu Projeto (master)
$ git init
Initialized empty Git repository in C:/Users/SeuUser/Desktop/Meu Projeto/.git/

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

(No meu caso, estou usando o terminal do próprio Git, portanto tive que escolher a opção "Git Bash Here" dentro da pasta "Meu Projeto")

O comando git init, inicializa o ambiente do Git na da pasta do projeto, para que futuramente possamos trabalhar com ele. Mas como ele faz isso?

Simples, ele cria uma pasta invisível chamada .git (talvez você tenha que editar as configurações do seu sistema operacional para visualizar essas pasta invisíveis).

Essa pasta irá armazenar todos os backups, todas as versões mais antigas do nosso projeto, ou seja, os arquivos que foram versionados estarão todos lá dentro.

Para executar o git init, a pasta precisa estar vazia?

Não, uma vez que você pode executar o git init mesmo quando existir milhares de arquivos dentro do seu projeto.

Pode ficar tranquilo que isso não vai influenciar em nada, ok? Dê o git init e seja feliz =)

Configurando o Usuário

Sempre quando você instala o Git em uma máquina nova, você PRECISA configurar um novo usuário, ou quem sabe um grupo de usuários que trabalharão em um determinado projeto. *(no nosso caso, dentro da pasta aonde se localiza o projeto rs)*

O comando responsável pela configuração de um novo usuário, é o git config, mas sozinho ele não faz milagre rs

Este comando ele conta com 3 tipos de parâmetros iniciais:

```
git config --global
```

O parâmetro --global, é responsável por fazer configurações globais, ou seja, tudo o que você criar, modificar ou excluir usando esse parâmetro, surtirá efeito em todos os usuários existentes no seu sistema operacional.

Por exemplo, se você altera o editor de código usado pelo Git usando esse parâmetro, quando uma outra conta de usuário existente no seu computador for usar o Git, ele vai dizer assim... "*ué, porque o Git agora está abrindo esse editor de códigos... não era um outro...*"

```
git config --local
```

O parâmetro --local, é responsável por fazer configurações somente locais, ou seja, tudo o que você criar, modificar ou excluir usando esse parâmetro, surtirá efeito SOMENTE na pasta do projeto na qual você deu o git init.

O prós, é que você não precisa ficar alterando configurações globais, o que depende muitas vezes da permissão de outras pessoas. O contra é que talvez você precise sempre ficar fazendo essas configurações iniciais, sempre que criar um novo projeto :S

Continuando...

No nosso caso, nos iremos usar a flag --global, pois dessa forma eu não preciso ficar sempre criando configurações diferentes a cada projeto que eu criar.

Maaaaaas, sozinho esse comando não faz absolutamente nada, e é agora que iremos usar um outro parâmetro (mais especificamente ao lado do --global), onde dirá ao Git o nome de usuário que eu estou criando.

Ele se chama user.name e após digitá-lo, precisamos dar um espaço para digitar o nome do usuário, que no caso é o nome do seu usuário do GitHub, ok?

```
git config --global user.name micilini
```

Tá, e se eu resolver colocar um nome diferente?

Vai dar na mesma rs O principal motivo de colocar o mesmo username do seu GitHub é para manter a consistência dos usuários que estão versionando aquele projeto.

Após criar um nome de usuário, ainda precisamos configurar o e-mail dele, e para isso usamos o user.email: (use o mesmo do seu GitHub, ok?)

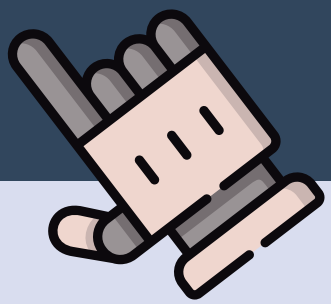
```
git config --global user.email micilini@micilini.com
```

E pronto, já temos o nosso usuário configurado =)

Caso você queira consultar as modificações que acabará de fazer, basta usar o comando git config --global -l, lembrando que o parâmetro -l vem de listing (cuja tradução é listagem), e nesse caso ele lista as configurações globais (se quiser as locais use git config --local -l):

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

SeuUser@DESKTOP-UHDBDBV MINGW64 ~/Desktop/Meu Projeto (master)
$ git config --global -l
user.email=micilini@gmail.com
user.name=micilini
```

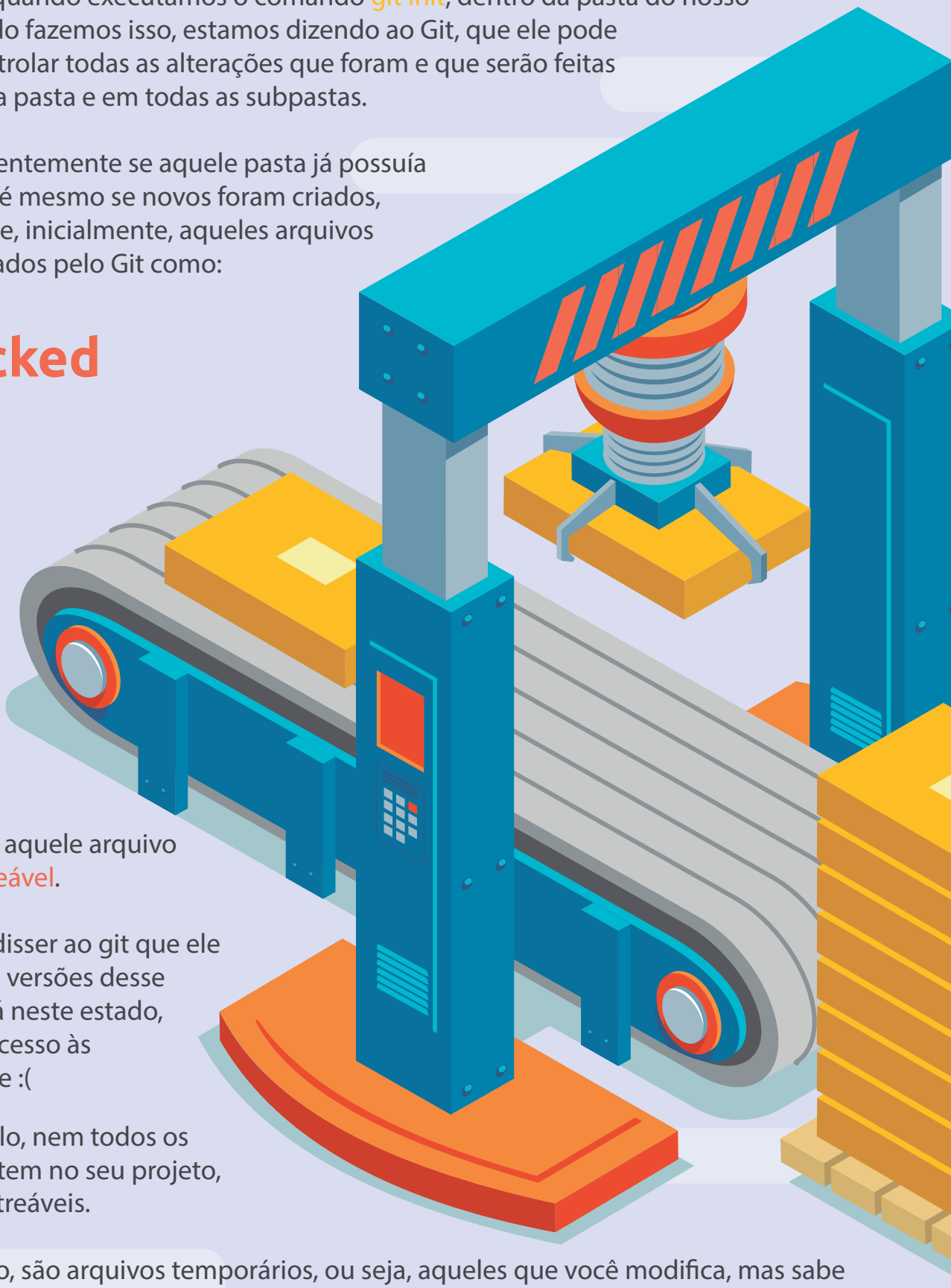


Como Funciona o Fluxo de Trabalho no Git?

Tudo começa quando executamos o comando git init, dentro da pasta do nosso projeto. Quando fazemos isso, estamos dizendo ao Git, que ele pode começar a controlar todas as alterações que foram e que serão feitas dentro daquela pasta e em todas as subpastas.

Mas independentemente se aquele pasta já possuía arquivos, ou até mesmo se novos foram criados, a verdade é que, inicialmente, aqueles arquivos serão identificados pelo Git como:

UnTracked



Isso significa que aquele arquivo ainda não é rastreável.

E caso você não disser ao git que ele deve controlar as versões desse arquivo, ele ficará neste estado, e você não terá acesso às modificações dele :(

Mas fique tranquilo, nem todos os arquivos que existem no seu projeto, precisarão ser rastreáveis.

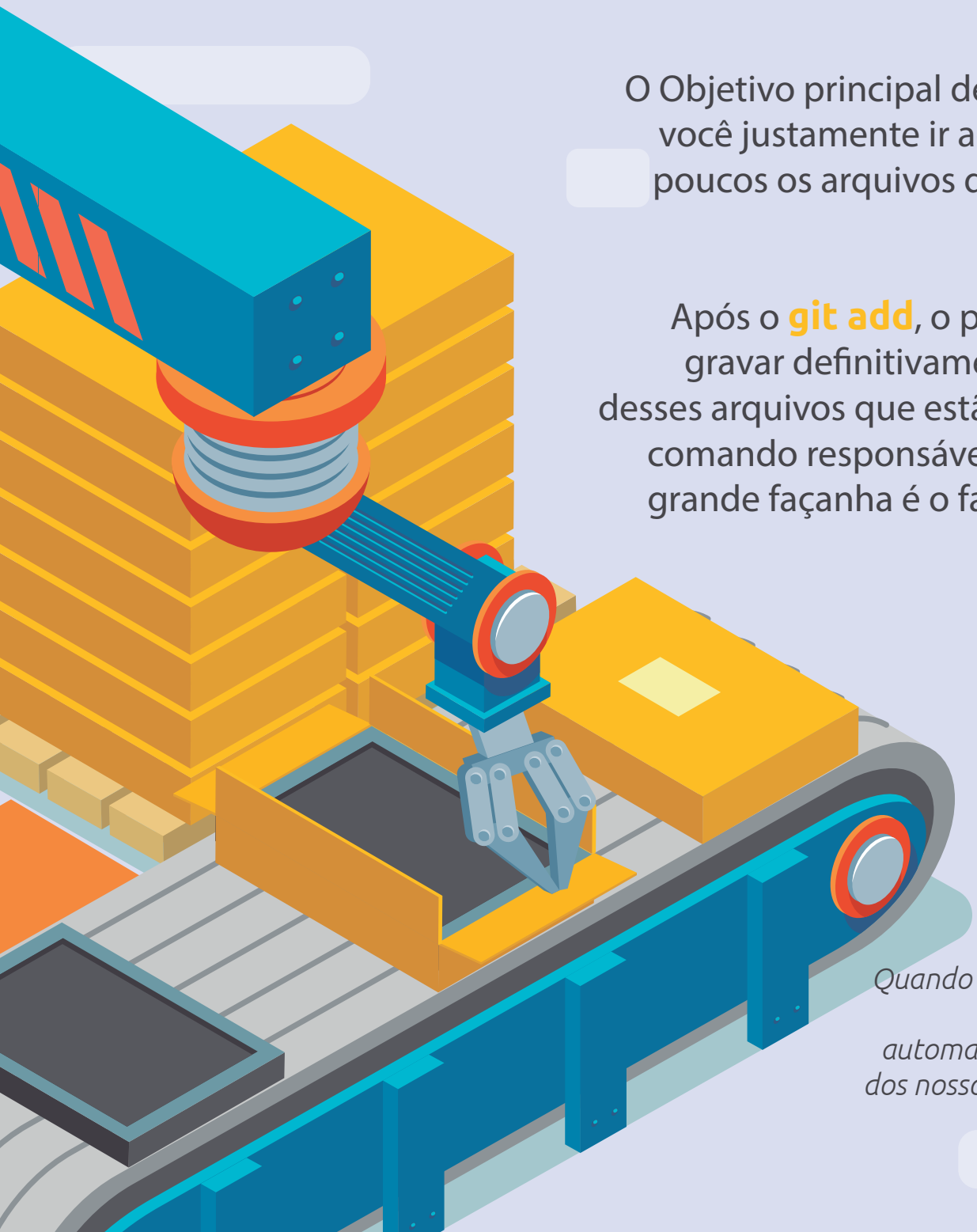
Um exemplo disso, são arquivos temporários, ou seja, aqueles que você modifica, mas sabe que vai apagar depois, ou que terá uma atualização futura o esperando...

Agora para fazermos com que o Git comece a controlar a versão dos arquivos, nós usamos o famoso comando **git add** que faz com que nosso arquivo, ou nossos arquivos (caso tenhamos selecionado mais de um), seja enviado para um tal local chamado de STAGE AREA.

O STAGE AREA, para o Git nada mais é do que uma área de transferência. Nesse caso seu arquivo ainda não foi versionado (histórico controlado), mas que ainda vai ser.

Ou seja, seu arquivo está na CARA DO GOL!

Sabe a memória cache do seu computador, ou quando você copia um texto no computador, mas não o salva em um arquivo? Então... o STAGE AREA funciona exatamente assim...



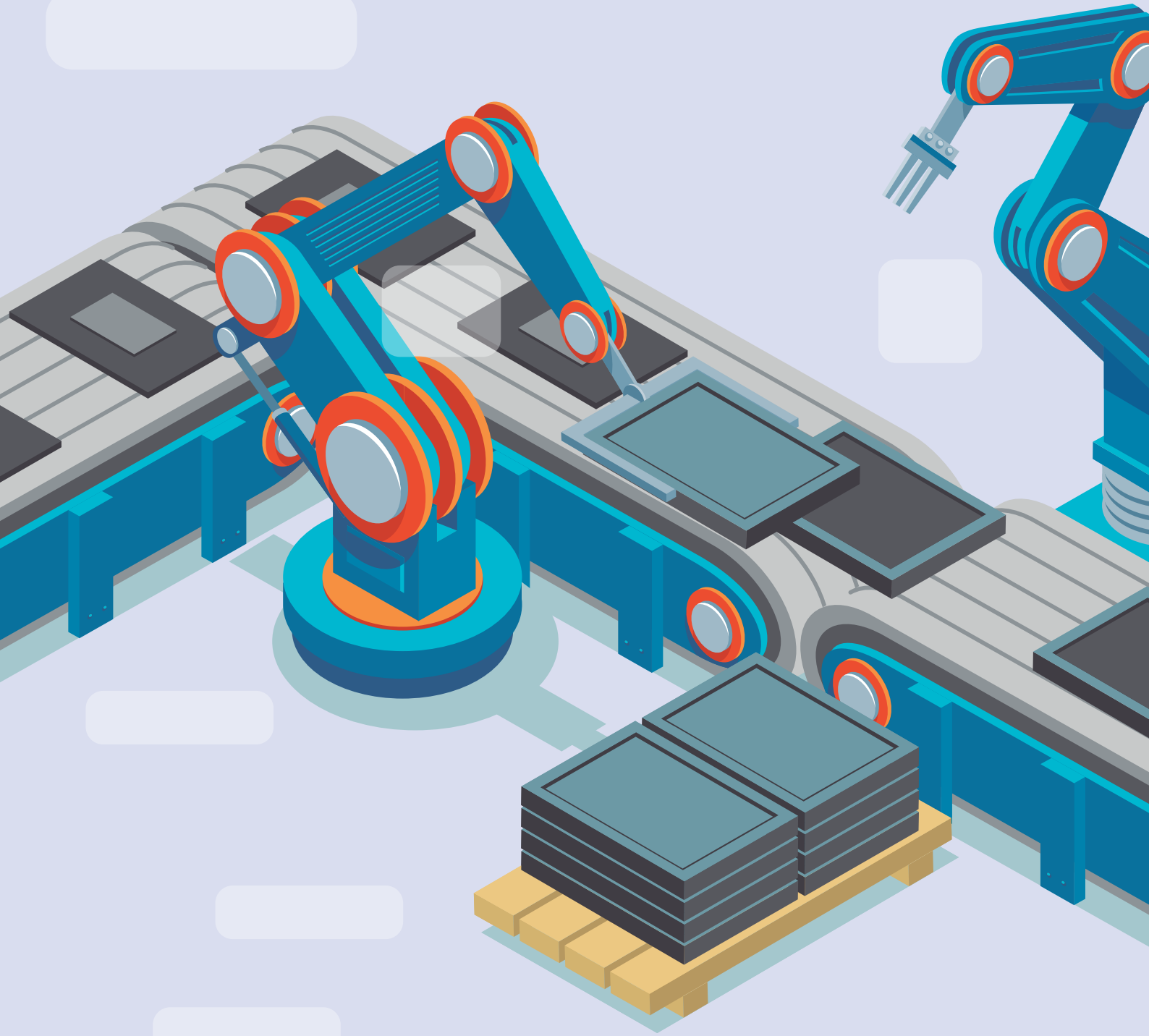
O Objetivo principal desta área é para você justamente ir adicionando aos poucos os arquivos que você deseja versionar ;)

Após o **git add**, o próximo passo é gravar definitivamente o histórico desses arquivos que estão na Stage, e o comando responsável por fazer essa grande façanha é o famoso **commit**

Quando usamos o comando git add automaticamente, o status dos nossos arquivos saem de untracked para unmodified.

O **commit** é o comando responsável por "tirar um print" dos arquivos que estão no STAGE AREA, e salvá-los na da pasta *.git*, para que futuramente eles possam ser acessados.

É nesse ponto que acontece o **VERSIONAMENTO!**



Toda vez que um **commit** é feito, um código sequencial, chamado de hashcode, é dado para cada arquivo "commitado". Com esse hash, podemos acessar as versões mais antigas daquele arquivo.

É importante ressaltar que toda vez que você altera um arquivo existente no stage area, seu status muda para modified.

Recapitulando o Fluxo de Trabalho do Git

Acredito que a partir de agora você já tenha mais ou menos uma ideia do que fazer para versionar seus códigos, certo?

Mas antes de passarmos para a próxima etapa, vamos fazer uma recapitulação rápida, ok?

```
git init
```

Diz ao Git que ele pode iniciar o controle de versão dos arquivos, pastas e subpastas existente no projeto.

- Adiciona uma pasta invisível .git
- Todos os arquivos da pasta continuam como não rastreável (untracked)

Diz ao Git para jogar seu(s) arquivo(s) para a área de transferência, preparando-o(s) para o backup.

```
git add
```

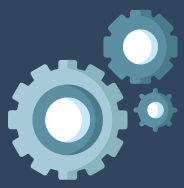
- Leva os arquivos para a famosa STAGE AREA
- O status de untracked se altera para unmodified

```
git commit
```

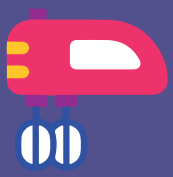
Diz ao Git que aqueles arquivos já podem sair do STAGE AREA e ir direto para a pasta .git aonde acontece o backup definitivo.

- um hashcode é adicionado a cada arquivo salvo
- O status de unmodified se altera para modified sempre quando alteramos um arquivo já versionado
- Basicamente o commit "tira um print" do arquivo para salvá-lo em backup.-

```
END (y/n)
```



Praticando o add, commit e status



Agora que você já tem uma noção do que deve ser feito, acredito que você esteja louco para colocar a mão na massa... digo.... começar a versionar o seu projeto, certo?

Pois, bem! Para começarmos a versionar os arquivos do nosso projeto, primeiro precisamos, é claro, ter arquivos dentro daquela pasta que nos criamos passos atrás, ainda se lembra?

Meu Projeto

Exemplo.txt

```
1  Olá Planeta Terra!
```

Como podemos ver, foi criado um arquivo de texto *(exemplo.txt)* dentro da nossa pasta, cujo conteúdo é *"Olá Planeta Terra!"*.

Recomendo que você faça o mesmo ;)

O próximo passo, é abrir o terminal do git para iniciarmos o processo de versionamento!

Mas tenha em mente, que o Git ainda não controla a versão do arquivo de texto que nós criamos, e nem precisamos fazer isso se quisermos.

Mas como nós queremos, até porque estamos aprendendo, né, nós iremos dar prosseguimento aos códigos do Giiit :D

```
git status
```

O primeiro comando que iremos usar é o git status, com ele nos veremos através do Git, o status de todos os arquivos existente dentro da nossa pasta.

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

SeuUser@DESKTOP-UHDBDBV MINGW64 ~/Desktop/Meu Projeto (master)
$ git status
On branch master

no commits yet

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

nothing added to commit but untracked files present use "git add" to track)
```

Como podemos observar, ele reconheceu o arquivo que nós criamos (*exemplo.txt*), mas que ele ainda está marcado como não rastreável (Untracked).

E como vimos anteriormente, se quisermos passar a rastrear esse arquivo, primeiro nós precisamos colocá-lo no Stage Area, que nada mais é do que a nossa área de transferência do Git.

```
git add exemplo.txt
```

Ainda com o terminal aberto, o segundo passo é executarmos o comando git add [NOME DO ARQUIVO], que no nosso caso, foi colocado o nome do arquivo que queremos versionar, que é exemplo.txt

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

SeuUser@DESKTOP-UHDBDBV MINGW64 ~/Desktop/Meu Projeto (master)
$ git add exemplo.txt
```

Legal, o nosso arquivo exemplo.txt, acaba de ser enviado a área de transferência, tanto é, que se executarmos novamente o comando git status, veremos que o status deste arquivo vai mudar:

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

SeuUser@DESKTOP-UHDBDBV MINGW64 ~/Desktop/Meu Projeto (master)
$ git status
On branch master

No commits yet

Changes to be committed:
     (use "git rm --cached <file>..." to unstage)
           new file:     exemplo.txt
```

Como podemos ver uma nova mensagem dizendo: "alterações para passar pelo commit" apareceu, e isso significa que existem arquivos para ser analisados antes de realizarmos o versionamento.

Observe que os arquivos estão na cor verde, e isso diz a respeito que eles estão prontos para serem versionados.

Existe uma forma de adicionar todos os arquivos? Ao invés de ficar fazendo arquivo por arquivo? Um por um?

Sim, realmente ficar fazendo um por um é uma tarefa tediosa, e uma forma de contornar isso é usando o comando git add .

Com o . (Ponto), nos enviamos para a área de transferência, todos os arquivos, pastas e subpastas existente na raiz da nossa pasta.

```
git add .
```

E se eu quiser enviar uma pasta, ou quem sabe uma subpasta, ou talvez um arquivo existente dentro dessa pasta, como faço?

```
git add nomedodiretorio
```

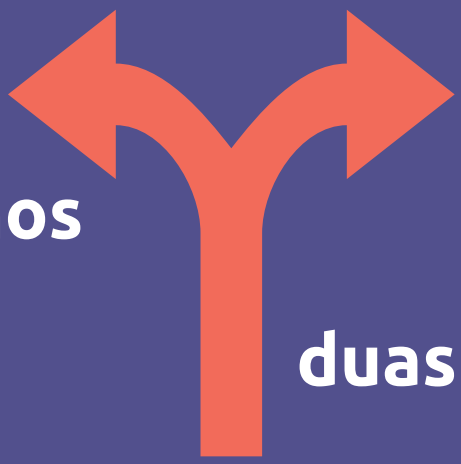
Use git add [nome da pasta], para adicionar uma pasta a área de transferência.
(Os arquivos existentes dentro da pasta também serão adicionados)

```
git add nomedodiretorio/nomedosubdiretorio
```

Use git add [nome da pasta]/[nome da sub pasta], para adicionar uma subpasta a área de transferência.

Agora para adicionar um arquivo existente dentro de uma pasta, só fazer como o exemplo do lado ;)

```
git add nomedodiretorio/file.txt
```



Agora nos temos duas opções a seguir

#1: Nós podemos continuar criando e editando nossos arquivos, no STAGE AREA....

#2: Ou, podemos dar a palavra final e partir para o GIT COMMIT

```
git commit -m “adição do arquivo exemplo.txt”
```

Como sabemos, o commit, é responsável por pegar todos os arquivos da área de transferência e versioná-los.

O comando -m é opcional, e ele cria uma mensagem, que nos ajuda a entender o que foi feito nesse versionamento.

Tente ser sempre beeeem claro na mensagem que você passa ali, pois futuramente poderá servir tanto para você quanto para outras pessoas que estão acessando esse projeto.

Será que é possível, fazer commit de só um arquivo, em vez de todos?

Sim, isso é totalmente possível, veja como é fácil...

```
git commit -m 'adição do exemplo.txt' exemplo.txt
```

Agora, caso esse arquivo esteja dentro de uma pasta:

```
git commit -m 'adição do exemplo.txt' minhaspasta/exemplo.txt
```

Agora, caso você queria fazer o commit de um diretório inteiro:

```
git commit -m 'adição da minha pasta' minhaspasta
```

Caso, você não queria adicionar uma mensagem no commit:

```
git commit exemplo.txt
```

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

SeuUser@DESKTOP-UHDBDBV MINGW64 ~/Desktop/Meu Projeto (master)
$ git commit -m “adição do exemplo.txt”

[master (root-commit] 8d77fa4] adição do exemplo.txt
  1 file changed, 1 insertion(+)
  create mode 100644 exemplo.txt
```


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

SeuUser@DESKTOP-UHDBDBV MINGW64 ~/Desktop/Meu Projeto (master)
$ git status

On branch master
nothing to commit, working tree clean
```

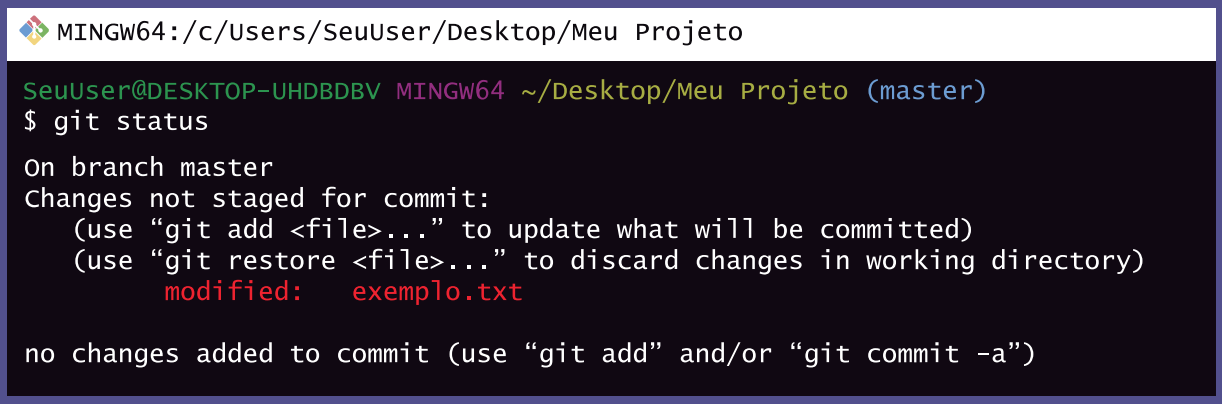
Veja o que aparece, quando você executa o comando git status, após fazer o commit de todos os arquivos.

Mas e se eu alterar esse arquivo?

Bem, supondo que após o commit, você alterou algumas linhas do exemplo.txt, e salvou.

1	Olá Planeta Terra!
2	Olá pra você tambeem Planeta Marte!

Se você executar o comando git status, você verá uma mensagem beem parecida com essa:



Como era de se esperar, o git nos informou quem o exemplo.txt, mudou seu status para modificado. Ou seja, houveram alterações em um dos arquivos na qual o Git estava rastreando.

Caso fechássemos este terminal, e parássemos de “trabalhar” naquele projeto, o Git não faria o commit automaticamente.

Por isso, precisaríamos fazer o git add e o git commit novamente, ok?

```
git add .
git commit -m ‘alteração na segunda linha do exemplo.txt’
```

Só que ficar fazendo esse processo para cada alteração, é uma coisa meio chata e muuuito redundante, e dai você pode se perguntar, será que existe algum meio do GIT fazer esse processo automaticamente para os arquivos que já estão sendo rastreáveis?

Ou seja, para aqueles arquivos que já foram comitados um dia?

```
git commit -am 'ultimas alterações'
```

Com este comando, estamos mesclando o add (de adicionar), com o parâmetro -m (de mensagem), formando o parâmetro -am.

Neste caso, não precisaríamos executar novamente o git add.

Mas lembre-se: *ESSE COMANDO SÓ FUNCIONA PARA ARQUIVOS RASTREÁVEIS, ENTÃO SE VOCÊ ADICIONOU UM NOVO ARQUIVO NA SUA PASTA, O GIT NÃO FARÁ O COMMIT DELE.*

Visualizando as versões anteriores

```
git log
```

Para visualizarmos o histórico dos arquivos que foram salvos, ou seja, ver basicamente o histórico de quem editou, quando editou e quais arquivos foram editados, nós podemos usar o comando git log.

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

SeuUser@DESKTOP-UHDBDBV MINGW64 ~/Desktop/Meu Projeto (master)
$ git log

commit dc43in834j0lkskm948hfdmnj20430dkkfk0499834d (HEAD -> master)
Author: Micilini <micilini@gmail.com>
Date: Thu Mar 18 21:18:33 2023 -0300

     ultimas changes

commit jjsu728ushbe98729843rbn90k9k29s92jj3rfh83d2
Author: Micilini <micilini@gmail.com>
Date: Thu Mar 18 20:32:12 2023 -0300

     adição do exemplo.txt
```

Como podemos observar a primeira coisa que ele mostra para gente é o hascode do commit, junto com o autor (responsável pelo versionamento), seu e-mail, a data em que isso foi feito, e a mensagem.

Esse histórico aparece o commit mais recente, para o mais antigo.

Para vermos como o nosso arquivo (exemplo.txt) se encontrava antes de realizarmos o segundo commit, é bem fácil. Então supondo que você queira rastrear como estava o arquivo no momento que você fez o primeiro commit, ou seja, visualizar a primeira versão do arquivo, basta usar o git log:

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

SeuUser@DESKTOP-UHDBDBV MINGW64 ~/Desktop/Meu Projeto (master)
$ git log
commit dc43in834j0lkskm948hfdmnj20430dkkfk0499834d (HEAD -> master)
Author: Micilini <micilini@gmail.com>
Date: Thu Mar 18 21:18:33 2023 -0300

     ultimas changes

commit jjsu728ushbe98729843rbn90k9k29s92jj3rfh83d2
Author: Micilini <micilini@gmail.com>
Date: Thu Mar 18 20:32:12 2023 -0300

     adição do exemplo.txt
```

E selecionar o hashcode do primeiro commit, ou seja, aquele que está lá no final da fila.

(jjsu728ushbe98729843rbn90k9k29s92jj3rfh83d2)

E com o hashcode em mãos, basta executar o comando git checkout informado o hashcode:

```
git checkout hashcodeaqui
```

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

SeuUser@DESKTOP-UHDBDBV MINGW64 ~/Desktop/Meu Projeto ((jjsu72...))
$ git checkout jjsu728ushbe98729843rbn90k9k29s92jj3rfh83d2
Note: switching to 'jjsu728ushbe98729843rbn90k9k29s92jj3rfh83d2'

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commit you make in this
state without impacting any branches by switching back to a branch.

.....
```

Estou preso no git log!!!

Caso o seu log não tenha terminando na mesma tela, use a tecla Q para sair do log, quando houver: (dois pontos)

A partir desse momento o Git te diz que você não está mais apontando para o fim dos seus commits, mas sim para a hashcode, ou seja, exatamente para o arquivo que estava no histórico.

```
MINGW64:/c/Users/SeuUser/Desktop
SeuUser@DESKTOP-UHDBDBV MINGW64 ~/Desktop/Meu Projeto ((jjsu72...))
```

Observe que não estamos mais no (master), ou seja, no último commit realizado!

Se você abrir o exemplo.txt no seu editor de códigos favorito, verá que o arquivo mudou, e voltou a aparecer a versão mais antiga, incrível não?

```
1  Olá Planeta Terra!
```

Mas calmaaaa, que você não perdeu as últimas alterações que você fez, ok?

Para visualizarmos outros commits, nos precisamos voltar para o estado atual dos nossos commits, e para isso, com o terminal aberto basta digitarmos:

```
git checkout master
```

Agora é só digitar git log novamente e realizar o mesmo processo para os outros arquivos.

E o mais interessante é que o arquivo exemplo.txt volta ao normal, incrível não acha?

Recapitulando log e checkout do git

E para não perder o costume, que tal recapitular tudo isso que acabamos de aprender?

```
git log
```

Diz ao Git para ele mostrar o histórico de todos os commits já feitos neste projeto.

- Mostra todos os hashcodes, autores, e-mails, datas de modificação e as mensagens de cada commit

Diz ao Git que você quer visualizar a versão mais antiga do commit, e nesse caso precisamos informar que versão é essa (por meio do hashcode).

```
git checkout hashcode
```

- Os arquivos existentes na sua pasta, serão modificados para as versões mais antigas.
- Para voltar a versão mais recente, use o comando git checkout master

E agora, o que fazer?

Neste modulo do e-book você aprendeu um pouco sobre:

* Preparar o ambiente de desenvolvimento do Git
* Fazer configurações iniciais na conta do usuário do Git
* Aprendeu um pouco mais a fundo sobre o fluxo de trabalho do git
* Aprendeu a versionar usando os comandos git add, status, commit, log e checkout

E então, pronto para subir seu código para o github?

END (y/n)

SEÇÃO BÔNUS

Como faço para acessar um segundo commit existente dentro um commit anterior?

Vamos supor que realizamos 3 commits, para um arquivo único chamado *'planetas.txt'*.

O **primeiro commit**, armazenou o seguinte valor:

```
1  Nenhum Planeta Descoberto!
```

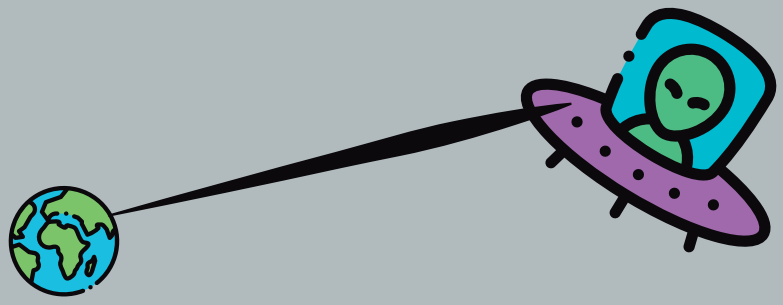
Já o **segundo commit**, armazenou o valor:

```
1  Nenhum Planeta Descoberto!
2    - 'Marte' Encontrada!
```

E o **terceiro commit**, armazenou o seguinte valor:

```
1  Nenhum Planeta Descoberto!
2    - 'Marte' Encontrada!
3    - 'Vênus' Encontrado!
```

Este exemplo, exemplifica a raça humana na descoberta de novos planetas em nosso sistema solar, ok?



Agora vamos supor, que em uma realidade alternativa, a raça humana foi dizimada por alienígenas, antes mesmo de encontrar o planeta marte.

Para representarmos isso com o Git, devemos voltar para a primeira versão e editar o primeiro arquivo, de modo a criar uma realidade paralela.

```
git log
git checkout hashcodedaprimeiraversão
```

```
1  Nenhum Planeta Descoberto!
2  [Raça Humana Dizimada por ALIENS]
```

```
git add .
git commit -m ‘raça humana dizimada por aliens’
```

Agora se dentro dessa primeira versão, dermos um git log, veremos uma versão a mais dentro de uma única versão, legal, isso nos prova que tudo deu certo.

Mas e se resolvêssemos voltar para o **MASTER** e executarmos um git log a partir ali? Será que o commit **'raça humana dizima...'** irá aparecer na frente do último commit? *vamos tentar!*

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

SeuUser@DESKTOP-UHDBDBV MINGW64 ~/Desktop/Mundos      (master)
$ git log

commit 77sa6hha871jjslko9188sjmj2nnsu72hhdskk299al (HEAD -> master)
Author: Micilini <micilini@gmail.com>
Date: Thu Mar 18 22:17:45 2023 -0300

     Vênus Encontrado

commit ia8j21nnsh7672jjdOkj2nd9iejf98j43fOo9kdO9jd
Author: Micilini <micilini@gmail.com>
Date: Thu Mar 18 22:16:24 2023 -0300

     Marte Encontrado

commit uua983kmfhju9oo2k92i30i03ef93j4f8ij4983nrf9
Author: Micilini <micilini@gmail.com>
Date: Thu Mar 18 22:14:12 2023 -0300

    inicio das descobertas
```

Hum.... não apareceu não é mesmo?

Faz sentido, uma vez que aquela nova versão que criamos, só existe dentro da primeira versão... Agora que tal voltarmos lá e darmos mais uma olhada?

"Socorro !!! Meu Arquivo sumiu, mesmo depois de gravar"

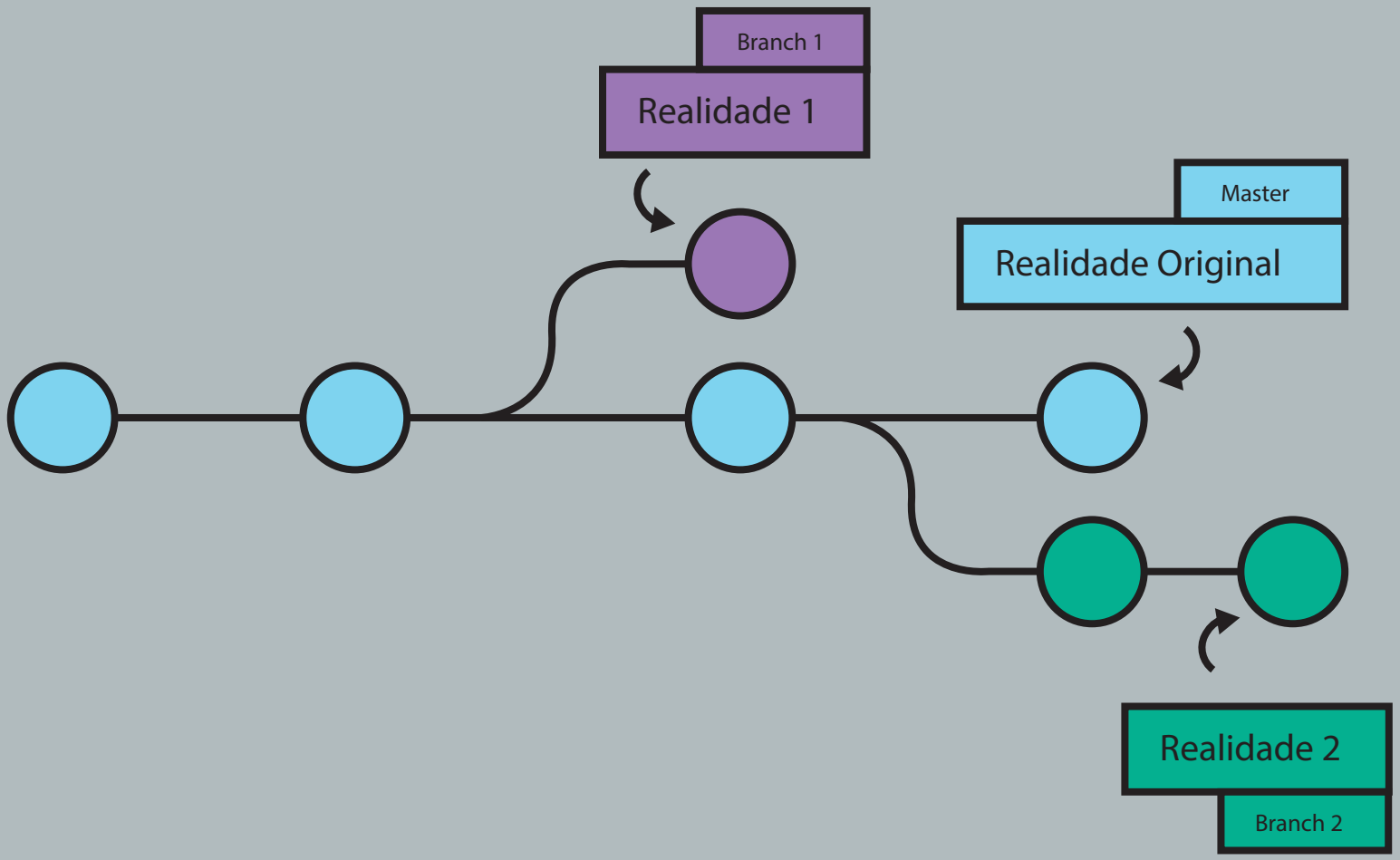
"Não se preocupe, ele ainda está lá, mas você não conseguirá acessá-lo da maneira convencional..."

Atualmente, através da linha de comando usando o git log, não é possível encontrar o arquivo modificado, ao menos que você tenha salvo o hashcode em algum lugar.

Isso é um erro do git? Provavelmente sim, mas essa ideia de criarmos uma realidade paralela, existe dentro do git e se chama Branch.

Como o nome já nos diz, um branch cria ramificações no seu projeto de versionamento.

Nesse caso é possível voltar a versões anteriores e criar novas ramificações, novas realidades e versões a partir de versões antigas, observe:



END (y/n)