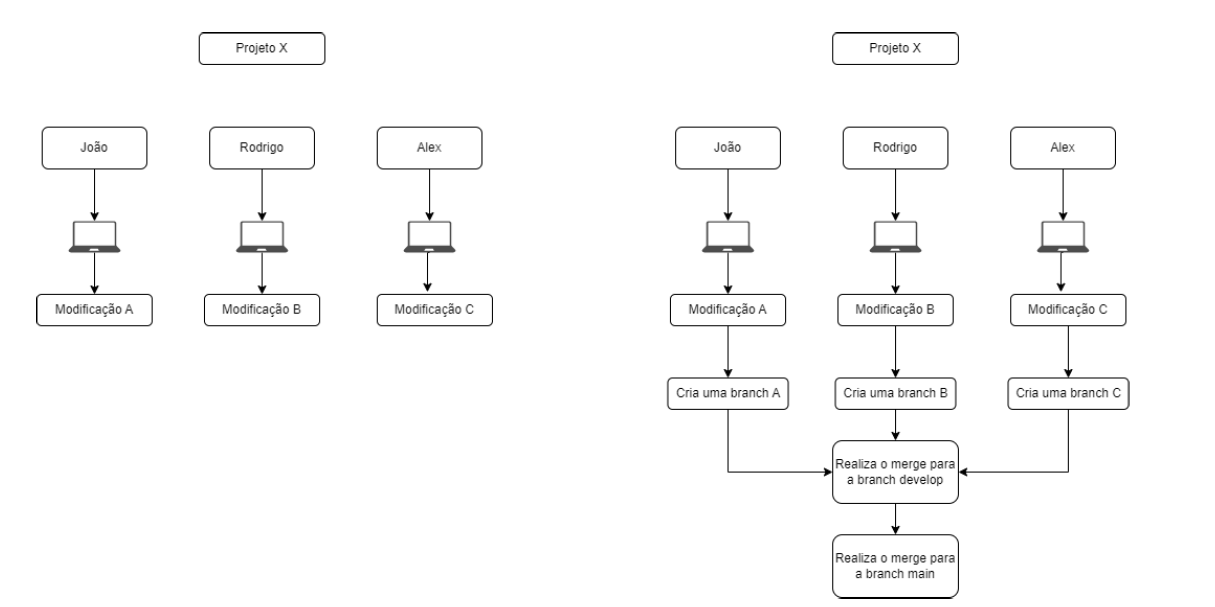
**GIT E GITHUB**

O que é o Git?

Git é um sistema de controle de versões distribuído, usado principalmente no desenvolvimento de software, mas pode ser usado para registrar o histórico de edições de qualquer tipo de arquivo.

Exemplo:

João, Rodrigo e Alex trabalham em um projeto x, eles tem cada um uma cópia do projeto na sua máquina local, eles precisam realizar alterações neste projeto em suas máquinas localmente e após realizar essas alterações precisam juntar o código desenvolvido a fim de gerar uma nova versão do projeto.



Como conseguir realizar esse processo sem ter que ficar passando arquivos por pendrive ou email? E como lidar para que a alteração de ninguém seja deixada para trás? Tudo isso se resolve com o Git.

● **Repositório:** É o local onde estão as modificações realizadas no projeto ele pode estar por exemplo no Github.

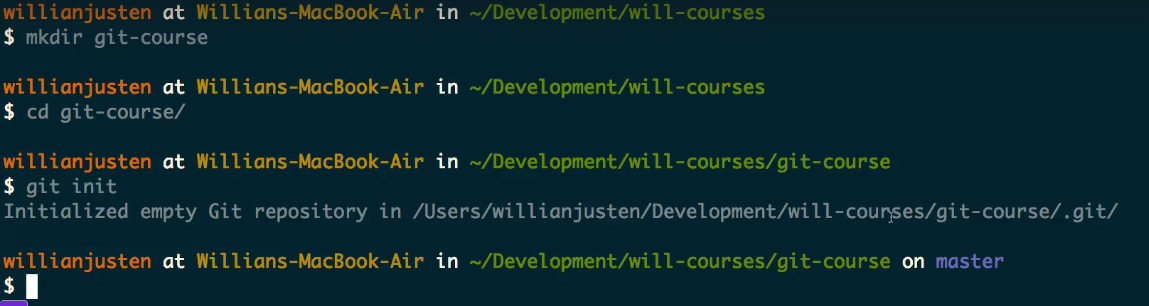
● **Branches:** São ramificações que permite desenvolver novas funcionalidades ou corrigir problemas de forma isolada, sem afetar o código principal. Ela funciona como um ponteiro móvel que aponta para commits específicos.

● **Commits:** São alterações concluídas presentes na branche.

Principais comandos do Git

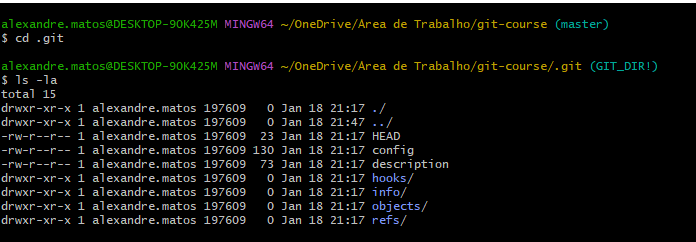
* **git clone**: Clona um projeto de outro repositório na internet para sua máquina local
* **git init:** Inicializa um novo repositório git localmente
* **git add**: Adiciona as modificações atuais feitas localmente para serem enviadas posteriormente à branch principal
* **git commit**: Adiciona as alterações já realizadas em estado de pronto para ser adicionada à branch principal
* **git pull**: Baixa as alterações realizadas de outra branch no repositório
* **git push**: Envia as alterações realizadas para o repositório
* **git merge**: Junta localmente o conteúdo de duas branches separadas
* **git checkout -b**: Cria outra branch separada
* **git branch -d**: Delete uma branch
* **git reset HEAD~1**: Realiza o reset da branch seleciona, voltando para um estado anterior no código
* **git log**: Consulta as alterações realizadas na branch localmente
* **git log --decorate**: exibe o histórico de commits com informações adicionais sobre referências como branches e tags associadas a cada commit.
* **git log --author=”Alexandre”**: vai listar todos os commit que tenha o nome com “Alexandre”
* **git shortlog**: Mostra em ordem alfabética quais foram os autores, quantos commits fizeram e quais eles foram.
* **git shortlog -sn**: Mostra só a quantidade de commits e a pessoa que commitou.
* **git shortlog --graph**: Mostra de forma gráfica o que está acontecendo com os branchs e versões.
* **git show 5a17162e90960c7dded65bfbfc113100186db68a**: Informando o has (5a17162e...) do commit, você consegue ver tudo que foi adicionado ou alterado.
* **git status**: Verifica se há alterações realizadas localmente.
* **git config --list**: Lista as configurações atuais do Git na sua máquina.

Abrindo o git na sua máquina ou o Terminal (Prompt de Comando) começamos criando uma pasta.



Após digitar **mkdir** e o nome da pasta que é **git-course** você dá **enter** e a pasta é criada;

**cd** é usado para acessar uma pasta, ex: cd git-course;



Na imagem que você enviou, você está dentro de um diretório chamado .git, que é o **diretório interno do Git** usado para armazenar todos os metadados do repositório. Esse diretório é automaticamente criado quando você inicializa um repositório com git init ou quando clona um repositório existente.

Aqui está uma explicação do que cada item listado significa:

1. **./** e **../**:
   * ./: Refere-se ao diretório atual (o próprio .git).
   * ../: Refere-se ao diretório pai (no caso, o diretório git-course).
2. **HEAD**:
   * Um arquivo que aponta para o commit ou branch atualmente "checado" no repositório. Ele determina onde você está trabalhando no momento.
3. **config**:
   * Contém as configurações específicas do repositório Git, como informações de usuário, branches remotos, etc.
4. **description**:
   * Um arquivo simples usado em repositórios "bare" (sem um diretório de trabalho) para descrever o repositório. Geralmente não é importante em repositórios locais.
5. **hooks/**:
   * Um diretório que contém scripts de **hooks do Git**, que permitem executar comandos personalizados em eventos específicos (como pre-commit ou post-merge).
6. **info/**:
   * Contém informações adicionais, como um arquivo exclude, que permite ignorar arquivos localmente (funciona de forma semelhante ao .gitignore, mas não é compartilhado com outros usuários).
7. **objects/**:
   * Um dos diretórios mais importantes. Contém todos os objetos do Git, como commits, árvores (trees), blobs (conteúdo dos arquivos) e tags. Tudo que você faz no Git (como adicionar, commitar, etc.) é registrado aqui.
8. **refs/**:
   * Contém referências para branches e tags no repositório. Por exemplo, as referências para main, develop ou outras branches criadas.

**Contexto Geral**

O que você está vendo é a estrutura interna de um repositório Git. Geralmente, você não precisa acessar o .git diretamente, pois o Git gerencia automaticamente esses arquivos e diretórios. No entanto, isso é útil para entender como o Git armazena informações e opera nos bastidores.



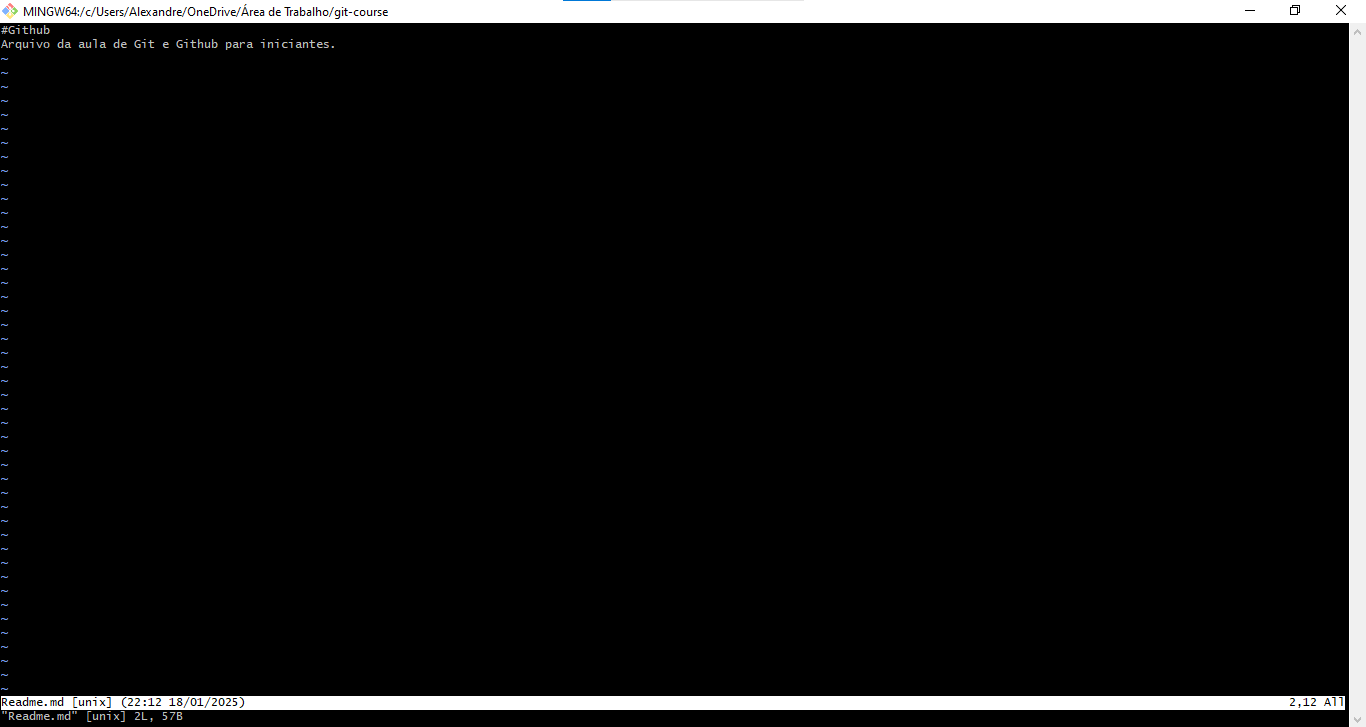
**ls**: Lista os arquivos e diretórios do diretório atual.

**-l**: Mostra os detalhes em formato de lista longa, incluindo:

* Permissões dos arquivos/diretórios.
* Número de links.
* Proprietário.
* Grupo.
* Tamanho.
* Data e hora da última modificação.
* Nome do arquivo/diretório.

**-a**: Mostra todos os arquivos, incluindo os ocultos (arquivos ou diretórios que começam com um ponto, como .git).

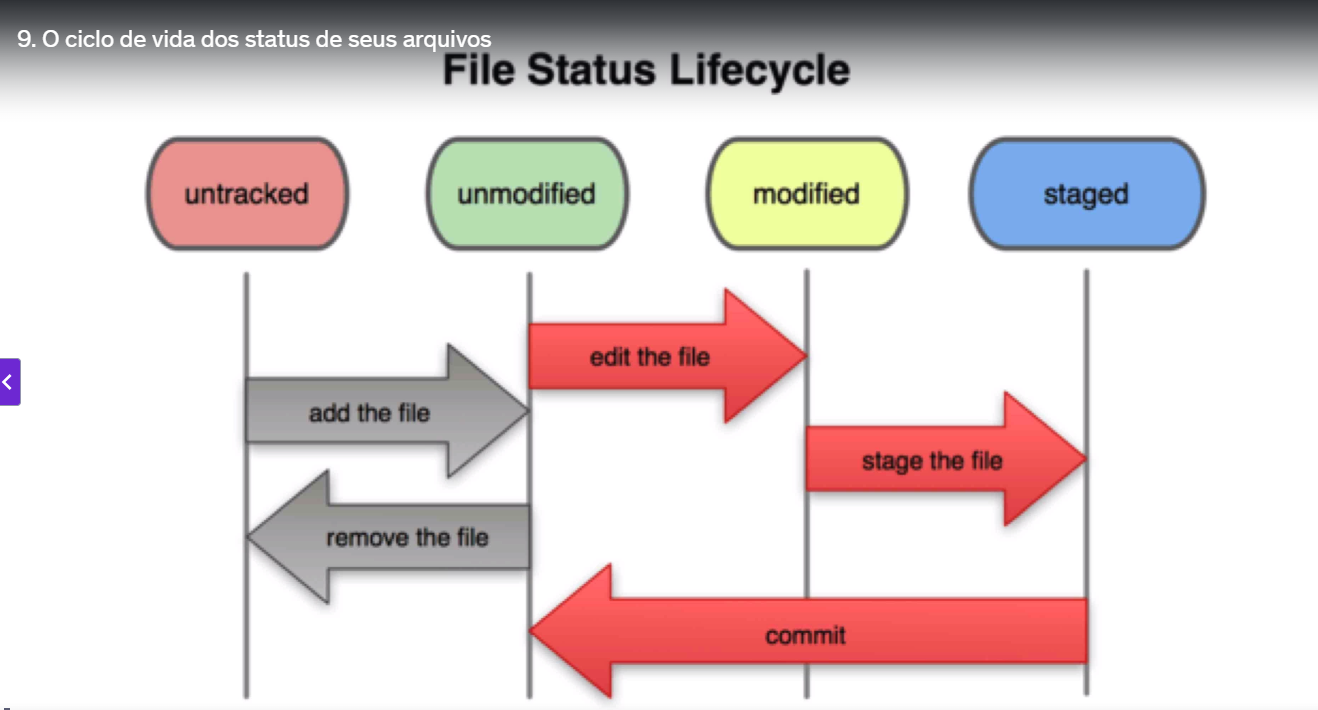




Ao você digitar: **vi** e o nome do arquivo **Readme.md,** você entrará no modo de edição do arquivo;

Depois seleciona a tecla **i** para inserir (insert);

Ao terminar de fazer a inserção só pertar **Esc** e em seguida clicar em (**:** e depois em **wq** e **Enter** e o arquivo será salvo com a modificação.



O ciclo de vida dos status dos arquivos no Git, representado na imagem, descreve as diferentes etapas pelas quais um arquivo pode passar dentro de um repositório Git. Aqui está o significado de cada status e a transição entre eles:

### 1. ****Untracked**** (Vermelho - Não rastreado):

* Representa arquivos que existem no diretório de trabalho, mas que o Git ainda não está rastreando.
* Esses arquivos não fazem parte do controle de versão.

#### Como mudar de Untracked para Staged:

* Use o comando git add <arquivo> para adicionar o arquivo ao índice (área de stage).

### 2. ****Unmodified**** (Verde - Não modificado):

* São arquivos que já estão sendo rastreados pelo Git e que **não foram modificados** desde o último commit.
* Eles estão no estado "limpo".

#### Como mudar de Unmodified para Modified:

* Edite o arquivo.

### 3. ****Modified**** (Amarelo - Modificado):

* Representa arquivos que foram **modificados** desde o último commit.
* O Git detecta as alterações, mas elas ainda **não foram preparadas para commit**.

#### Como mudar de Modified para Staged:

* Use o comando git add <arquivo> para mover o arquivo modificado para a área de stage.

### 4. ****Staged**** (Azul - Preparado):

* Arquivos que estão na **área de stage** e prontos para serem incluídos no próximo commit.
* O estágio serve como uma etapa intermediária antes de salvar as alterações no histórico do repositório.

#### Como mudar de Staged para Committed:

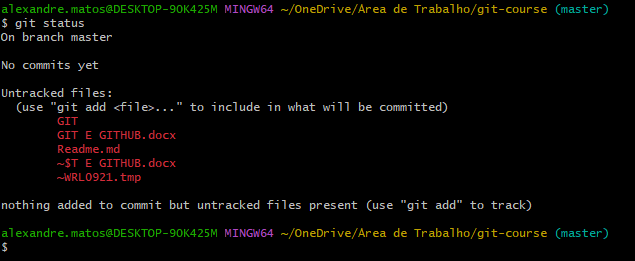
* Use o comando git commit -m "<mensagem>" para registrar as alterações no repositório.

### Transições no Ciclo de Vida:

1. **Untracked → Staged**: Adicione o arquivo com git add.
2. **Unmodified → Modified**: Edite o arquivo.
3. **Modified → Staged**: Use git add.
4. **Staged → Committed**: Use git commit.

### Resumo:

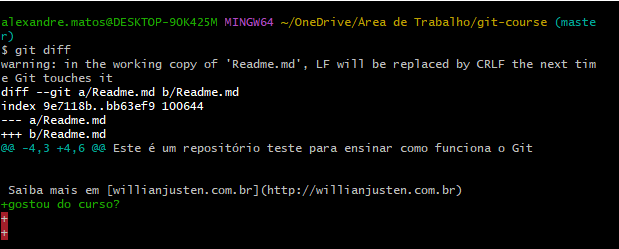
* O ciclo reflete como o Git gerencia arquivos para garantir que apenas as alterações necessárias sejam rastreadas e incluídas no histórico do repositório.
* Este fluxo ajuda a organizar as alterações antes de registrá-las com um commit. Isso é útil para manter o controle e evitar erros.



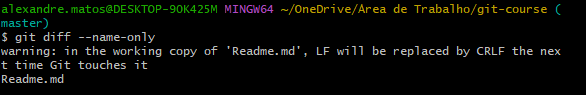
**git status**: serve para reportar como está o meu repositório nesse momento

**git add**: adiciona ao índice

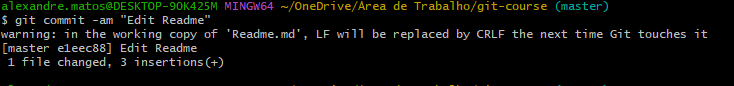
**git commit -m**: commitar o que foi adicionado e em seguida escreve um comentário dentro de “update...”



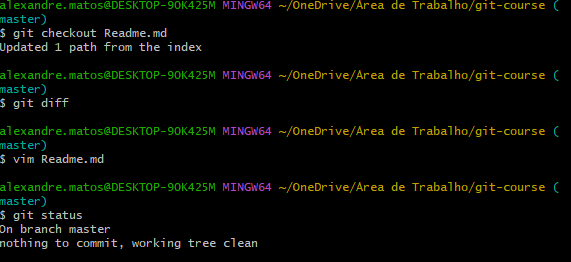
**git diff**: mostra a sua modificação, conforme imagem acima na parte do ‘+gostou do curso?’, sempre é bom usar o ‘git diff’ antes de você commitar, porque ajuda a revisar para não subir alguma coisa suja.



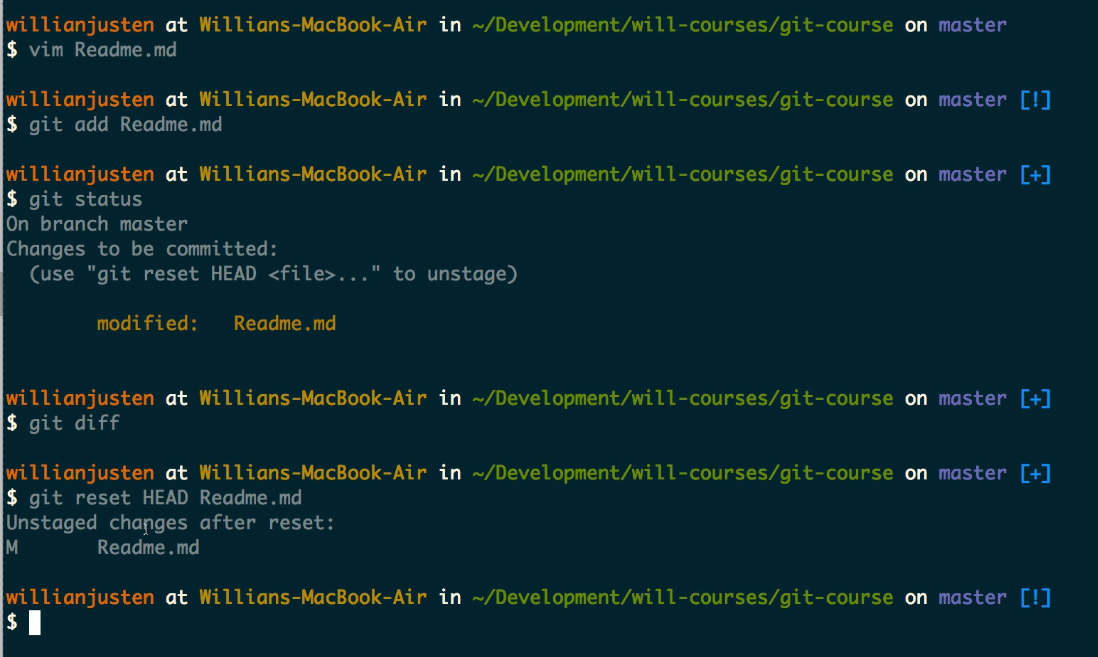
**git diff --name-only**: somente para mostrar o nome do arquivo que foi modificado.



**git commit -am “Edit Readme”**: quando for um arquivo que já existiu basta digitar assim, conforme imagem.



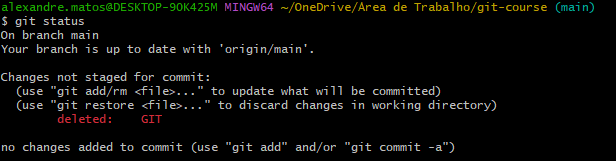
**git checkout**: fazendo uma alteração em um documento (Readme.md) e clicando em ‘ git status’, logo vai constar que eu tenho uma alteração e se eu quero commitar, só que se eu fazer o ‘git checkout’, logo voltará ao status anterior sem alterações, como imagem acima.



Agora se o arquivo estiver na fila do **STAGED (preparado)**, pronto para commitar só que ao clicar em ‘**git diff**’ vamos perceber que não aponta mais as alterações pois elas já estão preparadas para commitar, só que eu quero alterar as modificações e não quero que suba assim, então eu digito ‘**git reset HEAD Readme.md**’ ai consigo tirar de **STAGED** e assim posso modificar o arquivo.  
  
  
O comando **git reset** é usado para **desfazer alterações**, movendo commits, alterações na área de stage ou no diretório de trabalho para estados anteriores, dependendo da opção usada (--soft, --mixed, --hard).

**Principais opções:**

1. **--soft**: Mantém as alterações no stage e no diretório de trabalho, mas desfaz commits.
2. **--mixed** (padrão): Remove alterações da área de stage, mas mantém no diretório de trabalho.
3. **--hard**: Remove tudo — commits, alterações no stage e no diretório de trabalho.



Quando você deletou o arquivo no seu sistema local, o Git detectou essa exclusão, mas ela ainda precisa ser confirmada para que o arquivo seja removido do repositório Git. Para deletar o arquivo do Git, siga estes passos:

### 1. ****Confirmar a exclusão no Git:****

Use o comando abaixo para marcar a exclusão do arquivo para o próximo commit: git rm GIT

### 2. ****Fazer o commit para confirmar a remoção:****

Após confirmar a exclusão, faça um commit para registrar a mudança no repositório:

git commit -m "Removendo o arquivo GIT do repositório"

### 3. ****Enviar a alteração para o repositório remoto (opcional):****

Se o repositório estiver conectado a um repositório remoto, envie as alterações com:

git push

### Observação:

Se você quiser apenas **parar de rastrear o arquivo**, mas mantê-lo no diretório local, use:

git rm --cached GIT

Isso remove o arquivo do controle do Git, mas o mantém no sistema de arquivos.

Se você deseja recuperar o arquivo deletado no repositório local, pode usar o comando git restore ou um comando equivalente para restaurar o arquivo a partir do histórico do Git.

**Passos para recuperar o arquivo deletado:**

1. **Restaurar o arquivo deletado:** Use o comando abaixo para recuperar o arquivo deletado do repositório local:
2. git restore GIT
3. **Confirmar que o arquivo foi restaurado:** Verifique se o arquivo foi restaurado para o diretório local:
4. git status

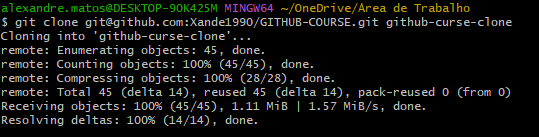
**Caso o comando git restore não funcione:**

Se o seu Git for uma versão mais antiga (antes do git restore), você pode usar:

git checkout -- GIT

**Observação:**

* Esse procedimento restaura o arquivo **à última versão confirmada (commit)** no repositório local.
* Caso o arquivo tenha sido alterado antes da exclusão, você precisará verificar os commits anteriores para recuperar versões específicas.



O **git clone** é um comando usado para **copiar um repositório remoto** (como no GitHub, GitLab ou Bitbucket) para o seu computador. Ele cria uma cópia completa do repositório, incluindo todo o histórico de commits, branches e arquivos.

**QUAL A DIFERENÇA ENTRE CLONAR E BAIXAR O REPOSITÓRIO PARA SUA MÁQUINA?**

**Para que serve?**

* **Baixar o repositório remoto para trabalhar localmente.**
* É útil para iniciar o trabalho em projetos existentes.

**Exemplo básico:**

git clone <URL-do-repositorio>

* **URL do repositório**: Pode ser HTTPS ou SSH.
* O comando cria uma pasta local com o mesmo nome do repositório e copia todo o conteúdo para ela.

**Exemplo completo:**

Se você clonar este repositório:

git clone https://github.com/Xande1990/GITHUB-COURSE.git

* O repositório remoto será copiado para o seu computador.
* Você terá todos os arquivos e histórico de alterações para começar a trabalhar localmente.

A diferença entre **clonar um repositório** e **baixar um repositório** do GitHub está na forma como você interage com o repositório e no que você obtém:

**1. Clonar o repositório (git clone)**

* **O que acontece?**
  + Cria uma **cópia completa** do repositório no seu computador.
  + Inclui **todo o histórico de commits**, branches, tags e o vínculo com o repositório remoto.
  + Permite **sincronizar alterações** com o repositório remoto usando comandos como git pull (para baixar mudanças) e git push (para enviar mudanças).
* **Vantagens:**
  + Você pode trabalhar no projeto localmente e contribuir com commits.
  + Pode enviar e receber alterações do repositório remoto (colaboração).
  + Mantém um controle total do histórico do projeto.

**2. Baixar o repositório (Download ZIP):**

* **O que acontece?**
  + Você baixa um **arquivo ZIP** com os arquivos do repositório no estado atual (branch escolhida no momento do download).
  + **Não inclui o histórico de commits** ou a conexão com o repositório remoto.
* **Vantagens:**
  + Rápido e simples, especialmente se você só quer os arquivos.
  + Ideal para quem não precisa colaborar ou entender o histórico do projeto.

**Diferença na prática:**

| **Aspecto** | **Clonar (git clone)** | **Baixar (ZIP)** |
| --- | --- | --- |
| Histórico de commits | Sim | Não |
| Branches | Sim | Apenas a branch baixada |
| Conexão com o remoto | Sim | Não |
| Sincronização futura | Sim (git pull, git push) | Não |
| Ideal para | Desenvolvimento e colaboração | Apenas usar os arquivos |

**Qual escolher?**

* **Clonar**: Se você vai trabalhar no projeto, colaborar ou precisa acessar o histórico.
* **Baixar**: Se você só precisa dos arquivos para leitura ou uso rápido, sem colaborar no projeto.

**FORK**

No GitHub, o **fork** é usado para criar uma **cópia completa de um repositório** de outro usuário no seu próprio perfil. É especialmente útil para colaborar em projetos públicos ou para personalizar um projeto existente sem afetar o original.

**Principais funcionalidades do fork:**

1. **Trabalhar em um projeto sem afetar o original**:
   * Você pode alterar o código no seu repositório forkado sem impactar o repositório original.
2. **Colaborar em projetos open source**:
   * Forkar um projeto permite que você faça alterações no seu repositório, teste suas mudanças e depois envie essas alterações para o repositório original usando um **Pull Request**.
3. **Personalizar um projeto existente**:
   * Se você quer usar um projeto público como base e fazer alterações específicas para seu uso, o fork é ideal.
4. **Explorar e aprender**:
   * Forks são uma boa maneira de estudar o código de outras pessoas, testar mudanças e aprender sem o risco de alterar o projeto original.

**Como funciona o fluxo com fork?**

1. **Forkar o repositório:**
   * Clique no botão **"Fork"** no GitHub para criar uma cópia do repositório no seu perfil.
2. **Clonar o fork para sua máquina:**
   * Use o comando:
   * git clone <URL-do-seu-fork>
3. **Fazer alterações no repositório local**:
   * Edite, crie ou remova arquivos.
4. **Enviar as alterações para o seu fork no GitHub**:
   * Use os comandos:
   * git add .
   * git commit -m "Descrição das alterações"
   * git push origin main
5. **Enviar um Pull Request** (opcional):
   * Se você quer contribuir com o repositório original, envie um Pull Request para que os mantenedores revisem e aceitem suas alterações.

**Diferença entre fork e clone:**

* **Fork**: Cria uma cópia no GitHub (no seu perfil), útil para colaboração.
* **Clone**: Baixa o repositório (original ou fork) para sua máquina local.

**BRANCHES**

Uma **branch** no Git é como uma **linha de desenvolvimento independente** dentro de um repositório. Ela permite que você trabalhe em novas funcionalidades, correções de bugs ou experimentos sem afetar diretamente o código principal (geralmente na branch **main** ou **master**).

**Por que usar branches?**

Branches permitem que você:

1. **Isolar mudanças**: Trabalhe em uma nova funcionalidade sem interferir no código já existente.
2. **Colaborar com equipes**: Cada pessoa pode trabalhar em uma branch separada e depois combinar as alterações.
3. **Manter a estabilidade**: O código principal (na branch main) permanece estável enquanto novas alterações são desenvolvidas.

**Como funcionam?**

* Por padrão, quando você inicia um repositório Git, ele cria uma branch chamada **main**.
* Você pode criar **outras branches** para trabalhar em partes específicas do projeto.
* Depois de finalizar o trabalho, as alterações de uma branch podem ser **mescladas (merged)** de volta à branch principal.

**Exemplos de uso:**

1. **Criar uma nova branch:**

git branch feature-nova

Isso cria uma branch chamada feature-nova.

1. **Mudar para outra branch:**

git checkout feature-nova

Ou, com versões mais recentes do Git:

git switch feature-nova

1. **Criar e mudar para a branch ao mesmo tempo:**

git checkout -b feature-nova

Ou:

git switch -c feature-nova

1. **Mesclar (merge) uma branch na branch principal:** Após finalizar o trabalho em uma branch, você pode integrá-la ao código principal:

git checkout main

git merge feature-nova

**Benefícios das branches:**

* **Colaboração eficiente**: Múltiplos desenvolvedores podem trabalhar simultaneamente sem conflitos.
* **Flexibilidade**: Permite experimentar novas ideias sem comprometer o código existente.
* **Controle**: Facilita a organização e a revisão de alterações antes de incorporá-las ao projeto principal.