

# E-BOOK GIT & GITHUB DIGITAL

*O BÁSICO PARA INICIAR*



Criado por: Lorenzo Uriel

# Table of Contents

<b>Sejam Bem-Vindos!</b>	<b>4</b>
Sobre o Autor	5
Ebook PDF	6
 <b>Iniciando no Universo do Git &amp; GitHub</b>	 <b>7</b>
O que é o Git?	8
Instalação e Configuração Inicial do Git	9
Instalação e Configuração Inicial do GitHub	10
 <b>Conceitos Fundamentais</b>	 <b>11</b>
Introdução aos Repositórios e Conceitos	12
Principais Comandos	14
Criando um Repo Local e Remoto	18
Introdução ao Markdown	22
 <b>Trabalhando com Branches</b>	 <b>25</b>
Criando e Gerenciando Branches Locais	26
Mesclando Branches (Merge)	27
Fluxo de Trabalho com Branches (Feature Branch, Hotfix e Release.)	29
 <b>Sincronização e Colaboração</b>	 <b>31</b>
Configurando Repositórios Remotos	32
Enviando Mudanças para o Repositório Remoto (git push)	33
Obtendo Mudanças do Repositório Remoto (git pull e git fetch)	34
Trabalhando com Forks e Pull Requests	35
Exemplo de Fluxo Completo	37

<b>Gerenciamento de Repositórios</b>	<b>41</b>
Removendo Repositórios Locais	42
Excluindo Repositórios Remotos no GitHub	43
Limpeza de Histórico (Rebase, Squash)	44
Tags e Releases	47
 <b>Práticas Recomendadas</b>	 <b>50</b>
Commits Claros e Frequentes	51
Uso de .gitignore	52
Reposicionamento de Branches	53
Manutenção de Histórico Limpo e Compreensível	54
Automação de Tarefas com Hooks do Git	55
Resumo	56
 <b>Solução de Problemas</b>	 <b>57</b>
Desfazendo Commits (git revert, git reset, git checkout)	58
Resolução de Conflitos Comuns	61
Recuperação de Commits Perdidos	63
Dicas de Troubleshooting	65
 <b>Ferramentas e Integrações</b>	 <b>67</b>
Integração do Git com IDEs Populares	68
Uso de Ferramentas Gráficas para Git	69
Integração com CI/CD	70
GitFlow	71

**Sejam Bem-Vindos!**

## Sobre o Autor

Meu nome é Lorenzo Uriel, nascido em 2000 e atualmente sou Administrador de Banco de Dados. Graduado em Sistemas de Informação pela Anhanguera Educacional e pós-graduado em Gestão de Projetos pelo Instituto Mackenzie.

Estou sempre fazendo o que eu posso para compartilhar meus aprendizados e conhecimento com a comunidade, esse e-book é um reflexo disso.

Você pode me seguir nas minhas redes e acompanhar um pouco mais do meu trabalho:

- [https://linktr.ee/lorenzo\\_uriel](https://linktr.ee/lorenzo_uriel)

## Ebook PDF

Esse ebook foi criado usando a ferramenta ibis, criada por: [Mohamed Said](https://github.com/themsaid). - <https://github.com/themsaid>

Ibis é uma ferramenta PHP que ajuda você a escrever e-books em Markdown e depois exportar para PDF.

Ela permite que você crie o seu e-book nos temas Light e Dark.

# Iniciando no Universo do Git & GitHub

## O que é o Git?

O Git é um **sistema de controle de versão**, projetado para rastrear alterações em projetos de software e coordenar o trabalho de várias pessoas neles.

Desenvolvido por **Linus Torvalds em 2005**, o Git se destaca por sua eficiência, flexibilidade e capacidade de lidar com projetos de qualquer tamanho.

Ele registra as alterações no código-fonte, permite que várias ramificações de desenvolvimento existam simultaneamente e facilita a fusão de código de diferentes colaboradores.

A principal função do Git é permitir que múltiplas pessoas trabalhem no mesmo código simultaneamente, sem causar conflitos ou perda de dados.

### Principais características do Git:

- **Distribuído:** Cada desenvolvedor possui uma cópia completa do histórico do repositório.
- **Desempenho:** Projetado para ser rápido e eficiente, mesmo com grandes quantidades de dados.
- **Segurança:** Garante a integridade do código e das mudanças realizadas.
- **Flexibilidade:** Suporta diversos fluxos de trabalho e processos de desenvolvimento.



# Instalação e Configuração Inicial do Git

## Passo 1: Instalação do Git

Para instalar o Git, siga os passos abaixo de acordo com o seu sistema operacional:

### Windows:

- Baixe o instalador do Git no site oficial: <https://git-scm.com/>.
- Execute o instalador e siga as instruções na tela, mantendo as configurações padrão.

### macOS:

- Abra o Terminal.
- Execute o comando: `brew install git`. (Requer Homebrew, que pode ser instalado a partir de <https://brew.sh/>).

### Linux:

- Abra o terminal.
- Execute o comando apropriado para sua distribuição:
  - Debian/Ubuntu: `sudo apt-get install git`.

## Passo 2: Configuração inicial do Git

Após instalar o Git, é necessário configurá-lo. Use os seguintes comandos no terminal para configurar seu nome de usuário e email, que serão usados em seus commits:

```
git config --global user.name "Seu Nome"
git config --global user.email "seuemail@example.com"
```

Você pode verificar suas configurações com o comando:

```
git config --list
```

# Instalação e Configuração Inicial do GitHub

**GitHub é uma plataforma de hospedagem de código-fonte que utiliza o Git para controle de versão.** Ele permite colaborar em projetos e compartilhar código com outros desenvolvedores.

## Passo 1: Criando uma conta no GitHub

- Acesse <https://github.com/>.
- Clique em "Sign up" e siga os passos para criar uma nova conta.

## Passo 2: Configurando o Git para usar o GitHub (Windows)

- Inicie o Git Bash e gere uma chave SSH para autenticação segura:
  - No terminal, execute: `ssh-keygen -t rsa -C "seuemail@example.com"`.
  - Pressione Enter para aceitar o local padrão do arquivo.
  - Digite uma senha segura (ou deixe em branco).
- Adicione a chave SSH ao seu GitHub:
  - Copie a chave pública que foi gerada.
  - Vá até as configurações do GitHub (Settings) > "SSH and GPG keys".
  - Clique em "New SSH key", cole a chave e salve.

Os passos são os mesmos para os outros sistemas operacionais, você vai precisar entender quais os comandos e onde eles salvam as chaves SSH que foram geradas.

**Passo 3: Conectando-se ao GitHub** Para clonar um repositório do GitHub, use o comando:

```
git clone git@github.com:usuario/nome-do-repositorio.git
```

# Conceitos Fundamentais

# Introdução aos Repositórios e Conceitos

Um repositório é o local onde o projeto, suas alterações e arquivos são armazenados. Ele pode ser local ou remoto, cada um com suas características específicas:

- **Remoto:** Hospedado em plataformas como GitHub, outros sistemas de controle de versão (CVS) ou servidores dedicados.
- **Local:** Armazenado diretamente na máquina do desenvolvedor.

## Principais Conceitos

### Branches (ou Ramos)

As **branches (ou ramos)** são **versões paralelas do projeto**. A branch principal geralmente é chamada de **main**, mas é possível criar novas branches para desenvolver alterações específicas sem afetar a branch principal.

- **Exemplo:** Crie um branch chamada **dev-task-13**, nesse branch você vai focar em finalizar a task 13 do seu backlog, quando finalizar você **commita** e faz um **merge** com a branch principal. Assim, você não afeta o seu branch principal com alterações que podem quebrar o código.

### Commit

Um **commit** é uma operação que captura o estado atual do projeto. É como tirar uma foto dos arquivos naquele momento. Cada commit tem um identificador único (**hash**) e uma mensagem descritiva.

### Merge

O **merge** é a operação de combinar as mudanças de diferentes branches. Geralmente, você realiza um merge quando deseja integrar as alterações de uma branch de feature ou correção de bug de volta à branch principal.

### Issues

No contexto do Git, uma issue refere-se a uma maneira de rastrear tarefas, melhorias, erros (bugs) ou discussões relacionadas a um projeto específico.

## Estrutura do Repositório

### HEAD

O **HEAD** é um ponteiro que aponta para o **commit mais recente na branch atual**. Quando

quando você faz um novo commit, o **HEAD** se move para esse novo commit. Ele também é usado para determinar o estado atual da Working Tree e do Index.

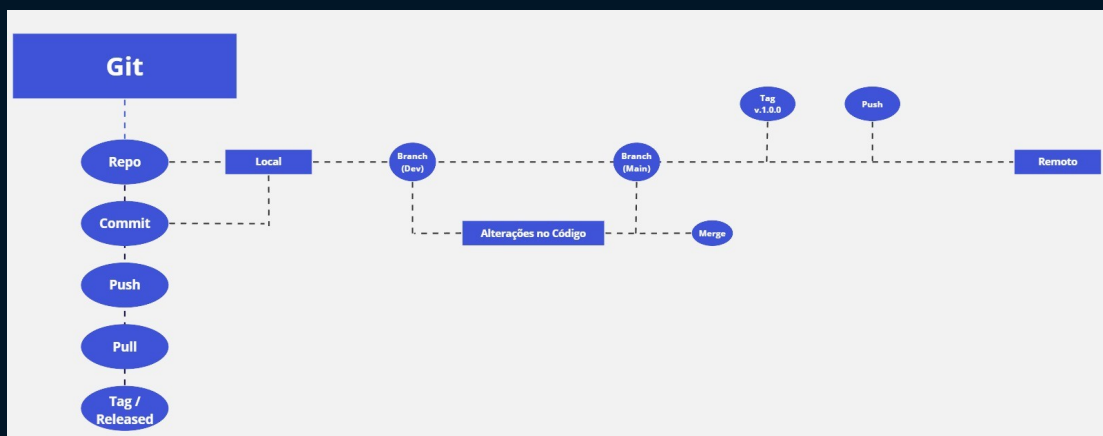
## Working Tree

A Working Tree é o local onde os arquivos estão realmente armazenados e onde você faz suas alterações.

## Index (Índice)

O index (ou índice) é o local onde o Git armazena o que será commitado. Ele atua como uma área intermediária entre a Working Tree e o repositório Git. Para adicionar arquivos ao índice, usamos o comando: `git add arquivo.txt`

## Arquitetura Git e Conceitos



# Principais Comandos

## Commit

Acontece quando queremos salvar as última atualizações que foram realizadas.

### Processos relacionados ao Commit:

- Verificar o status dos arquivos modificados:

```
git status
```

- Adicionar os arquivos para o commit:

```
git add arquivo1.txt arquivo2.js
```

```
git add .
```

- Realizar o commit:

```
git commit -m "Primeiro Commit"
```

O **-m** é utilizado para adicionarmos uma mensagem no commit, em seguida temos o texto **"Primeiro Commit"**.

## Push

Acontece quando enviamos as alterações para o repositório - "empurra" as modificações para o repositório remoto

### Processos relacionados ao Push:

- Realizar o push para o repositório remoto:

```
git push
```

```
# ou
```

```
git push origin nome-da-branch
```

- Se for a primeira conexão e primeiro push:

```
git push -u origin nome-da-branch
```

É necessário configurar a relação entre as branches locais e remotas usando o comando acima.

O **-u** estabelece uma relação de acompanhamento, facilitando futuros pushes e pulls.

## Pull

Atualiza o seu repositório local com as alterações no repositório remoto - "puxa" as alterações do repositório remoto.

Ele traz todas as alterações do seu repositório remoto para o local.

### Processos relacionados ao Pull:

- Realizar o pull para obter as alterações mais recentes:

```
git pull origin nome-da-branch
```

- Se você já configurou a relação de acompanhamento durante o git push **-u**, pode usar apenas:

```
git pull
```

## Tag

Uma tag no Git é uma referência específica a um ponto na história do seu repositório. É comumente usado para marcar versões estáveis ou importantes do seu projeto.

As tags são úteis para criar pontos de referência fixos que não se movem à medida que novos commits são feitos.

### Processos relacionados ao Tag:

- Listar as tags existentes:

```
git tag
```

- Criar uma nova tag:

```
git tag -a v1.0 -m "Versão 1.0"
```

- Este comando cria uma tag anotada chamada "**v1.0**" com uma mensagem descritiva "**Versão 1.0**".
  - **1. a: Cria uma tag anotada.**
  - **2. v1.0: Nome da tag.**
  - **3. -m "Versão 1.0": Mensagem descritiva associada à tag.**
- Compartilhar a tag no repositório remoto:

```
git push origin v1.0
```

### init


Inicializa um novo repositório Git no diretório atual.

```
git init
```

### clone

Clona um repositório remoto para o seu computador.

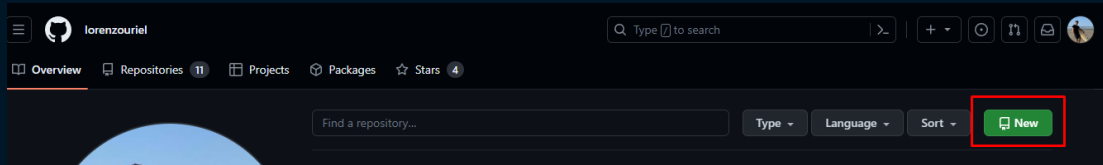




```
git clone <url-do-repositorio>
```

# Criando um Repo Local e Remoto

## 1. Vá até o seu GitHub e crie um novo repositório:



## 2. Navegue até o diretório local onde deseja criar o repositório—use o comando `cd` para entrar no diretório desejado.

```
cd c:\path\vagrant
```

## 3. Este comando cria um arquivo chamado `README.md` e insere o texto `#up-website-with-vagrant` nele. O `>>` é um operador de redirecionamento que acrescenta o texto ao final do arquivo, ou cria o arquivo se ele não existir.

```
echo "# up-website-with-vagrant" >> README.md
```

## 4. Inicializa um novo repositório Git no diretório atual.

```
git init
```

## 5. Adiciona o arquivo `README.md` ao índice. Isso prepara o arquivo para ser incluído no próximo commit.

```
git add README.md
```

## 6. Adiciona todas as alterações (Se houver)

```
git add .
```

## 7. Cria o primeiro commit no repositório com uma mensagem. O `-m` permite

**adicionar a mensagem de commit diretamente na linha de comando.**

```
git commit -m "first commit"
```

**8. Renomeia a branch padrão do repositório para main. Este comando é usado para atualizar o nome da branch principal para seguir as práticas mais recentes em relação ao uso de nomes, substituindo a antiga master**

```
git branch -M main
```

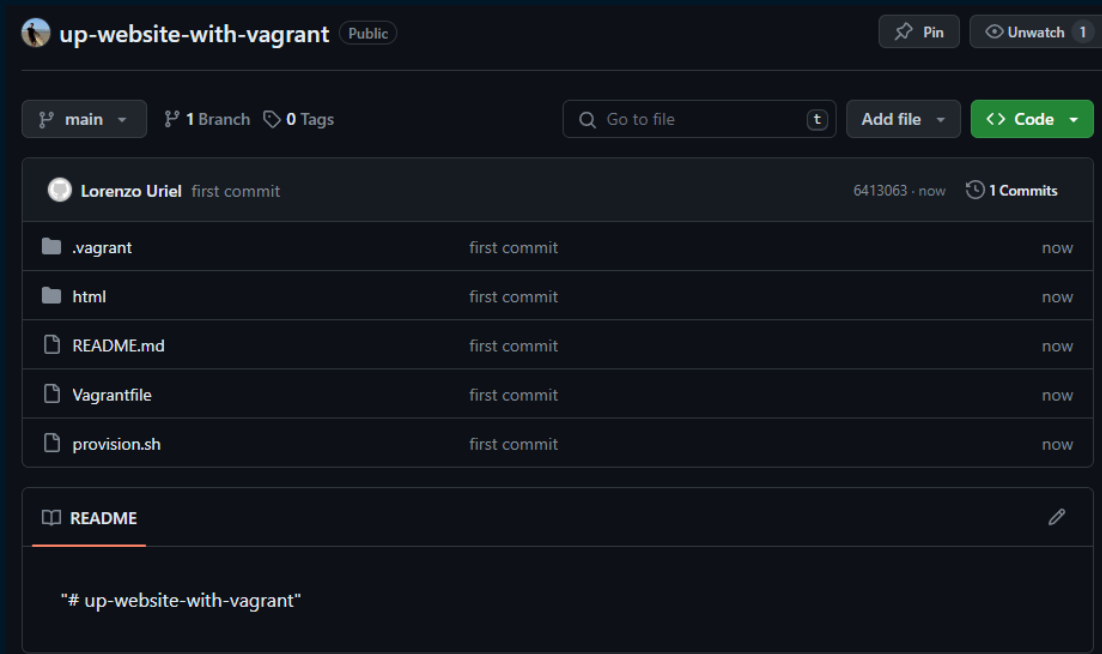
**9. Adiciona um repositório remoto chamado "origin". O termo "origin" é um padrão utilizado para referenciar o repositório remoto principal. O URL é o endereço do repositório no GitHub.**

```
git remote add origin  
https://github.com/lorenzouriel/up-website-with-vagrant.git
```

**10. Envia o repositório local para o repositório remoto ("origin") na branch principal (main). O -u estabelece uma relação de acompanhamento, associando automaticamente a branch local com a branch remota. Isso é útil para futuros git pull e git push sem a necessidade de especificar a branch.**

```
git push -u origin main
```

Podemos verificar o nosso repositório remoto:



## O que é Versionamento e Tags?

Já trabalhou com lançamento de versões?

Ou com Tags no Git?

As tags são bem importantes em nossos projetos - com elas conseguimos identificar em qual momento no tempo ocorreram os principais lançamentos e versões.

É um meio de organizar e documentar o seu trabalho.

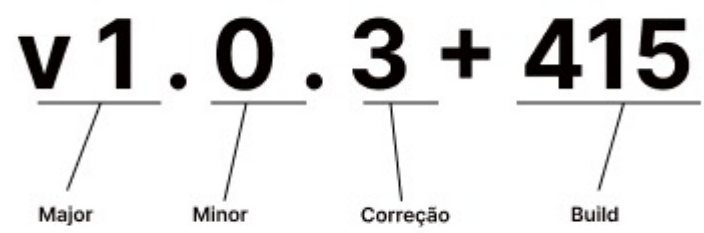
Mas quando você for adicionar as tags, vai saber a diferença de cada número?

Vou te explicar a diferença de forma simples e prática:

- **Major:** São alterações incompatíveis com as versões anteriores (Reestruturou tudo? Adicione +1 - v2.0.0)
- **Minor:** São alterações importantes e compatíveis com a versão anterior (Adicionou uma nova funcionalidade? Adicione +1 v2.1.0)
- **Correção:** Erros e bugs que não afetam a versão (Encontrou um bug e corrigiu? Adicione +1 v2.1.1)
- **Build:** É um controle interno do Git e do versionamento, ele não fica necessariamente à vista quando você especifica a versão.

**Exemplo:**

**v 1 . 0 . 3 + 415**



The diagram illustrates the components of a semantic version number. The text 'v 1 . 0 . 3 + 415' is displayed in a large, bold font. Below each of the four main parts—'v', '1', '0', and '3 + 415'—there is a horizontal line. A diagonal line extends from the end of each horizontal line down to a label. The labels are 'Major' under 'v', 'Minor' under '1', 'Correção' under '0', and 'Build' under '3 + 415'.

Major      Minor      Correção      Build

# Introdução ao Markdown

Markdown é uma linguagem de marcação leve usada para formatar texto de forma simples e fácil de ler.

Criada por John Gruber e Aaron Swartz em 2004, seu objetivo é ser legível em texto plano e, ao mesmo tempo, permitir a conversão para HTML de forma automática. Markdown é amplamente utilizado para criar documentos, escrever posts em blogs, e produzir `README.md` em repositórios de código.

## Cabeçalhos:

- Os cabeçalhos são criados usando o símbolo `#`. O número de `#` indica o nível do cabeçalho.

```
# Cabeçalho Nível 1
## Cabeçalho Nível 2
### Cabeçalho Nível 3
```

## Listas:

- Markdown suporta listas ordenadas e não ordenadas.

```
- Item 1
- Item 2
  - Subitem 2.1
  - Subitem 2.2

1. Primeiro item
2. Segundo item
  1. Subitem 2.1
  2. Subitem 2.2
```

## Links:

- Para criar links, use colchetes para o texto do link e parênteses para a URL.

[Link para o meu site](https://www.exemplo.com)

### Imagens:

- As imagens são inseridas de forma semelhante aos links, mas com um ponto de exclamação ! antes.

![Logo do Projeto](imagens/logo.png)

### Ênfase (negrito e itálico):

- Para criar ênfase em texto, você pode usar asteriscos ou sublinhados.

Este é um **projeto incrível** que usa tecnologias modernas.

### Citações:

- Para criar uma citação, use o símbolo >.

> "Seja a sua melhor versão." - Einstein

### Tabelas:

- Tabelas podem ser criadas usando | para separar colunas e - para criar a linha do cabeçalho.

Nome	Função
João	Desenvolvedor
Maria	Designer

### Código:

- Para incluir blocos de código, utilize três acentos graves antes e depois do bloco. Você pode especificar a linguagem de programação para realce de sintaxe.

○

Markdown	HTML	Rendered Output
<code>``Use `code` in your Markdown file.``</code>	<code>&lt;code&gt;Use `code` in your Markdown file.&lt;/code&gt;</code>	Use `code` in your Markdown file.



# Trabalhando com Branches

# Criando e Gerenciando Branches Locais

## 1. Criar uma nova branch:

```
git branch <nome-da-branch>
```

## 2. Mudar para uma branch:

```
git checkout <nome-da-branch>
```

## 3. Criar e mudar para uma nova branch:

```
git checkout -b <nome-da-branch>
```

## 4. Listar todas as branches:

```
git branch
```

## 5. Renomear uma branch:

```
git branch -m <nome-da-branch>
```

## 6. Excluir uma branch:

```
git branch -d <nome-da-branch>
```

## Mesclando Branches (Merge)

### 1. Mesclar uma branch na branch atual:

```
git merge <nome-da-branch>
```

## Resolução de Conflitos de Merge

Durante um merge, se houver conflitos, o Git irá pausar o processo e te informar sobre os arquivos conflitantes. Edite esses arquivos para resolver os conflitos, marcados com:

```
<<<<<<< HEAD
(mudanças na branch atual)
=====
(mudanças na branch que está sendo mesclada)
>>>>>>> <nome-da-branch>
```

Depois de resolver os conflitos, marque os arquivos como resolvidos:

```
git add .
```

Finalize o merge:

```
git commit -m "Arquivos conflitantes resolvidos."
```

## Branches Remotos e Rastreamento de Branches

### 1. Listar branches remotos:

```
git branch -r
```

### 2. Criar uma branch rastreando uma branch remota:

```
git checkout --track origin/<nome-da-branch-remota>
```

- Só precisa realizar o rastreo se a branch remota não existir em seu repositório local.

### **3. Atualizar branches remotos:**

```
git fetch
```

### **4. Mesclar mudanças de uma branch remota na branch atual:**

```
git pull origin <nome-da-branch-remota>
```

```
# Ou apenas
```

```
git pull
```

# Fluxo de Trabalho com Branches (Feature Branch, Hotfix e Release.)

## 1. Feature Branch:

- Criar uma nova branch para a feature:

```
git checkout -b feature/<nome-da-feature>
```

- Trabalhar na feature e fazer commits.
- Mesclar a branch da feature na branch principal (geralmente **main** ou **dev**):

```
git checkout main  
git merge feature/<nome-da-feature>
```

## 2. Hotfix Branch:

- Criar uma nova branch para o hotfix a partir da branch principal:

```
git checkout -b hotfix/<nome-do-hotfix> main
```

- Trabalhar no hotfix e fazer commits.
- Mesclar a branch do hotfix na branch principal:

```
git checkout main  
git merge hotfix/<nome-do-hotfix>
```

- Mesclar a branch do hotfix na branch de **dev** (se houver):

```
git checkout dev
git merge hotfix/<nome-do-hotfix>
```

### 3. Release Branch:

- Criar uma nova branch para a release a partir da branch de desenvolvimento:

```
git checkout -b release/<versao> <nome-da-branch>
```

# Código

```
git checkout -b release/1.0.0 dev
```

- Testar e preparar a release, fazendo commits conforme necessário.
- Mesclar a branch da release na branch principal e taguear a versão:

```
git checkout main
git merge release/<versao>
git tag -a v<versao> -m "Versão <versao>"
```

# Código

```
git checkout main
git merge release/1.0.0
git tag -a v1.0.0 -m "Versão 1.0.0"
```

- Mesclar a branch da release de volta na branch de **dev**:

```
git checkout dev
git merge release/<versao>
```

# Sincronização e Colaboração

# Configurando Repositórios Remotos

## 1. Adicionar um repositório remoto:

```
git remote add origin <url-do-repositório-remoto>
```

- Substitua **<url-do-repositório-remoto>** pelo URL do seu repositório remoto (por exemplo, no GitHub, GitLab ou Bitbucket).

## 2. Verificar os repositórios remotos configurados:

```
git remote -v
```

## 3. Alterar a URL de um repositório remoto:

```
git remote set-url origin <nova-url-do-repositório-remoto>
```

## 4. Remover um repositório remoto:

```
git remote remove origin
```



## Enviando Mudanças para o Repositório Remoto (**git push**)

### 1. Enviar mudanças para a branch principal (main):

```
git push origin main
```

### 2. Enviar uma branch específica para o repositório remoto:

```
git push origin <nome-da-branch>
```

### 3. Enviar todas as tags locais para o repositório remoto:

```
git push --tags
```

## Obtendo Mudanças do Repositório Remoto (`git pull` e `git fetch`)

**1. Atualizar o repositório local com mudanças do repositório remoto e mesclar automaticamente:**

```
git pull origin main
```

- Isso é equivalente a `git fetch` seguido de `git merge`.

**2. Obter mudanças do repositório remoto sem mesclar:**

```
git fetch origin
```

**3. Mesclar mudanças obtidas do repositório remoto:**

```
git merge origin/main
```

# Trabalhando com Forks e Pull Requests

## 1. Criar um fork de um repositório:

- No GitHub, GitLab ou Bitbucket, use a interface web para criar um fork do repositório desejado.

## 2. Clonar o repositório forkado:

```
git clone <url-do-repositório-forkado>
```

## 3. Adicionar um repositório upstream para sincronizar com o repositório original:

```
git remote add upstream <url-do-repositório-original>
```

- O termo "**upstream**" é utilizado para se referir ao repositório remoto que é a **fonte oficial de um projeto**. É o repositório principal do qual você geralmente deseja sincronizar as atualizações.

## 4. Sincronizar o fork com o repositório original:

- Buscar mudanças do repositório original:

```
git fetch upstream
```

- Mesclar mudanças na branch principal:

```
git checkout main  
git merge upstream/main
```

- Enviar mudanças para o seu fork no GitHub:

```
git push origin main
```

## 5. Criar uma pull request:

- No GitHub, GitLab ou Bitbucket, **use a interface web para criar uma pull request do seu fork para o repositório original.** Descreva as mudanças que você fez e solicite a revisão.

# Exemplo de Fluxo Completo

## 1. Adicionar Repositórios Remotos:

```
git remote add origin <url-do-seu-fork>
git remote add upstream <url-do-repositório-original>
```

## 2. Fazer Mudanças e Comitar:

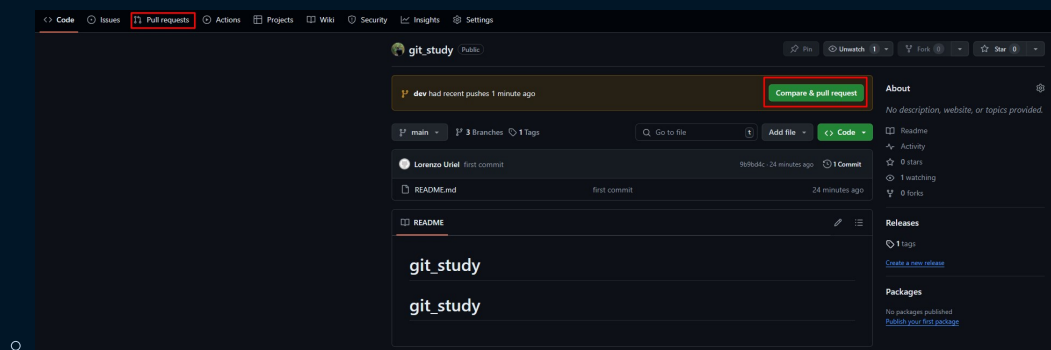
```
git add .
git commit -m "Minha contribuição"
```

## 3. Enviar Mudanças para o Fork:

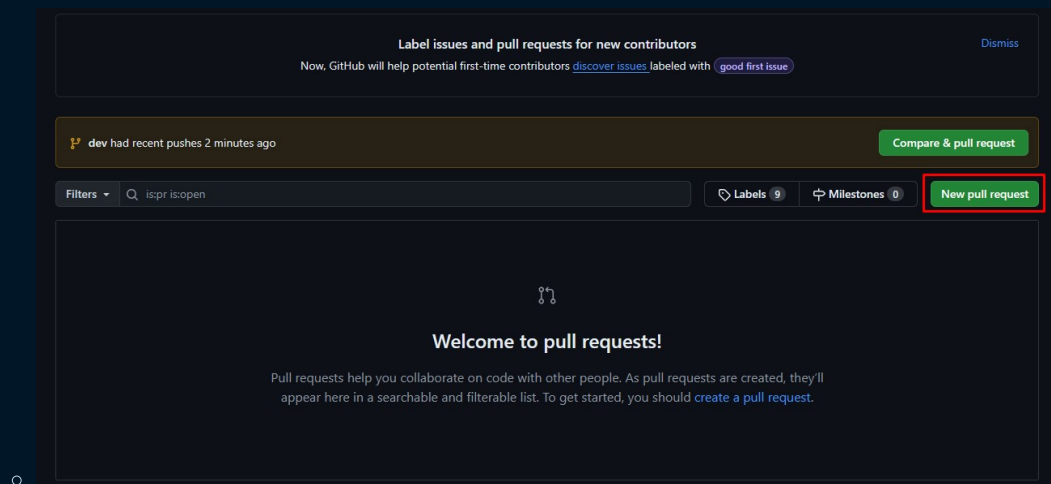
```
git push origin main
```

## 4. Criar uma Pull Request:

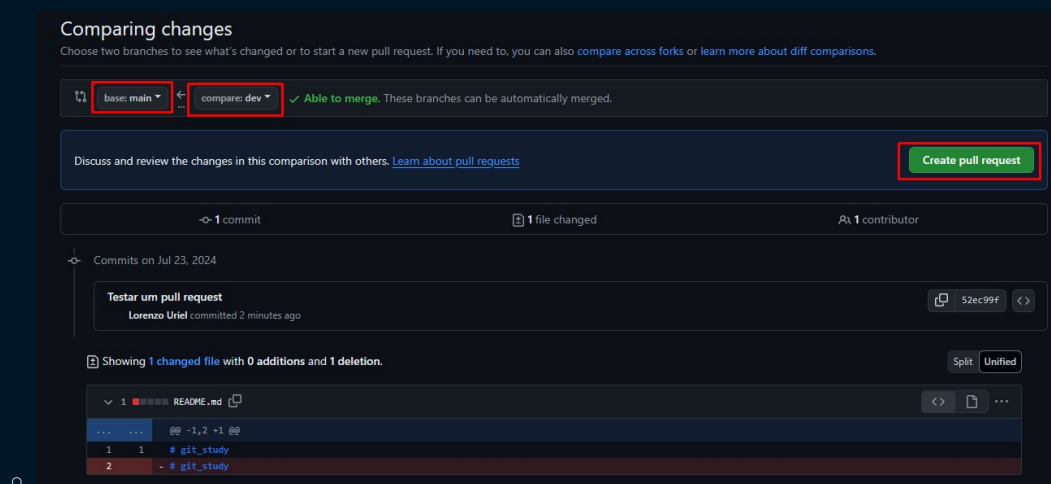
- No GitHub, vá para o repositório original e clique em "New Pull Request".



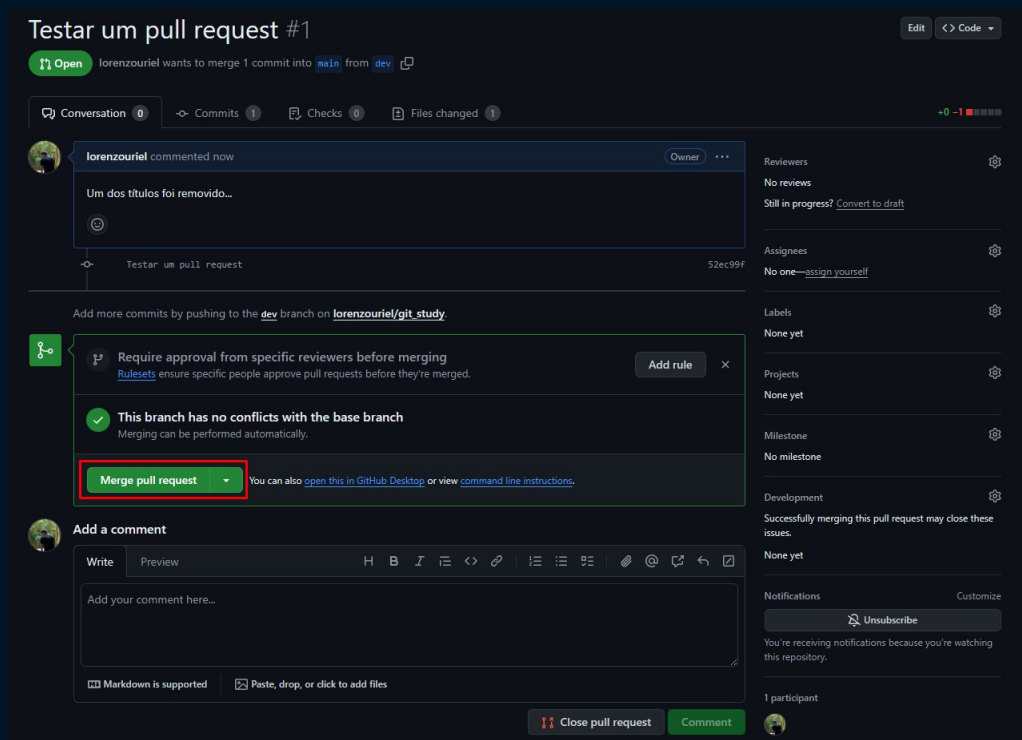
- Na aba **Pull Request**, clique em **New Pull Request**



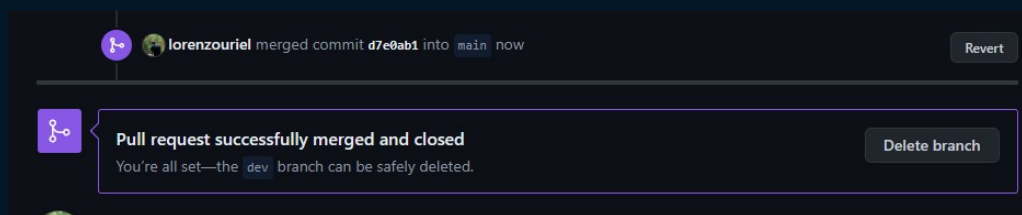
- Selecione a branch que possui alterações (**compare:dev**) e a que deseja fazer o merge (**base:main**). Clique em **Create Pull Request**



- O GitHub vai navegar para essa aba e testar novamente se as alterações não possuem conflitos, depois da verificação clique em **Merge pull request**



- Caso possua essa cultura, já pode deletar a branch direto na etapa de **Pull Request**



## 5. Sincronizar com o Repositório Original:

- Após aplicar as alterações, você precisa enviar para o seu repositório local:

```
git fetch upstream
git checkout main
git merge upstream/main
git push origin main
```

- Depois, pode deletar a branch do repositório local:

```
git checkout -d dev
```





# Gerenciamento de Repositórios

# Removendo Repositórios Locais

## 1. Remover um repositório localmente:

- Para excluir um repositório Git localmente, você pode simplesmente excluir o diretório onde ele está localizado. Por exemplo:

```
# bash
rm -rf /caminho/para/seu/repositório
git commit -m "Remove tests_git directory"
git push

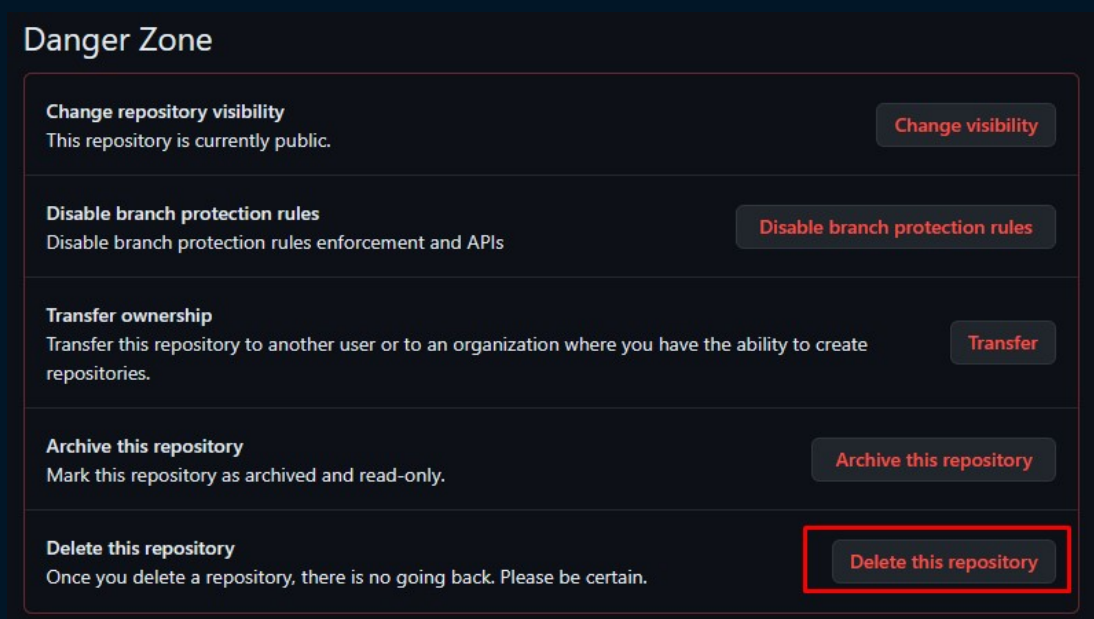
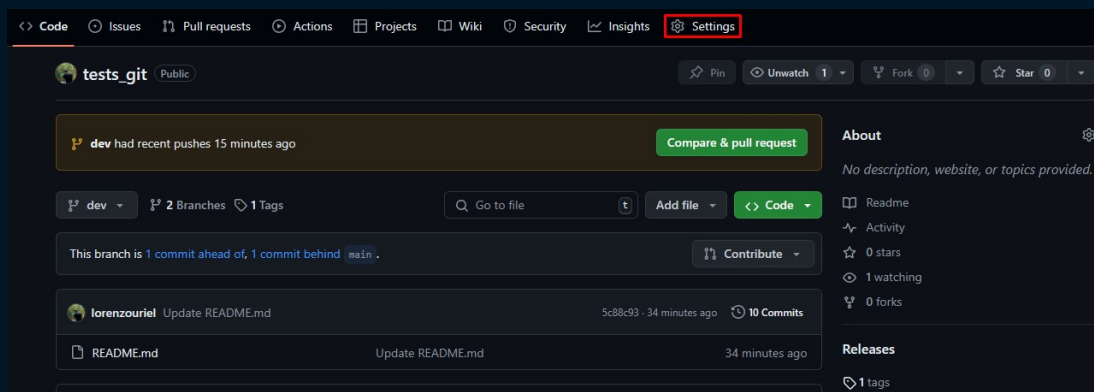
# power shell
git rm -r --cached /projects/portfolio/tests_git
Remove-Item -Path "C:\Projects\portfolio\tests_git" -Recurse -Force
git commit -m "Remove tests_git directory"
git push
```

- Use este comando com cautela, pois ele excluirá permanentemente todos os arquivos e o histórico de commits no repositório.

# Excluindo Repositórios Remotos no GitHub

Vá para a página do repositório no serviço de hospedagem.

Nas configurações do repositório, procure a opção para excluir o repositório.



Confirme a exclusão com a senha ou acessando o GitHub Mobile.

# Limpeza de Histórico (**Rebase**, **Squash**)

## 1. Rebase:

O rebase é usado para reescrever o histórico de commits.

O Git Rebase move um branch para o topo de outro, reescrevendo o histórico no processo. Ao invés de realizar um merge ele reescreve o histórico.

Por exemplo, para rebasear sua branch **dev** na branch **main**:

```
git checkout dev  
git rebase main
```

A ideia do rebase é criar um histórico de commits mais linear, facilitando futuros reviews de código.

## 2. Interactive Rebase:

Para combinar ou reordenar commits, use o rebase interativo:

```
git rebase -i HEAD~n
```

- Onde **n** é o número de commits que você deseja revisar. Você verá uma lista de commits que pode editar, combinar (squash) ou reordenar.

Quando rodar o rebase, seu terminal vai retornar dessa maneira:

```

pick f7f3f6d Meu commit 01
edit 310154e Meu commit 02
squash a5f4a0d Meu commit 03
drop a5f4a0d Meu commit 04

# Rebase 9b9bd4c..9b9bd4c onto 9b9bd4c (1 command)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                               commit's log message, unless -C is used, in which
case
#                               keep only this commit's message; -c is same as -C
but
#                               opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --
continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
#       create a merge commit using the original merge commit's
#       message (or the oneline, if no original merge commit was
#       specified); use -c <commit> to reword the commit message
# u, update-ref <ref> = track a placeholder for the <ref> to be updated
#                       to this position in the new commits. The <ref>
is
#                       updated at the end of the rebase
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
~

```

Cada comando especifica exatamente o que você pode fazer com os commits.

Os seus commits vão aparecer acima dos comandos, basta você selecionar a linha do

commit desejado e clicar apenas na letra inicial de cada comando. O comando vai adicionar automaticamente no início do commit.

### O que o script completo faz?

- Aplica o commit `f7f3f6d` como está.
- Aplica o commit `310154e` e para para permitir edição.
- Combina o commit `a5f4a0d` com o commit `310154e` anterior.
- Remove o commit `a5f4a0d` do histórico.

### Isso resulta em:

- O primeiro commit (`f7f3f6d`) sendo aplicado normalmente.
- O segundo commit (`310154e`) sendo aplicado e editado conforme necessário.
- O terceiro commit (`a5f4a0d`) sendo combinado (`squash`) com o segundo commit.
- O quarto commit (`a5f4a0d`) sendo removido do histórico.

### Observações adicionais

- O comando `squash` vai pedir para editar a mensagem do commit resultante.
- O `squash` combina vários commits em um único commit.
- O comando `edit` vai interromper o rebase para permitir modificações no commit.
- O comando `drop` remove completamente o commit do histórico.

# Tags e Releases

## 1. Criar uma Tag:

- Tags são usadas para marcar pontos específicos na história do commit, geralmente usadas para marcar versões de release.

```
git tag <nome-da-tag>
```

**2. Criar uma Tag Anotada:** Tags anotadas são recomendadas para releases porque contêm informações adicionais.

```
git tag -a <nome-da-tag> -m "Mensagem da tag"
```

**3. Enviar uma Tag para o Repositório Remoto:** Sempre que criar um tag você precisa enviar para o repositório remoto.

```
git push origin <nome-da-tag>
```

## 4. Enviar Todas as Tags Locais para o Repositório Remoto:

```
git push --tags
```

## 5. Listar Todas as Tags:

```
git tag
```

## 6. Excluir uma Tag Localmente:

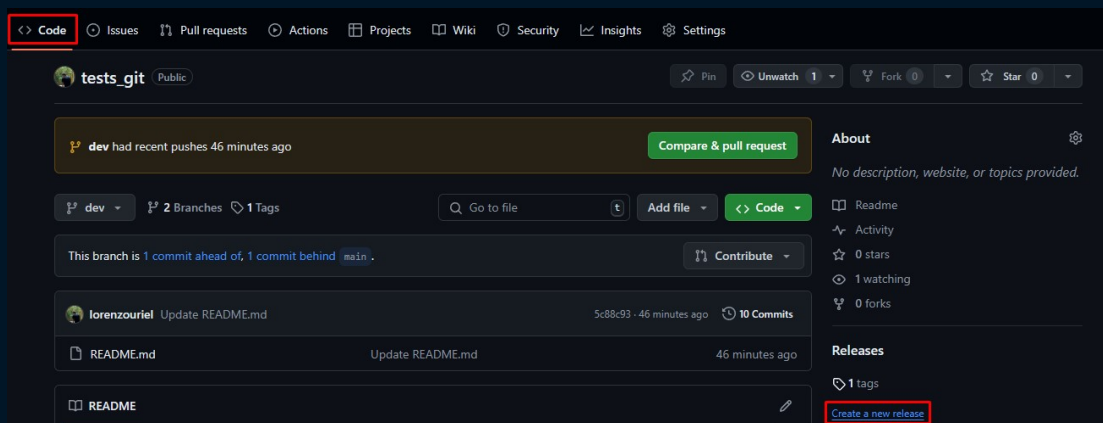
```
git tag -d <nome-da-tag>
```

## 7. Excluir uma Tag no Repositório Remoto:

```
git push origin --delete <nome-da-tag>
```

## 8. Criar uma Release no GitHub:

- No GitHub, vá para a página do repositório.
- Vá para a seção "Code" > "Create a New Release".



- Preencha as informações, adicione a tag e clique em "Publish a New Release":



ReleasesTags

v1.0.0

Previous tag: auto

Generate release notes

Existing tag

Primeiro release de um projeto

WritePreview

H

B

I

≡

<>

↗

≡

≡

≡

🔗

@

↶

Esse release está sendo feito para fins educativos

Markdown is supported

Paste, drop, or click to add files

↓ Attach binaries by dropping them here or selecting them.

☐ Set as a pre-release
 

This release will be labeled as non-production ready

Publish release

Save draft

- Suas releases ficam organizados na seção "Code"

tests\_gjtPublic

Pin

Unwatch1

Fork0

Star0

dev had recent pushes 49 minutes ago

Compare & pull request

v1.0.02 Branches1 Tags

Go to file

Code

Lorenzo Uriel

Commit da versão 01 indo

2634918 · yesterday

9 Commits

README.md

Commit da versão 01 indo

yesterday

README

Vamos iniciar

About

No description, website, or topics provided.

Readme

Activity

0 stars

1 watching

0 forks

Releases1

Primeiro release de um projeto

Latest

now

# Práticas Recomendadas

# Commits Claros e Frequentes

## Commits Frequentes:

- Faça commits pequenos e frequentes para facilitar a identificação de problemas e a reversão de mudanças específicas.
- Cada commit deve representar uma única mudança lógica no código.
- Toda alteração deve ter um commit.

## Mensagens de Commit Claras:

- Use mensagens de commit descritivas e significativas.
- A linha deve ser um resumo curto e conciso (50 caracteres ou menos), pode adicionar uma descrição mais detalhada, se necessário.

```
git commit -m "Corrige bug na função de login"
git commit -m "Adiciona testes unitários para o módulo de autenticação"
```

## Uso de `.gitignore`

### Criar e Usar o Arquivo `.gitignore`:

- O arquivo `.gitignore` é usado para especificar quais arquivos e diretórios devem ser ignorados pelo Git.
- Exemplos comuns incluem arquivos de configuração local, diretórios de build, e arquivos temporários.
- Você pode usar o site [toptal](https://toptal.com) para buscar arquivos `.gitignore` completos e preenchidos para a respectiva linguagem.

### Exemplo de um arquivo `.gitignore`:

```
# Logs
*.log

# Diretórios de build
/build/
/dist/

# Arquivos de configuração
.env
.vscode/

# Arquivos temporários
*.tmp
```

- Para criar ou editar o arquivo `.gitignore`, adicione-o na raiz do seu repositório.

# Reposicionamento de Branches

## Rebase em vez de Merge:

Use `git rebase` para aplicar commits de uma branch no topo de outra, criando um histórico mais linear.

```
git checkout feature-branch  
git rebase main
```

- O comando `git rebase` é usado para integrar mudanças de um ramo em outro, mas de uma forma diferente do comando `git merge`. Enquanto o `merge` cria um novo commit de `merge`, o `rebase` reaplica os commits de um ramo sobre a ponta de outro ramo, resultando em um histórico linear.

Resolve conflitos se necessário e continue com:

```
git rebase --continue
```

## Mesclar Branches:

Use `git merge` para combinar mudanças de uma branch em outra, preservando o histórico de commits.

```
git checkout main  
git merge feature-branch
```

# Manutenção de Histórico Limpo e Compreensível

## Rebase Interativo para Limpar Histórico:

Use `git rebase -i` para reordenar, editar, combinar (squash) ou descartar commits.

```
git rebase -i HEAD~n
```

- No editor que se abre, altere `pick` para `squash` para combinar commits.

Evite commits de merge desnecessários usando `git pull --rebase` em vez de `git pull`.

# Automação de Tarefas com Hooks do Git

## Configurar Hooks do Git:

Os hooks do Git são scripts que são executados automaticamente em determinados eventos, como antes de um commit ou após um push.

Exemplos de hooks incluem **pre-commit**, **pre-push**, **post-commit**, entre outros.

Exemplo de um hook **pre-commit** para verificar o estilo de código:

```
@echo off
# Pre-commit hook para rodar pytest

echo Running pytest
pytest

# se pytest falhar, não faça o commit
if errorlevel 1 (
    echo Tests failed
    exit /b 1
)
```

Coloque os scripts de hook no diretório **.git/hooks/** do seu repositório e torne os scripts executáveis:

```
chmod +x .git/hooks/pre-commit
```

## Resumo

- Commits claros e frequentes facilitam a colaboração e o gerenciamento de código.
  - Uso de `.gitignore` mantém o repositório limpo de arquivos desnecessários.
  - Reposicionamento de branches com rebase e merge adequados mantém o histórico legível e linear.
  - Manutenção de histórico limpo com ferramentas como rebase interativo evita complexidade desnecessária.
  - Automação de tarefas com hooks do Git assegura a qualidade do código e a consistência do repositório.
-



# Solução de Problemas

## Desfazendo Commits (`git revert`, `git reset`, `git checkout`)

### `git revert`

Usado para **criar um novo commit que desfaz as mudanças de um commit específico, preservando o histórico.**

```
git revert <commit-hash>
```

Isto cria um novo commit que desfaz as mudanças do commit especificado.

Assim fica o seu log, após o `revert`:

```

loren@ MINGW64 /c/Projects/portfolio/git_study (dev)
$ git revert 0d3bd1def0f408e07b1249d7c599759b46380640
hint: Waiting for your editor to close the file...
[No write since last change]

Press ENTER or type command to continue
[No write since last change]

Press ENTER or type command to continue
[dev 68feb15] Revert "Commit para reverter"
 1 file changed, 1 insertion(+), 3 deletions(-)

loren@ MINGW64 /c/Projects/portfolio/git_study (dev)
$ git log
commit 68feb159e81c636e290febc9bbe0d94ac8a15ccb (HEAD -> dev)
Author: Lorenzo Uriel <lorenzo.uriel@com.br>
Date: Thu Jul 25 22:05:30 2024 -0300

    Revert "Commit para reverter"

    This reverts commit 0d3bd1def0f408e07b1249d7c599759b46380640.

commit 0d3bd1def0f408e07b1249d7c599759b46380640
Author: Lorenzo Uriel <lorenzo.uriel@com.br>
Date: Thu Jul 25 22:05:02 2024 -0300

    Commit para reverter

```

Eu fiz um **commit** inicial com a mensagem **Commit para reverter**, logo depois eu peguei o hash do commit e fiz um **git revert 0d3bd1def0f408e07b1249d7c599759b46380640** com a mensagem **Revert "Commit para reverter"**.

## git reset

Usado para **redefinir o estado do repositório para um commit anterior**. O comando **reset** pode alterar o histórico.

Principais modos de **git reset**:

- **--soft**: Mantém os arquivos e indexação como estão, apenas reseta o ponteiro HEAD

```
git reset --soft <commit-hash>
```

- **--mixed**: Mantém os arquivos no diretório de trabalho, mas desfaz as mudanças do index.

```
git reset --mixed <commit-hash>
```

- **--hard**: Reseta o ponteiro HEAD, o index e o diretório de trabalho.

```
git reset --hard <commit-hash>
```

## git checkout

Usado para **mudar branches ou restaurar arquivos no diretório de trabalho**. O **checkout** não afeta o histórico.

```
git checkout <branch-name>  
git checkout <commit-hash> -- <file-path>
```

# Resolução de Conflitos Comuns

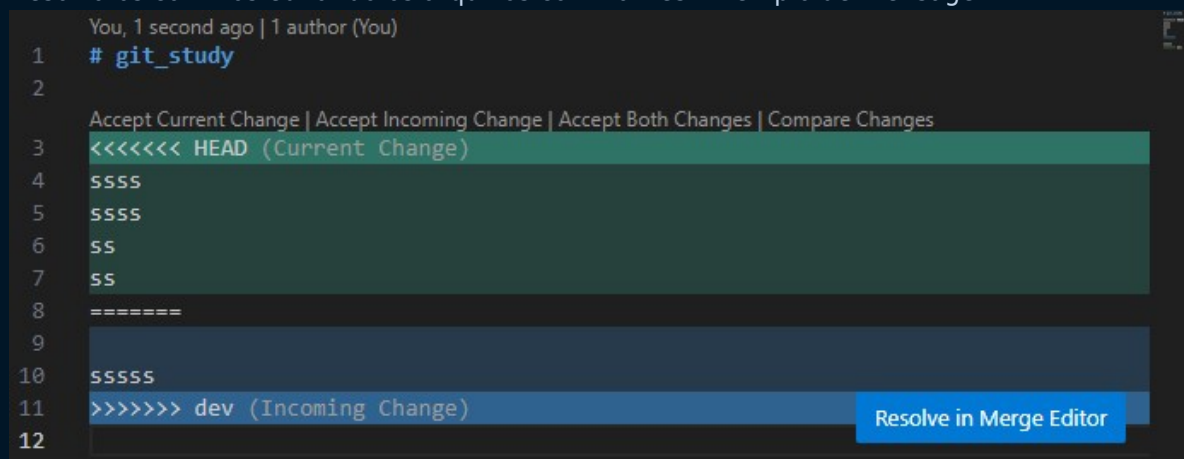
## Conflitos de Merge

Quando um conflito de merge ocorre, o Git marca os arquivos conflitantes para que você possa resolvê-los manualmente.

### Exemplo:

```
git merge <branch-name>
```

Resolva os conflitos editando os arquivos conflitantes. Exemplo de mensagem:



```
You, 1 second ago | 1 author (You)
1  # git_study
2
3  Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
4  <<<<<< HEAD (Current Change)
5  sssss
6  sssss
7  ss
8  ss
9  =====
10 sssss
11 >>>>>> dev (Incoming Change)
12
```

```
loren@ MINGW64 /c/Projects/portfolio/git_study (main)
$ git merge dev
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

- Marque os conflitos como resolvidos:

```
git add <file-path>
```

- Complete o merge:

```
git commit -m "Erros corrigidos, pronto para seguir"
```

## Conflitos de Rebase

Durante o rebase, conflitos são resolvidos de maneira semelhante aos conflitos de merge.

### Exemplo:

```
git rebase <branch-name>
```

- Resolva os conflitos editando os arquivos conflitantes.
- Continue o rebase:

```
git rebase --continue
```

- Para abortar o rebase:

```
git rebase --abort
```

# Recuperação de Commits Perdidos

## git reflog

Mantém um registro de todos os **HEADs** que foram alterados no repositório local.

É muito utilizado para commits perdidos.

### Exemplo:

```
git reflog
```

- Identifique o commit que você deseja recuperar e use `git checkout` ou `git reset` para restaurá-lo.

Exemplo de retorno no terminal:

```
$ git reflog
76e8ecf (HEAD -> main) HEAD@{0}: commit (merge): Erros ok
3fd367d HEAD@{1}: commit: Vou fazer um merge
7ce5614 (origin/main) HEAD@{2}: checkout: moving from dev to main
1dbfda0 (dev) HEAD@{3}: checkout: moving from dev to dev
1dbfda0 (dev) HEAD@{4}: reset: moving to
1dbfda0a8363529fdbb0abb2023f7ee8f5d43691
1dbfda0 (dev) HEAD@{5}: reset: moving to
1dbfda0a8363529fdbb0abb2023f7ee8f5d43691
1dbfda0 (dev) HEAD@{6}: reset: moving to
1dbfda0a8363529fdbb0abb2023f7ee8f5d43691
1dbfda0 (dev) HEAD@{7}: commit: Commit para dar um reset
68feb15 HEAD@{8}: revert: Revert "Commit para reverter"
0d3bd1d HEAD@{9}: commit: Commit para reverter
7ce5614 (origin/main) HEAD@{10}: rebase (finish): returning to
refs/heads/dev
7ce5614 (origin/main) HEAD@{11}: rebase (start): checkout HEAD
7ce5614 (origin/main) HEAD@{12}: rebase (finish): returning to
refs/heads/dev
7ce5614 (origin/main) HEAD@{13}: rebase (start): checkout HEAD
7ce5614 (origin/main) HEAD@{14}: rebase (finish): returning to
refs/heads/dev
7ce5614 (origin/main) HEAD@{15}: rebase (start): checkout HEAD
7ce5614 (origin/main) HEAD@{16}: rebase (finish): returning to
```

```
refs/heads/dev
7ce5614 (origin/main) HEAD@{17}: rebase (start): checkout HEAD
7ce5614 (origin/main) HEAD@{18}: rebase (finish): returning to
refs/heads/dev
7ce5614 (origin/main) HEAD@{19}: rebase (start): checkout main
39f9d27 HEAD@{20}: checkout: moving from main to dev
7ce5614 (origin/main) HEAD@{21}: rebase (finish): returning to
refs/heads/main
7ce5614 (origin/main) HEAD@{22}: rebase (start): checkout HEAD
7ce5614 (origin/main) HEAD@{23}: commit (merge): Commit 04
39f9d27 HEAD@{24}: rebase (finish): returning to refs/heads/main
39f9d27 HEAD@{25}: rebase (start): checkout HEAD
39f9d27 HEAD@{26}: checkout: moving from dev to main
39f9d27 HEAD@{27}: checkout: moving from main to dev
39f9d27 HEAD@{28}: checkout: moving from dev to main
39f9d27 HEAD@{29}: rebase (finish): returning to refs/heads/dev
39f9d27 HEAD@{30}: rebase (start): checkout main
52ec99f (origin/dev) HEAD@{31}: rebase (finish): returning to
refs/heads/dev
52ec99f (origin/dev) HEAD@{32}: rebase (start): checkout HEAD
52ec99f (origin/dev) HEAD@{33}: checkout: moving from main to dev
```

## git cherry-pick

Usado para **aplicar um commit específico de uma branch para outra.**

### Exemplo:

```
git cherry-pick <commit-hash>
```



## Dicas de Troubleshooting

### Diagnóstico de Problemas

Use comandos como `git status`, `git log`, e `git diff` para entender o estado atual do repositório e identificar problemas.

- `git status`: Fornece uma visão geral do estado atual do seu repositório. Ele exibe quais arquivos foram modificados, quais estão prontos para o commit e quais não estão rastreados. Isso ajuda a entender rapidamente o que mudou desde o último commit.
- `git log`: Usado para visualizar o histórico de commits. Isso é útil para identificar mudanças feitas ao longo do tempo e entender como o repositório evoluiu. Você pode adicionar opções como `--oneline` para uma visualização mais concisa ou `--graph` para uma representação gráfica dos commits.
- `git diff`: Compara alterações entre commits, entre o repositório e o diretório de trabalho, ou entre diferentes branches. É útil para revisar alterações específicas em arquivos e entender como o código foi modificado.

### Ignorar Arquivos Localmente

Se você precisar ignorar mudanças em arquivos que estão rastreados, use

```
git update-index --assume-unchanged <file-path>
```

Para desfazer:

```
git update-index --no-assume-unchanged <file-path>
```

### Descartar Mudanças Locais

Para descartar mudanças em arquivos modificados:

```
git checkout -- <file-path>
```

Para descartar todas as mudanças locais:

```
git reset --hard
```

## Resolver Problemas de Push/Pull

Se você encontrar problemas ao fazer push ou pull, como rejeições de push, primeiro faça fetch para sincronizar seu repositório:

```
git fetch
```

- Resolva quaisquer conflitos, então tente novamente
-

# Ferramentas e Integrações

# Integração do Git com IDEs Populares

## 1. VSCode:

- VSCode possui integração nativa com Git. Você pode inicializar repositórios, realizar commits, pull, push, visualizar históricos de commits e resolver conflitos diretamente pelo editor.
- Existem várias extensões como **GitLens**, que aprimoram a experiência de uso do Git no VSCode, oferecendo funcionalidades como visualização de autorias de linhas, comparação de branches e mais.

## 2. IntelliJ:

- IntelliJ oferece integração robusta com Git, permitindo gerenciamento de repositórios, commits, branches, merges, e mais, diretamente pelo IDE.
- Ferramentas gráficas para comparação de arquivos e resolução de conflitos de merge são integradas e bastante intuitivas.

# Uso de Ferramentas Gráficas para Git

## 1. GitKraken:

- Oferece uma interface visual clara para o gerenciamento de repositórios Git, com funcionalidades como visualização de commits, criação e gerenciamento de branches, e resolução de conflitos.
- Suporte a integrações com várias plataformas de hospedagem de repositórios como GitHub, GitLab, Bitbucket, entre outras.

## 2. SourceTree:

- Permite clonar, criar, e gerenciar repositórios locais e remotos com uma interface gráfica intuitiva.
- Fornece uma visualização detalhada do histórico de commits e branches, facilitando o acompanhamento de alterações.

# Integração com CI/CD

## 1 Jenkins:

- Jenkins pode ser configurado para iniciar builds automaticamente em eventos Git como commits e pull requests.
- Existem plugins específicos para integração com Git, que facilitam o monitoramento de repositórios e execução de pipelines.

## 2. GitHub Actions:

- Permite a criação de workflows automatizados que são disparados por eventos Git (push, pull request, etc.).
- Workflows são configurados usando arquivos YAML, oferecendo grande flexibilidade e controle sobre os processos de CI/CD.

## 3. GitLab CI:

- Define pipelines CI/CD diretamente no repositório com arquivos `.gitlab-ci.yml`.
- Integração nativa e profunda com o repositório GitLab, oferecendo recursos avançados de CI/CD como build, test, e deploy automáticos.

# GitFlow

GitFlow é um modelo de branching que define um processo de desenvolvimento organizado e eficiente.

## Branches principais:

- **main**: Contém o código de produção.
- **develop**: Contém o código para a próxima versão que será lançado.

## Branches de suporte:

- **feature/\***: Usado para desenvolvimento de novas funcionalidades.
- **release/\***: Preparação de uma nova versão de produção.
- **hotfix/\***: Correções urgentes em produção.