

UM GUIA GIT & GITHUB



Criado por: Lorenzo Uriel

Table of Contents

Sejam Bem-Vindos!	5
Sobre o Autor	6
Ebook PDF	7
 Um Guia para Começar no Git & GitHub	 8
O que é Git?	9
Instalação e Configuração Inicial do Git	10
O que é GitHub?	12
Autenticações do GitHub	13
 Um Guia para Conceitos Fundamentais	 16
Introdução a Repositórios e Conceitos	17
Conceitos Principais	18
Estrutura do Repositório	19
Comandos Principais	20
Criando um Repositório Local e Remoto	23
O que é Versionamento e Tags?	26
Introdução ao Markdown	27
 Um Guia para Trabalhar com Branchs	 30
 O que é uma Git Branch?	 31
 Por que usar Branches?	 32
 Criando e Gerenciando Branches Locais	 33

Fazendo Merge de Branches	36
Fazendo Rebase de Branches	39
Branches Remotas e Rastreamento de Branches	43
Guardando Alterações (Stashing)	46
Workflow com Branches (Feature Branch, Hotfix e Release)	49
Selecionando Commits (Cherry-Picking)	52
Desfazendo Alterações	54
 Um Guia para Sincronização e Colaboração	57
Configurando Repositórios Remotos	58
Enviando Alterações para o Repositório Remoto (git push)	60
Obtendo Alterações do Repositório Remoto (git pull e git fetch)	61
Removendo Repositórios Locais e Remotos	62
Exemplo Completo de Fluxo	63
 Trabalhando com Tags e Releases	65
Trabalhando com Forks e Pull Requests	69
Mais Sobre o Comando git remote	71
 Um Guia para Solução de Problemas	74
Desfazendo Commits (git revert, git reset, git checkout)	75
git revert	76
git reset	78
git checkout	80
Recuperando Commits Perdidos	81
Mais Sobre git status, git log, git show e git diff	87
Mais Sobre git commit --amend	93
 Um Guia para Boas Práticas	95
Commits Claros e Frequentes	96

Gerenciamento de Branch	98
Use .gitignore	100
Git Hooks	101
Um Guia para Tipos de Workflows	102
Trunk-Based Development	103
GitFlow	105
Feature Branch	109
Forking Workflow	110
O Fim	112

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

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.

Um Guia para Começar no Git & GitHub

Quem são eles? Onde vivem? O que comem? E, o melhor de tudo, como instalar?

O que é Git?

Git é um **sistema de controle de versão** projetado para rastrear mudanças 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 alterações no código-fonte, permite que múltiplos branches de desenvolvimento existam simultaneamente e facilita a mesclagem de código de diferentes contribuidores.

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

Instalação e Configuração Inicial do Git

Instalando o Git

Windows

1. Baixe o Git em: <https://git-scm.com/downloads>
2. Execute o instalador e selecione as opções padrão
3. Abra o Git Bash e verifique a instalação:

```
git --version
```

macOS:

1. Instale o Git via Homebrew:

```
brew install git
```

2. Verifique a instalação:

```
git --version
```

Linux (baseado em Debian):

```
sudo apt update && sudo apt install git  
git --version
```

Configuração Inicial do Git

Após instalar o Git, configure sua identidade:

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

Verifique suas configurações com:

```
git config --list
```

Essas configurações garantem que seus commits sejam corretamente atribuídos.

O que é GitHub?

GitHub é uma plataforma para controle de versão e colaboração usando Git. Ele permite que vários desenvolvedores trabalhem no mesmo projeto, rastreiem mudanças e gerenciem repositórios de código de forma eficiente.

Com o GitHub, você pode:

- Hospedar e compartilhar repositórios
- Colaborar com equipes usando pull requests e revisões de código
- Automatizar fluxos de trabalho com GitHub Actions
- Gerenciar tarefas do projeto com Issues e Projects

Autenticações do GitHub

O GitHub não suporta mais a autenticação por senha para operações Git via HTTPS desde agosto de 2021. Se você tentou se conectar, provavelmente recebeu um erro.

Para resolver isso, é necessário usar um dos métodos de autenticação atualmente suportados. A maneira mais comum é usar um **personal access token (PAT)** ou uma **chave SSH**.

SSH

1. Verificar Chaves SSH Existentes

Antes de gerar uma nova chave SSH, verifique se você já possui uma no seu sistema:

```
ls -al ~/.ssh
```

Se você vir arquivos como `id_rsa` e `id_rsa.pub`, já possui um par de chaves SSH. Caso contrário, prossiga para gerar uma nova.

2. Gerar uma Nova Chave SSH

Para gerar um novo par de chaves SSH, execute o seguinte comando:

```
ssh-keygen -t ed25519 -C "your_email@gmail.com"
```

- Substitua `your_email@gmail.com` pelo e-mail que você usa no GitHub.

Se seu sistema não suportar o algoritmo `ed25519`, use `rsa`:

```
ssh-keygen -t rsa -b 4096 -C "your_email@gmail.com"
```

- Substitua `your_email@gmail.com` pelo e-mail que você usa no GitHub.

Esse comando cria uma nova chave SSH usando o e-mail fornecido como etiqueta.

3. Salvar a Chave SSH

Quando for solicitado a "Inserir um arquivo para salvar a chave", pressione **Enter** para aceitar o local padrão (`~/.ssh/id_ed25519` ou `~/.ssh/id_rsa`).

Em seguida, será solicitado que você insira uma senha. Você pode:

- Digitar uma senha segura (recomendado para maior segurança)
- Pressionar **Enter** para pular a criação da senha (menos seguro, mas mais conveniente)

4. Adicionar Sua Chave SSH ao Agente SSH

Para gerenciar sua chave SSH, é necessário garantir que o agente SSH esteja em execução:

```
eval "$(ssh-agent -s)"
```

Agora, adicione sua chave SSH privada ao agente:

```
ssh-add ~/.ssh/id_ed25519
```

- (Ou `~/.ssh/id_rsa` se você usou RSA.)

5. Adicionar a Chave SSH ao GitHub

Agora você precisa adicionar a chave pública ao GitHub:

- Exiba a chave pública:

```
cat ~/.ssh/id_ed25519.pub
```

- Copie a saída (essa é sua chave pública).
- Acesse [SSH and GPG Keys](#) no GitHub.
- Clique em **New SSH key**, forneça um título (exemplo: "Meu Computador SSH") e cole sua chave pública no campo "Key".
- Clique em **Add SSH key**.

6. Testar Sua Conexão SSH

Teste se a chave SSH está funcionando executando:

```
ssh -T git@github.com
```

Se for bem-sucedido, você verá uma mensagem como:

```
Hi lorenzouriel! You've successfully authenticated, but GitHub does not  
provide shell access.
```

7. Alterar a URL Remota do Git para Usar SSH

Agora atualize seu repositório Git para usar SSH:

```
git remote set-url origin git@github.com:lorenzouriel/ebook-git-  
github.git
```

8. Enviar Suas Alterações

Agora tente enviar suas alterações:

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

Isso deve funcionar sem pedir seu nome de usuário ou senha.

Agora você pode começar a trabalhar com Git & GitHub!

Um Guia para Conceitos Fundamentais

Sem entender os fundamentos, sua história e o que o compõe, não faz sentido para um homem continuar trilhando o caminho.

Ok, eu não precisava dizer essa frase de efeito. Mas vamos focar neste capítulo como uma introdução geral ao que constitui Git & GitHub.

Introdução a Repositórios e Conceitos

Um **repositório** é onde os arquivos, histórico e alterações de um projeto são armazenados. Ele pode ser:

- **Remoto:** Hospedado em plataformas como GitHub, GitLab ou Bitbucket.
- **Local:** Armazenado diretamente na máquina do desenvolvedor.

Conceitos Principais

Branches

Branches permitem que desenvolvedores trabalhem em diferentes funcionalidades ou correções sem afetar o código principal. A branch principal geralmente é chamada de **main**, mas você pode criar novas branches para tarefas específicas.

- **Exemplo:** Suponha que você esteja trabalhando na **task 13**. Você cria uma branch chamada **dev-task-13**, faz alterações, realiza commits e depois faz um merge de volta para **main** quando o trabalho estiver concluído. Isso mantém sua branch principal estável.

Commit

Um **commit** captura o estado atual do projeto, como uma fotografia de todos os arquivos. Cada commit tem:

- Um identificador único (**hash**).
- Uma mensagem descritiva explicando a alteração.

Merge

Um **merge** combina alterações de uma branch em outra, geralmente integrando uma branch de funcionalidade de volta para **main**.

Issues

No GitHub, **issues** ajudam a rastrear tarefas, bugs ou solicitações de funcionalidades, proporcionando uma forma estruturada de gerenciar o desenvolvimento.

Estrutura do Repositório

HEAD

HEAD é um ponteiro para o commit mais recente na branch atual. Quando você faz um novo commit, **HEAD** é atualizado para apontar para ele.

Working Tree

O **Working Tree** contém os arquivos do seu projeto, onde você pode modificar e criar conteúdo.

Index (Staging Area)

O **index** é uma área de armazenamento intermediária onde o Git rastreia alterações antes do commit.

Para adicionar arquivos ao index:

```
git add file.txt
```

Comandos Principais

git commit

Realiza o commit das alterações preparadas, criando uma nova versão no repositório.

Fluxo do Commit:

1. Verifique o status dos arquivos modificados:

```
git status
```

2. Adicione arquivos ao commit:

```
git add file1.txt file2.js  
  
# ou adicione todos os arquivos  
  
git add .
```

3. Crie um commit com uma mensagem:

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

- A flag **-m** permite especificar uma mensagem.

git push

Envia as alterações locais para o repositório remoto.

Fluxo do Push:

1. Envie as alterações para o repositório remoto:

```
git push
```

2. Se for o primeiro push para uma nova branch remota:

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

- A flag **-u** configura o rastreamento para futuros pushes e pulls.

git pull

Busca e mescla alterações do repositório remoto para a branch local.

Fluxo do Pull:

1. Obtenha as atualizações mais recentes:

```
git pull origin main
```

2. Se o rastreamento já estiver configurado:

```
git pull
```

git tag

Uma **tag** marca um commit específico no histórico do Git, comumente usada para versionamento.

Fluxo de Tags:

1. Liste as tags existentes:

```
git tag
```

2. Crie uma nova tag anotada:

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

3. Envie a tag para o repositório remoto:

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

git init

Inicializa um novo repositório Git no diretório atual.

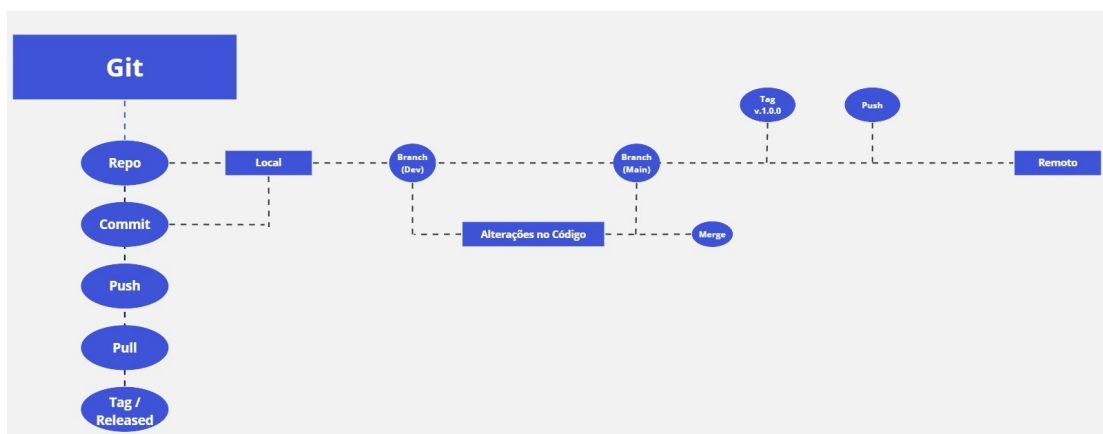
```
git init
```

git clone

Cria uma cópia local de um repositório remoto.

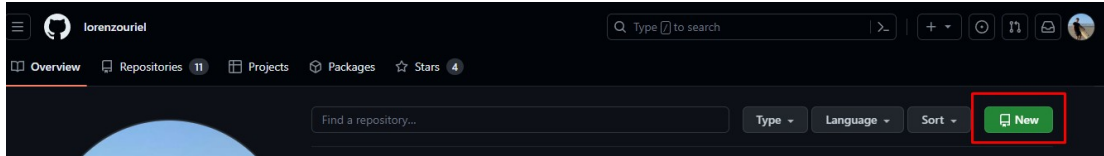
```
git clone https://github.com/lorenzouriel/ebook-git-github.git
```

Arquitetura, Conceitos e Comandos:



Criando um Repositório Local e Remoto

1. Vá para seu GitHub e crie um novo repositório:



2. Navegue até o diretório local onde você quer criar o repositório

```
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 anexa 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 que você adicione a mensagem de commit diretamente na linha de comando.

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

8. Renomeia o branch padrão do repositório para **main**. Este comando é usado para atualizar o nome do branch principal para seguir as práticas de nomenclatura mais recentes, substituindo o antigo master.

```
git branch -M main
```


9. Adiciona um repositório remoto chamado "origin". O termo "origin" é um termo padrão usado para se referir ao repositório remoto principal. A 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") no branch principal. O -u estabelece um relacionamento de rastreamento, associando automaticamente o branch local ao branch remoto. Isso é útil para futuros git pull e git push sem precisar especificar o branch.

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

Podemos verificar nosso repositório remoto:


**up-website-with-vagrant** Public






Pin Unwatch 1



main 1 Branch 0 Tags

Add file

<> Code

**Lorenzo Uriel** first commit 6413063 · now 1 Commits

 .vagrant	first commit	now
 html	first commit	now
 README.md	first commit	now
 Vagrantfile	first commit	now
 provision.sh	first commit	now

 **README** 

"# up-website-with-vagrant"

O que é Versionamento e Tags?

Você já trabalhou com versões de lançamento?

Ou com Tags no Git?

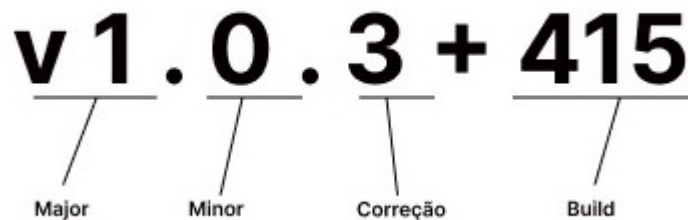
Tags são muito importantes nos nossos projetos - com elas podemos identificar em que momento ocorreram os principais lançamentos e versões.

Mas quando você adiciona tags, saberá a diferença entre cada número?

Vou explicar de forma simples e prática:

- **Major:** Mudanças incompatíveis com versões anteriores (Reestruturou tudo? Adicione +1 - v2.0.0)
- **Minor:** Mudanças importantes compatíveis com a versão anterior (Adicionou uma nova funcionalidade? Adicione +1 - v2.1.0)
- **Correction:** Correções de erros e bugs que não afetam a versão (Encontrou um bug e corrigiu? Adicione +1 - v2.1.1)
- **Build:** Controle interno do Git e versionamento, não necessariamente visível ao especificar a versão.

Exemplo:



Introdução ao Markdown

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

Criada por John Gruber e Aaron Swartz em 2004, seu objetivo é ser lida em texto puro enquanto permite conversão automática para HTML. Markdown é amplamente utilizada para criar documentos, escrever posts de blog e produzir **README.md** em repositórios de código.

Cabeçalhos:

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

```
# Nível de cabeçalho 1
## Nível de cabeçalho 2
### Nível de cabeçalho 3
```

Listas:

- O 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 o URL.

```
[Link para meu site](https://www.example.com)
```

Imagens:

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

```
![Logotipo do projeto](images/logo.jpg)
```

Ênfase (negrito e itálico):

- Para criar ênfase no 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 melhor versão de si mesmo." - Einstein
```

Tabelas:

- As tabelas podem ser criadas usando **|** para separar colunas e **-** para criar a linha de cabeçalho.

```
| Nome | Função |  
|-----|-----|  
| John | Desenvolvedor |  
| Mary | Designer |
```

Código:

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

Markdown	HTML	Rendered Output
<pre>``Use `code` in your Markdown file.``</pre>	<pre><code>Use `code` in your Markdown file.</code></pre>	Use <code>`code`</code> in your Markdown file.

Até agora entendemos o que compõe o Git e como começar seu primeiro projeto, vamos nos

aprofundar no que vimos até agora!

Um Guia para Trabalhar com Branchs

Então, você já sabe o que é um branch e por que você deveria ter um. Mas agora, como você trabalha e utiliza elas?

O que é uma Git Branch?

Uma Git branch é uma linha independente de desenvolvimento dentro de um repositório. Pense nela como um espaço de trabalho separado onde você pode fazer alterações sem impactar a branch principal.

As branches permitem que os desenvolvedores trabalhem em novas funcionalidades, correções de bugs ou experimentos sem interromper a versão atual do projeto.

Por que usar Branches?

As branches oferecem diversos benefícios em um fluxo de trabalho de desenvolvimento:

- **Isolamento:** Mantenha tarefas diferentes (funcionalidades, correções de bugs, experimentos) separadas.
- **Colaboração:** Vários desenvolvedores podem trabalhar em branches diferentes sem interferir uns nos outros.
- **Experimentação Segura:** Teste alterações sem impactar o código em produção.
- **Controle de Versão:** Reverta ou alterne facilmente entre diferentes versões do projeto.

No Git, a branch padrão geralmente é chamada de **main** ou **master**. No entanto, novas branches podem ser criadas para diversos propósitos, como:

- **Feature branches (**feature/new-ui**):** Usadas para desenvolver novas funcionalidades.
- **Bugfix branches (**hotfix/login-fix**):** Usadas para corrigir problemas em produção.
- **Release branches (**release/v1.2.0**):** Usadas para preparar versões estáveis para implantação.

Criando e Gerenciando Branches Locais

As branches no Git permitem que você trabalhe em novas funcionalidades, correções de bugs ou experimentos sem afetar a base de código principal. Aqui está como criar, alternar, renomear e deletar branches de forma eficiente.

1. Criar uma Nova Branch

```
git branch feature/create-chapter
```

- Isso cria uma nova branch chamada `feature/create-chapter`, mas não muda para ela.
- Útil quando você precisa preparar várias branches, mas não deseja alternar imediatamente.

2. Alternar para uma Branch

```
git checkout feature/create-chapter
```

- Isso move você para a branch especificada para começar a codificar.

Alternativa Moderna: Desde o Git 2.23+, use `git switch`:

```
git switch feature/create-chapter
```

É uma alternativa mais limpa e segura ao `checkout`.

3. Criar e Alternar para uma Nova Branch (Atalho)

```
git checkout -b feature/create-chapter
```

- Cria uma nova branch e alterna para ela imediatamente.
- Economiza tempo ao combinar dois comandos em um só.

Alternativa Moderna:

```
git switch -c feature/create-chapter
```

Mais intuitivo e recomendado para versões mais recentes do Git.

4. Listar Todas as Branches

```
git branch
```

- Exibe todas as branches locais. A branch atual é marcada com *****.

```
* dev/ebook-v2  
main  
feature/add-search  
bugfix/fix-typo
```

- Isso ajuda a rastrear as branches disponíveis e navegar entre elas.

Para listar também as branches remotas:

```
git branch -a
```

5. Renomear uma Branch

```
git branch -m feature/create-chapter feature/create-chapter-branch
```

- Renomeia a branch `feature/create-chapter` para `feature/create-chapter-branch`.

Se você já estiver na branch que deseja renomear:

```
git branch -m new-branch-name
```

6. Deletar uma Branch Local (Com Segurança)

```
git branch -d feature/create-chapter-branch
```

- Deleta a branch apenas se ela já foi mesclada em outra branch.
- Se houver alterações não mescladas, o Git impedirá a exclusão para evitar perda de dados.

7. Forçar a Exclusão de uma Branch (Perigoso)

```
git branch -D feature/create-chapter-branch
```

- Isso exclui a branch incondicionalmente, mesmo se houver alterações não mescladas.
- Use com cuidado para evitar perder trabalho importante.

Melhores Práticas para Gerenciar Branches

1. Use nomes de branches claros e descritivos (`feature/add-login`, `bugfix/fix-typo`).
2. Delete branches antigas para manter seu repositório organizado.
3. Use `-d` em vez de `-D`, a menos que tenha certeza de que deseja forçar a exclusão.
4. Envie regularmente branches para o remoto (`git push origin branch-name`) para evitar perda acidental.

Fazendo Merge de Branches

No Git, **merge** é o processo de integrar alterações de uma branch em outra. É usado para combinar atualizações de diferentes branches de desenvolvimento na branch principal, como **main** ou **develop**.

1. Fazer Merge de uma Branch na Branch Atual

Para integrar alterações de outra branch na sua branch atual:

```
git merge feature/create-chapter-branch
```

- Isso integra as alterações da branch **feature/create-chapter-branch** na branch em que você está atualmente.
- Se não houver conflitos, o Git concluirá o merge automaticamente.

Certifique-se de estar na branch correta antes de fazer o merge:

```
git checkout main # ou git switch main  
git merge feature/create-chapter-branch
```

2. Resolvendo Conflitos de Merge

Durante um merge, se houver conflitos, o Git pausará o processo e informará os arquivos conflitantes. Edite esses arquivos para resolver os conflitos, marcados com:

```
<<<<<<< HEAD  
(changes in the current branch)  
=====  
(changes in the branch being merged)  
>>>>>>> main
```

- A seção HEAD representa as alterações da sua branch atual.
- A seção abaixo de **=====** vem da branch sendo mesclada.

Passos para Resolver Conflitos:

1. Abra os arquivos conflitantes em um editor de texto.
2. Edite manualmente e mantenha a versão correta do código.
3. Marque os arquivos como resolvidos:

```
git add .
```

4. Conclua o merge com um commit:

```
git commit -m "Conflicting files resolved."
```

Para abortar um merge e retornar ao estado anterior:

```
git merge --abort
```

Tipos de Git Merges

O Git suporta diferentes estratégias de merge dependendo se as branches divergiram ou não.

1. Fast-Forward Merge (Sem Divergência)

Um fast-forward merge acontece quando a branch que está sendo mesclada está à frente da branch atual sem nenhuma alteração na branch atual.

O Git move o ponteiro da branch para frente em vez de criar um commit de merge.

Exemplo:

```
git checkout main  
git merge feature/create-chapter-branch
```

- Isso funciona quando **main** não foi alterado desde que **feature/create-chapter-branch** foi criado.

2. Three-Way Merge (Branches Divergentes)

Um three-way merge ocorre quando as duas branches têm históricos diferentes e não podem ser fast-forwarded.

O Git cria um novo commit de merge para combinar as alterações.

```
git checkout main  
git merge feature/create-chapter-branch
```

Você verá uma mensagem de commit como:

```
Merge branch 'feature/create-chapter-branch' into main
```

Melhores Práticas para Fazer Merge

1. Sempre faça pull das últimas alterações antes de fazer merge.
2. Teste seu código após fazer merge para garantir que tudo funcione.
3. Use feature branches para desenvolvimento para manter **main** estável.
4. Delete branches mescladas para manter o repositório limpo.

Fazendo Rebase de Branches

Rebasing é um recurso do Git que **permite integrar alterações de uma branch em outra, movendo sua branch para o estado mais recente da branch de destino**. Ao contrário do merge, que cria um novo commit de merge, **o rebasing repete seus commits em cima das últimas alterações, mantendo o histórico de commits mais limpo**.

Quando usar rebase?

- Para manter uma feature branch atualizada com a branch principal.
- Para limpar o histórico de commits antes de fazer merge.
- Para reescrever o histórico de commits para melhor legibilidade.

1. Fazer Rebase de uma Branch em Outra Branch

```
git checkout feature/create-chapter-branch
git rebase main
```

Isso atualiza a **feature/create-chapter-branch** com os commits mais recentes de **main**, garantindo que ela seja baseada nas alterações mais recentes.

2. Cancelar um Rebase em Andamento

Se algo der errado durante o rebasing e você quiser cancelar o processo, use:

```
git rebase --abort
```

Isso restaura sua branch para o estado anterior ao início do rebase.

3. Continuar um Rebase Interrompido Após Resolver Conflitos

Quando ocorre um conflito, o Git pausa o rebase e pede para você resolver os conflitos manualmente nos arquivos afetados.

Após corrigir os conflitos, adicione os arquivos resolvidos ao stage:

```
git add .
```

Então, continue o rebase:

```
git rebase --continue
```

O Git continuará aplicando os commits restantes. Se outro conflito ocorrer, repita o processo.

4. Rebase Interativo (modificar histórico de commits)

Para editar, reordenar, juntar (squash) ou remover commits, use o rebase interativo:

```
git rebase -i HEAD~3
```

HEAD~3 significa que você está interagindo com os 3 últimos commits.

Após executar este comando, o Git abre um editor interativo com opções como:

- **pick** → Manter o commit como está.
- **reword** → Alterar a mensagem do commit.
- **edit** → Modificar o conteúdo do commit.
- **squash** → Juntar os commits.
- **drop** → Deletar um commit.

Exemplo de uma janela de editor de rebase interativo:


```

pick 123abc filename3
squash 456def commit
pick 789ghi first commit

# Rebase db14708..ca382f6 onto db14708 (1 command)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup [-C | -c] = 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 = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --
continue')
# d, drop = remove commit
# l, label = label current HEAD with a name
# t, reset = reset HEAD to a label
# m, merge [-C | -c ] [# ]
#       create a merge commit using the original merge commit's
#       message (or the oneline, if no original merge commit was
#       specified); use -c to reword the commit message
# u, update-ref = track a placeholder for the to be updated
#                to this position in the new commits. The 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.
#
~
.git/rebase-merge/git-rebase-todo [unix] (20:33 16/02/2025)

```

Isso irá juntar o segundo commit no primeiro, combinando suas alterações.

Melhores Práticas para Rebasing:

- Sempre faça rebase de branches locais antes de fazer merge para manter o histórico limpo.
- Evite fazer rebase de branches compartilhadas (**main**, **dev**), pois isso reescreve o histórico de commits.
- Use **git rebase --interactive** para reescrever o histórico de forma organizada.
- Se não tiver certeza, crie uma branch de backup antes de fazer rebase.

Branches Remotas e Rastreamento de Branches

Branches remotas são versões das suas branches armazenadas em um repositório remoto. Essas branches permitem a colaboração entre vários desenvolvedores, mantendo seus repositórios locais sincronizados com o remoto.

Por que usar branches remotas?

- Para colaborar com outras pessoas enviando e recebendo alterações.
- Para manter um backup do seu trabalho em um repositório central.
- Para gerenciar diferentes ambientes (**main**, **dev**, **staging**).

1. Listar Branches Remotas

Para ver todas as branches armazenadas no repositório remoto:

```
git branch -r
```

- Isso lista apenas as branches remotas, prefixadas por **origin/**.

Para listar as branches locais e remotas:

```
git branch -a
```

Isso ajuda a verificar quais branches existem localmente e remotamente.

2. Criar uma Branch Rastreando uma Branch Remota

Se você deseja trabalhar em uma branch remota localmente, use:

```
git checkout --track origin/dev
```

- Isso cria uma branch local que rastreia automaticamente a remota.

O retorno será:

```
branch 'dev' set up to track 'origin/dev'.  
Switched to a new branch 'dev'
```

Isso é útil quando o Git não rastreia automaticamente a branch.

3. Atualizar Branches Remotas

Buscar atualizações do repositório remoto garante que você tenha a lista de branches mais recente:

```
git fetch
```

- Isso baixa as alterações remotas, mas não as aplica ao seu diretório de trabalho.

4. Mesclar Alterações de uma Branch Remota na Branch Atual

Para obter as atualizações mais recentes de uma branch remota na sua branch local:

```
git pull origin main
```

Isso é equivalente a executar:

```
git fetch  
git merge origin/main
```

Se a branch já estiver rastreando **origin/main**, você pode simplesmente executar:

```
git pull
```

5. Enviar uma Nova Branch Local para o Repositório Remoto

Depois de criar uma nova branch localmente, envie-a para o repositório remoto:

```
git push -u origin dev
```

- O sinalizador `-u` configura o rastreamento, para que os futuros comandos `git pull` e `git push` possam ser executados sem especificar a branch remota.

6. Excluir uma Branch Remota

Se uma branch remota não for mais necessária, exclua-a usando:

```
git push origin --delete dev
```

- Isso remove `origin dev` do repositório remoto.

Para excluir uma referência local à branch remota excluída:

```
git remote prune origin
```

Melhores Práticas para Gerenciamento de Branch Remota

1. Use nomes de branch descritivos (`feature/login`, `bugfix/navbar-issue`).
2. Sempre faça fetch antes de mesclar para evitar conflitos.
3. Limpe branches excluídas regularmente
4. Não envie branches de trabalho em andamento, a menos que seja necessário.

Guardando Alterações (Stashing)

O Git stash **permite que você salve temporariamente as alterações não commitadas sem commitá-las**. Isso é útil quando você precisa:

- Trocar de branch sem commitar trabalho incompleto.
- Manter seu diretório de trabalho limpo enquanto recebe alterações.
- Salvar trabalho temporário que você não está pronto para commitar.

Ao usar o stashing, o Git armazena suas alterações com segurança para que você possa recuperá-las posteriormente.

1. Guardar Alterações Não Commitadas

Para guardar todos os arquivos modificados e adicionados ao stage:

```
git stash
```

- Isso remove as alterações do diretório de trabalho e as salva em uma pilha.

Para guardar com uma mensagem personalizada:

```
git stash push -m "Fixing ETL"
```

- Ajuda você a identificar o que cada stash contém.

2. Listar Alterações Guardadas

Para visualizar todos os stashes salvos:

```
git stash list
```

- Cada entrada de stash é rotulada como **stash@{n}**, onde **n** é seu índice.

Exemplo de saída:

```
stash@{0}: On main: Fixing ETL  
stash@{1}: On dev: Debugging API issue
```

- O stash mais recente é sempre `stash@{0}`.

3. Aplicar as Alterações Guardadas Mais Recentes

Para aplicar o stash mais recente sem removê-lo da lista de stashes:

```
git stash apply
```

- Isso restaura as alterações guardadas, mas as mantém no stash.

Para aplicar um stash específico:

```
git stash apply --index 2
```

- Substitua `2` pelo índice do stash desejado.

4. Aplicar e Remover as Alterações Guardadas Mais Recentes

Para restaurar e remover o stash mais recente:

```
git stash pop
```

Para remover um stash específico:

```
git stash pop --index 1
```

- Isso remove `stash@{1}` após aplicar suas alterações.

5. Descartar um Stash Específico

Para excluir um stash específico sem aplicá-lo:

```
git stash drop --q 1
```

- Remove `stash@{1}` da lista de stashes.

6. Limpar Todos os Stashes

Para excluir todos os stashes de uma vez:

```
git stash clear
```

- Use com cautela - isso exclui permanentemente todas as alterações guardadas.

7. Criar uma Nova Branch a partir de um Stash

Para criar uma branch com as alterações guardadas:

```
git stash branch feature-fix
```

- Isso cria uma nova branch `feature-fix` a partir do stash mais recente e aplica o stash.

Melhores Práticas para Stashing

1. Use mensagens de stash significativas.
2. Aplique o stash antes de trocar de branch, se necessário.
3. Limpe stashes antigos para manter seu repositório limpo.
4. Use stash branches para alterações complexas.

Workflow com Branches (Feature Branch, Hotfix e Release)

Este guia descreve o workflow típico usando branches Git para features, hotfixes e releases. Ele garante que os processos de desenvolvimento, correção de bugs e lançamento sejam otimizados.

1. Feature Branch

As feature branches permitem que você trabalhe em novas funcionalidades de forma independente, sem afetar a base de código principal.

- Crie uma nova branch para a feature:

```
git checkout -b feature/create-chapter-branch
```

- Trabalhe na feature e faça commits, commitando regularmente suas alterações à medida que avança.
- Faça merge da feature branch na branch principal (**main** ou **dev**).

```
git checkout main  
git merge feature/create-chapter-branch
```

2. Hotfix Branch

Hotfixes são usados para correções urgentes de bugs no ambiente de produção, geralmente com base na branch principal.

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

```
git checkout -b hotfix/main-app-filter main
```

- Trabalhe no hotfix e faça commit.
- Faça merge da hotfix branch na branch principal:

```
git checkout main
git merge hotfix/main-app-filter main
```

- Também faça merge do hotfix na branch **dev**: Se você tiver uma branch **dev**, certifique-se de que o **hotfix** também seja mesclado lá, para manter a branch de desenvolvimento atualizada.

```
git checkout dev
git merge hotfix/main-app-filter main
```

3. Release Branch

As release branches são criadas para preparar uma nova versão do software para produção. Elas permitem testes, ajustes finais e versionamento.

- Crie uma nova release branch a partir da dev branch: A release branch deve ser baseada na dev branch para incluir todas as novas funcionalidades.

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

- Teste e prepare o release, commitando conforme necessário. Faça os ajustes finais, realize os testes e commite quaisquer alterações necessárias.
- Faça merge da release branch na branch principal e tag o release: Após os testes, faça merge da release branch na main e tag com o número da versão para marcar o lançamento oficial.

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

- Faça merge da release branch de volta na dev branch: Para garantir que quaisquer alterações finais feitas durante o processo de lançamento sejam refletidas na branch de desenvolvimento, faça merge da release branch de volta na dev.

```
git checkout dev  
git merge release/1.0.0
```

Melhores Práticas para Workflows

1. Use Nomes de Branch Descritivos
2. Mantenha as Branches de Curta Duração
3. Commite Frequentemente e Logicamente
4. Mantenha a Branch principal Estável
5. Use dev para Desenvolvimento Contínuo
6. Faça Merge em vez de Envio Direto para main ou dev
7. Use Tags para Releases
8. Garanta Testes Adequados nas Release Branches
9. Limpe as Branches Após o Merge

Selecionando Commits (Cherry-Picking)

Cherry-picking **permite que você aplique seletivamente commits específicos de uma branch para outra**. Em vez de fazer merge ou rebase de uma branch inteira, você pode escolher apenas os commits de que precisa.

Isso é útil quando:

- Uma correção de bug está em uma feature branch e precisa ser aplicada à main.
- Um commit da branch de outro desenvolvedor deve ser adicionado à sua.
- Um commit foi adicionado incorretamente à branch errada e precisa ser movido.

1. Aplicar um Commit Específico de Outra Branch

Para aplicar um commit de outra branch à sua branch atual:

```
git cherry-pick a1b2c3d
```

- Isso aplica o commit com o hash **a1b2c3d** à sua branch atual.

2. Aplicar Vários Commits

Para cherry-pick vários commits em um comando:

```
git cherry-pick a1b2c3d e4f5g6h
```

- Isso aplica ambos os commits em sequência.

3. Abortar uma Operação de Cherry-Pick

Se você encontrar um conflito e quiser cancelar a operação:

```
git cherry-pick --abort
```

- Isso restaura sua branch ao estado anterior ao início do cherry-picking.

4. Cherry-Picking Sem Commitar

Se você quiser aplicar as alterações de um commit sem commitar:

```
git cherry-pick -n alb2c3d
```

- Isso aplica as alterações, mas não cria um commit.
- Você pode modificar os arquivos antes de commitar manualmente.

5. Cherry-Picking com uma Mensagem de Commit Personalizada

Para usar uma mensagem personalizada em vez da mensagem de commit original:

```
git cherry-pick -e alb2c3d
```

- Isso abre um editor onde você pode modificar a mensagem de commit.

Melhores Práticas para Cherry-Picking

1. Certifique-se de estar na branch correta antes de fazer cherry-picking.
2. O uso de cherry-picking pode levar a commits duplicados.
3. Verifique o histórico de commits após o cherry-picking para evitar conflitos.
4. Considere fazer merge ou rebase em vez de cherry-picking quando apropriado.

Desfazendo Alterações

Às vezes, erros acontecem, e você precisa desfazer alterações no Git. Dependendo do cenário, você pode querer:

- Manter as alterações adicionadas ao stage enquanto desfaz um commit.
- Apagar completamente um commit e suas alterações.
- Reverter um commit mantendo o histórico intacto.
- Restaurar arquivos para um estado anterior.

1. Desfazer o Último Commit (Manter Alterações no Stage)

Se você quiser desfazer o último commit, mas manter as alterações adicionadas ao stage:

```
git reset --soft HEAD~1
```

- O commit é desfeito, mas as alterações permanecem na área de stage.
- Útil quando você commita acidentalmente muito cedo e deseja modificar o commit antes de enviar.

Exemplo:

```
git reset --soft HEAD~1  
git commit -m "Updated commit message"
```

2. Desfazer o Último Commit (Descartar Alterações)

Se você quiser desfazer o último commit e descartar todas as alterações:

```
git reset --hard HEAD~1
```

- Isso remove completamente o commit e suas alterações.

Exemplo:

```
git reset --hard HEAD~1  
git log --oneline
```

3. Restaurar um Arquivo Excluído

Se você excluiu acidentalmente um arquivo e ainda não commitou a exclusão:

```
git checkout -- nome_do_arquivo.txt
```

- Isso restaura o arquivo para seu último estado commitado.

4. Redefinir um Arquivo para o Último Estado Commitado

Se um arquivo foi modificado, mas não adicionado ao stage, e você deseja descartar as alterações:

```
git checkout HEAD -- nome_do_arquivo.txt
```

- Isso restaura o arquivo para sua última versão commitada, descartando as modificações locais.

5. Reverter um Commit

Se você quiser desfazer um commit, mas manter o histórico intacto:

```
git revert 0a91ea2
```

- Isso cria um novo commit que reverte as alterações do commit especificado.
- Ao contrário do **reset**, ele não remove o histórico, tornando-o mais seguro para repositórios compartilhados.

6. Desfazer um Commit Enviado (Pushed)

Se você enviou (pushed) um commit acidentalmente e precisa desfazê-lo antes que outros o recebam:

```
git reset --hard HEAD~1
git push --force
```

- *git push --force* sobrescreve o histórico e pode causar problemas se outros já tiverem recebido o commit.

Se o commit já tiver sido compartilhado, use revert em vez disso:

```
git revert 0a91ea2
git push
```

7. Desfazer Todas as Alterações Locais

Se você quiser redefinir tudo para o último estado commitado:

```
git reset --hard HEAD
```

- Isso remove todas as alterações não commitadas. Use com cautela.

Melhores Práticas para Desfazer Alterações

- Use **reset --soft** quando quiser refazer um commit, mas manter suas alterações no stage.
- Use **reset --hard** com cautela, pois ele exclui as alterações permanentemente.
- Use **revert** em vez de **reset** ao trabalhar em um repositório compartilhado.
- Sempre verifique o histórico de commits (**git log --oneline**) antes de fazer alterações destrutivas.

Um Guia para Sincronização e Colaboração

Aqui está o ponto principal para usar o GitHub: sincronização e colaboração entre equipes ou projetos pessoais com um amigo. Foque em utilizá-lo dessa forma e você entenderá como ele pode ser funcional.

Configurando Repositórios Remotos

Repositórios remotos são versões do seu projeto hospedadas na internet, permitindo que vários desenvolvedores colaborem. Eles servem como o hub central para armazenar e compartilhar alterações no projeto.

Para trabalhar com repositórios remotos, você precisa adicioná-los, modificá-los ou removê-los conforme necessário. Esse é um 80/20 dos repositórios remotos:

1. Adicionar um Repositório Remoto

```
git remote add origin  
https://github.com/lorenzouriel/ebook-git-github.git
```

- Substitua pela URL do seu repositório remoto.

2. Verificar Repositórios Remotos Configurados

```
git remote -v
```

- Este comando lista todos os repositórios remotos configurados, juntamente com suas URLs de **fetch** e **push**.

3. Alterar a URL de um Repositório Remoto

```
git remote set-url origin  
https://github.com/lorenzouriel/ebook-git-github.git
```

- Útil ao mudar a localização remota, como migrar para outro provedor ou repositório.

4. Remover um Repositório Remoto

```
git remote remove origin
```

- Remove a conexão com o repositório remoto especificado.

Enviando Alterações para o Repositório Remoto (**git push**)

Depois de fazer e confirmar alterações localmente, você precisa enviá-las para o repositório remoto.

1. Enviar Alterações para a Branch Principal

```
git push origin main
```

- Envia as atualizações da branch **main** local para o repositório remoto.

2. Enviar uma Branch Específica para o Repositório Remoto

```
git push origin dev
```

- Substitua **dev** pelo nome da branch que deseja enviar.

3. Enviar Todas as Tags Locais para o Repositório Remoto

```
git push --tags
```

- Envia todas as tags criadas localmente para o repositório remoto.

Obtendo Alterações do Repositório Remoto (`git pull` e `git fetch`)

Para se manter atualizado com as alterações remotas, você pode usar `pull` ou `fetch`.

1. Atualizar Seu Repositório Local com as Últimas Alterações e Fazer Merge Automaticamente

```
git pull main
```

- Equivalente a `git fetch` seguido de `git merge`, baixa e integra alterações remotas.

2. Buscar Alterações do Repositório Remoto Sem Fazer Merge

```
git fetch
```

- Baixa atualizações, mas não as aplica automaticamente.

3. Fazer Merge Manual das Alterações Buscadas

```
git merge
```

- Faz o merge das atualizações buscadas na sua branch local.

Removendo Repositórios Locais e Remotos

1. Remover um Repositório Localmente

Para deletar um repositório Git do seu computador, você pode remover o diretório onde ele está armazenado. Tenha cuidado, pois isso removerá todos os arquivos e histórico de commits dentro do repositório.

```
# Usando Bash (Linux/macOS)
rm -rf /path/to/your/repository

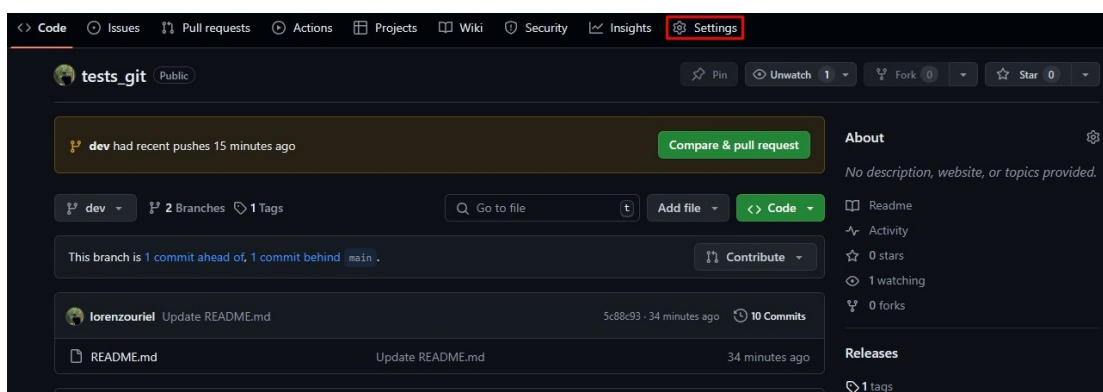
# Usando PowerShell (Windows)
Remove-Item -Path "C:\Projects\portfolio\tests_git" -Recurse -Force

git commit -m "Remove tests_git directory"
git push
```

2. Deletar Repositórios Remotos no GitHub

Para deletar um repositório remoto no GitHub:

1. Navegue até a página do repositório no GitHub.
2. Vá até a seção **Configurações** do repositório.



3. Role até encontrar a opção **Deletar repositório**.
4. Confirme a exclusão inserindo sua senha ou usando o GitHub Mobile.

Exemplo Completo de Fluxo

Este exemplo demonstra um fluxo completo de contribuição via fork e pull request:

1. Clonar o Repositório

```
git clone https://github.com/lorenzouriel/ebook-git-github.git
git remote add upstream
https://github.com/lorenzouriel/ebook-git-github.git

cd ebook-git-github/
```

2. Criar uma Nova Branch, Fazer Alterações e Confirmar (Commit)

```
git checkout -b my-chapter

# Modifique os arquivos

git add .
git commit -m "Added a new chapter"
```

3. Enviar Alterações para Seu Fork

```
git push
```

4. Criar um Pull Request no GitHub

- Navegue até o repositório original.
- Clique em **New Pull Request**.
- Selecione sua branch (**my-chapter**) e a branch principal do repositório original.
- Adicione uma descrição e envie o pull request.

5. Sincronizar com o Repositório Original Após um Merge

Atualize seu repositório local com as últimas alterações:

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

Envie a branch principal sincronizada de volta ao seu fork:

```
git push
```

6. Deletar a Branch Mesclada (Merged)

```
git branch -d my-chapter  
  
# Opcionalmente, delete também a branch remota:  
git push origin --delete my-chapter
```

Este fluxo garante sincronização e colaboração enquanto contribui para projetos open-source.

Trabalhando com Tags e Releases

1. Criar uma Tag

As tags são usadas para marcar pontos específicos no histórico de commits, como lançamentos.

```
git tag v1
```

2. Criar uma Tag Anotada

Tags anotadas incluem metadados adicionais, como uma mensagem de tag, tornando-as ideais para lançamentos.

```
git tag -a v1 -m "Versão 01"
```

3. Enviar uma Tag para o Repositório Remoto

Após criar uma nova tag, envie-a para o repositório remoto:

```
git push origin v1
```

4. Enviar Todas as Tags Locais para o Repositório Remoto

Para enviar todas as tags de uma vez:

```
git push --tags
```

5. Listar Todas as Tags

Para listar todas as tags no seu repositório local:

```
git tag
```

6. Deletar uma Tag Localmente

Para deletar uma tag do seu repositório local:

```
git tag -d v1
```

7. Deletar uma Tag no Repositório Remoto

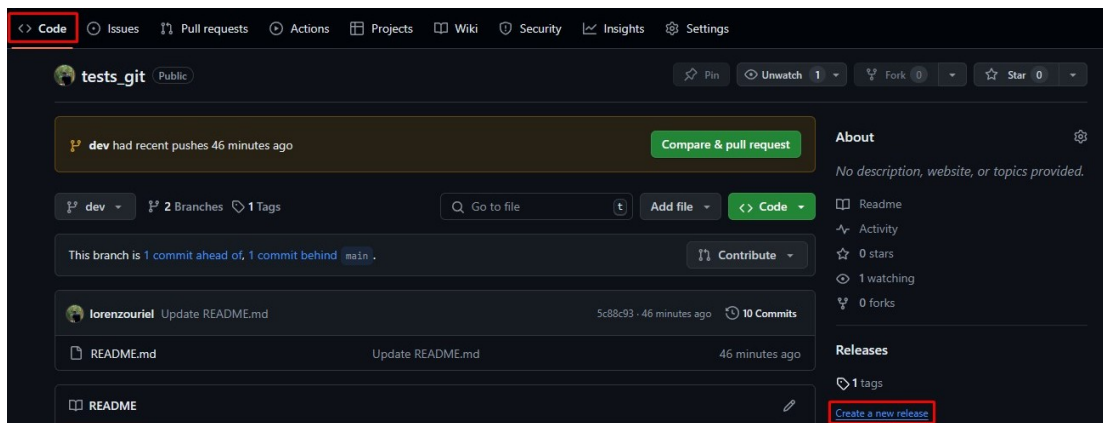
Para deletar uma tag do repositório remoto:

```
git push origin --delete v1
```

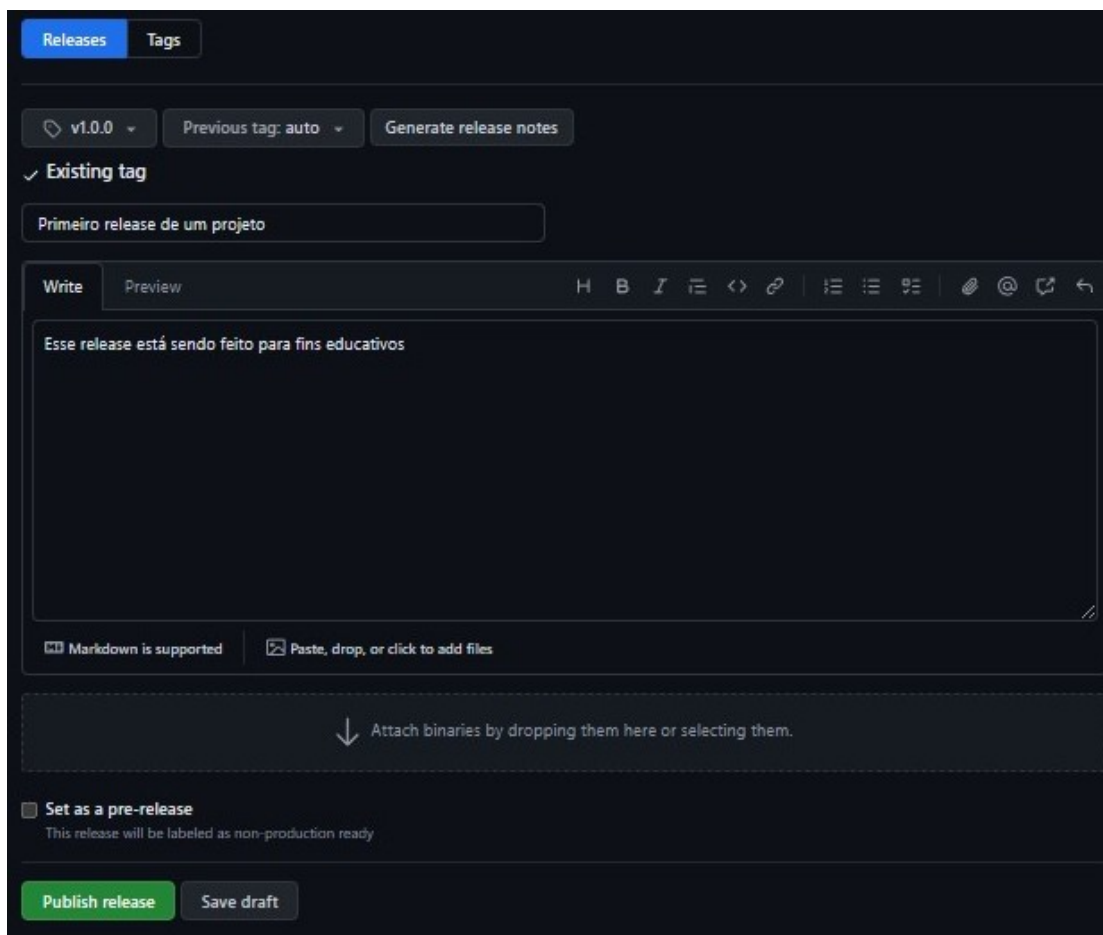
8. Criar um Release no GitHub

Para criar um release no GitHub:

1. Navegue até a seção **Releases** do repositório.
2. Clique em **Create a New Release**.



3. Preencha as informações da versão, associe uma tag e clique em **Publish a New Release**:



4. Seus releases serão organizados na seção releases na aba **Code**:

tests_gitPublic

PinUnwatch1Fork0Star0

dev had recent pushes 49 minutes agoCompare & pull request

v1.0.02 Branches1 TagsGo to fileCode

Lorenzo UrielCommit da versão 01 indo263498 · yesterday9 Commits

README.mdCommit da versão 01 indo · yesterday

README

Vamos iniciar

About

No description, website, or topics provided.

ReadmeActivity0 stars1 watching0 forks

Releases1

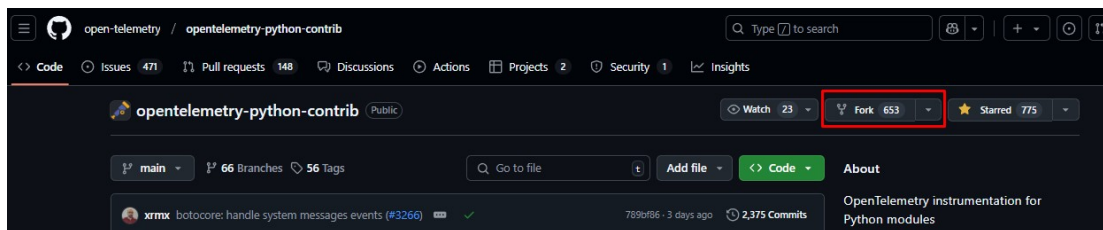
Primeiro release de um projetoLatestnow

Trabalhando com Forks e Pull Requests

Fazer um fork de um repositório permite que você contribua para um projeto sem modificar diretamente o repositório original. É uma forma comum de colaborar em projetos open-source.

1. Fazer Fork de um Repositório

No GitHub, use a interface web para fazer fork do repositório desejado.



2. Clone o Repositório

```
git clone  
https://github.com/lorenzouriel/opentelemetry-python-contrib.git
```

- Isso cria uma cópia local do seu repositório forkado.

3. Adicionar um Repositório Upstream para Sincronizar com o Projeto Original

```
git remote add upstream  
https://github.com/open-telemetry/opentelemetry-python-contrib.git
```

- O **upstream** refere-se ao repositório original de onde você fez o fork.

4. Sincronizar Seu Fork com o Repositório Original

Busque alterações do repositório upstream:

```
git fetch upstream
```

- Faça merge das atualizações na sua branch principal:

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

- Envie as atualizações para o seu fork:

```
git push origin main
```

5. Criar um Pull Request

- No GitHub, vá até o repositório original e clique em **New Pull Request**.
- Compare as alterações entre seu fork e o repositório original.
- Clique em **Create Pull Request** e adicione uma mensagem descritiva.
- Após revisão e aprovação, o mantenedor pode fazer o merge da sua contribuição.

Mais Sobre o Comando `git remote`

O comando `git remote` é usado para gerenciar repositórios remotos no Git. Ele permite visualizar, adicionar e remover remotos no seu repositório.

1. Visualizar os Remotos Atuais

Para listar todos os repositórios remotos vinculados ao seu repositório local, use o seguinte comando:

```
git remote -v
```

- Isso exibirá as URLs dos repositórios remotos para fetch e push.

2. Adicionar um Novo Remote

Para adicionar um novo repositório remoto, use o seguinte comando:

```
git remote add nome-repo url-repo
```

Por exemplo:

```
git remote add new-origin  
https://github.com/lorenzouriel/ebook-git-github.git
```

3. Remover um Remote

Para remover um remote existente, use o seguinte comando:

```
git remote remove nome-repo
```

Por exemplo, para remover o remote `origin`:

```
git remote remove new-origin
```

4. Renomear um Remote

Se você quiser renomear um repositório remoto, use:

```
git remote rename nome-antigo nome-novo
```

Por exemplo:

```
git remote rename new-origin upstream
```

5. Alterar a URL de um Remote

Para alterar a URL de um remote existente:

```
git remote set-url nome-repo nova-url-repo
```

Por exemplo:

```
git remote set-url origin  
https://github.com/lorenzouriel/ebook-git-github.git
```

6. Mostrar Informações Sobre um Remote

Para visualizar informações detalhadas sobre um repositório remoto, use:

```
git remote show nome-repo
```

Por exemplo:


```
git remote show origin
```

Isso exibirá informações detalhadas, incluindo URLs de fetch e push, branches rastreadas e mais.

7. Buscar Atualizações de um Remote

Para buscar atualizações de um repositório remoto:

```
git fetch nome-repo
```

Por exemplo:

```
git fetch origin
```

Isso recupera alterações do repositório remoto sem fazer merge automático na branch atual.

A ideia aqui foi mostrar alguns exemplos e fluxos que você repetirá ao desenvolver seus projetos.

Um Guia para Solução de Problemas

Se algo der errado, você precisa saber como responder, o Git não é apenas sobre o caminho feliz:

- Você trabalha
- Você faz um commit
- Mescla o branch
- Faz o deploy

Nós amamos o caminho feliz, é por isso que é feliz!

Mas, na maioria das vezes, as coisas não saem como planejado e precisamos saber como reagir nesses casos.

Como você recupera alterações perdidas? Como você reverte erros?

Isso não é apenas sobre o Git, se aplica a todos os projetos.

É exatamente isso que abordarei neste capítulo.

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

Ao trabalhar com Git, pode ser necessário desfazer commits por alguns motivos, como corrigir erros, reverter alterações inesperadas ou limpar o histórico.

O Git fornece três comandos principais para desfazer commits: `git revert`, `git reset` e `git checkout`. Cada um tem um comportamento diferente e deve ser usado de acordo.

git revert

git revert cria um novo commit que desfaz as alterações introduzidas por um commit específico sem modificar o histórico do repositório.

Isso é útil quando você deseja manter o histórico limpo e não remover commits.

Uso:

```
git revert hash-do-commit
```

Exemplo:

1. Verifique o commit que você deseja reverter:

```
git log --oneline

# Retorno
PS C:\ebooks\git-github-ebook-test> git log --oneline
56c9845 revert
3d87987 herge tag 'v1.0' into dev Showw show
d207431 (tag: v1.0, main) Merge branch 'release/v1.0'
28b146b Yes
0a91ea2 Updated commit message
ca382f6 (origin/main) filename3
db14708 new file
5a26f39 (tag: v1.0.0) first commit
PS C:\ebooks\git-github-ebook-test>
```

2. Copie o hash e execute **git revert**:

```
git revert 56c9845
```

```
# Retorno
```

```
PS C:\ebooks\git-github-ebook-test> git log --oneline
```

```
4134ad2 (HEAD -> dev) Revert "revert"
```

```
56c9845 revert
```

```
3d87987 herge tag 'v1.0' into dev Showw show
```

```
d207431 (tag: v1.0, main) Merge branch 'release/v1.0'
```

```
28b146b Yes
```

```
0a91ea2 Updated commit message
```

```
ca382f6 (origin/main) filename3
```

```
db14708 new file
```

```
5a26f39 (tag: v1.0.0) first commit
```

```
PS C:\ebooks\git-github-ebook-test>
```

- Isso cria um novo commit que reverte as alterações do commit **56c9845**.

git reset

`git reset` move o ponteiro HEAD para um commit anterior; isso é usado para modificar o histórico de commits. Ele tem três modos principais:

Modos de `git reset`:

- **Soft (`--soft`)**: Redefine o HEAD para um commit anterior, mas mantém as alterações na área de staging, permitindo que você modifique ou emende o commit.
- **Mixed (`--mixed`)**: Redefine o HEAD para um commit anterior e remove as alterações da área de staging, mas deixa as alterações no diretório de trabalho.
- **Hard (`--hard`)**: Remove completamente o commit e todas as alterações, tanto da área de staging quanto do diretório de trabalho.

Exemplo `--soft`:

A opção `--soft` desfaz o último commit, mas mantém as alterações na área de staging, para que você possa modificar facilmente o commit ou alterar a mensagem do commit. Isso é útil se você deseja refazer um commit sem perder o trabalho que fez.

Você volta para a etapa `git add ..`

```
git reset --soft HEAD~1
```

- Neste exemplo, `HEAD~1` move o HEAD um commit para trás; todas as alterações permanecem em stage, para que você possa re-commitá-las após fazer ajustes.

Exemplo `--mixed`:

A opção `--mixed` redefine o HEAD para o commit especificado, remove as alterações da área de staging, mas as deixa no diretório de trabalho. Este é o comportamento padrão de `git reset` se nenhum modo for especificado.

É semelhante a `--soft`; a principal diferença é que com `--mixed` você volta antes de `git add ..`

```
git reset --mixed HEAD~1
```

- Este comando desfaz o último commit e remove as alterações do stage, mas as alterações permanecem no seu diretório de trabalho, para que você ainda possa modificá-las.

Exemplo **--hard**:

A opção **--hard** redefine tanto a área de staging quanto o diretório de trabalho para corresponder ao commit especificado. Isso significa que todas as alterações são perdidas e não podem ser recuperadas, a menos que você tenha backups ou use **git reflog**.

Ação destrutiva, cuidado aqui!

```
git reset --hard HEAD~1
```

- Isso removerá o último commit e todas as alterações associadas a ele, tanto na área de staging quanto no diretório de trabalho.

git checkout

`git checkout` pode ser usado para desfazer alterações em arquivos individuais ou trocar de branch. Falamos sobre ele apenas para branches até agora.

`git checkout` também é usado para desfazer commits, mas agora `git switch` e `git restore` são recomendados para essas tarefas.

Uso para restaurar arquivos específicos:

`git checkout` pode ser usado para desfazer alterações em arquivos individuais e restaurá-los ao último estado commitado.

```
git checkout -- nome_do_arquivo.txt
```

- Este comando descartará quaisquer alterações não commitadas no arquivo `nome_do_arquivo.txt` e o restaurará ao seu último estado commitado.

O `--` é usado para informar ao Git que você está se referindo a um arquivo (não a uma branch) e que deseja descartar as alterações no diretório de trabalho, restaurando o arquivo ao estado do último commit.

Exemplo `git restore`:

`git restore` é o comando recomendado para restaurar arquivos, e é mais intuitivo do que usar `git checkout` para esse propósito.

```
git restore nome_do_arquivo.txt
```

- Este comando descartará quaisquer alterações não commitadas no arquivo `nome_do_arquivo.txt` e o restaurará à versão do último commit, assim como `git checkout -- nome_do_arquivo.txt`.

Recuperando Commits Perdidos

Se você perder acidentalmente um commit no Git, poderá recuperá-lo usando comandos como `git reflog` e `git fsck`. Essas ferramentas ajudam você a identificar e restaurar commits que parecem ter sido perdidos.

Com certeza, é um tipo de comando que você não usa muito, mas precisa saber que existe.

Recuperando um Commit Perdido com `git reflog`

`git reflog` registra atualizações na ponta das branches (HEAD), permitindo que você veja o histórico de commits e outras ações do Git, mesmo que o commit não faça mais parte de nenhuma branch ou referência.

1. Visualize as mudanças recentes do HEAD:

```
git reflog
```

- Este comando exibirá uma lista de mudanças recentes no HEAD, incluindo commits, merges, rebases e resets.

2. Identifique o hash do commit perdido.

```

PS C:\ebooks\git-github-ebook-test> git reflog
4134ad2 (HEAD -> dev) HEAD@{0}: reset: moving to HEAD~1
73ac997 HEAD@{1}: commit: git reset example
4134ad2 (HEAD -> dev) HEAD@{2}: reset: moving to HEAD~1
bed2cc3 HEAD@{3}: commit: git reset example
4134ad2 (HEAD -> dev) HEAD@{4}: revert: Revert "revert"
56c9845 HEAD@{5}: commit: revert
3d87987 HEAD@{6}: merge vv1.0: Merge made by the 'ort' strategy.
28b146b HEAD@{7}: checkout: moving from main to dev
d207431 (tag: vv1.0, main) HEAD@{8}: checkout: moving from main to main
d207431 (tag: vv1.0, main) HEAD@{9}: merge release/v1.0: Merge made by
the 'ort' strategy.
ca382f6 (origin/main) HEAD@{10}: checkout: moving from release/v1.0 to
main
28b146b HEAD@{11}: checkout: moving from dev to release/v1.0
28b146b HEAD@{12}: checkout: moving from feature/feature-article to dev
28b146b HEAD@{13}: checkout: moving from dev to feature/feature-article
28b146b HEAD@{14}: reset: moving to HEAD

```

- Examine a saída para encontrar o commit que você precisa recuperar. Cada entrada está associada a uma referência, como **HEAD@{2}**, e um hash de commit.

3. Faça checkout ou reset para o commit perdido:

Depois de identificar o hash do commit, você pode fazer checkout ou reset para esse commit.

- **Usando git checkout:** Isso moverá seu HEAD para o commit específico, mas não modificará seu diretório de trabalho ou área de staging.

```
git checkout hash-do-commit
```

- **Usando git reset --hard:** Isso redefinirá seu HEAD e diretório de trabalho para o commit específico, descartando quaisquer alterações não commitadas.

```
git reset --hard hash-do-commit
```

Exemplo:

Visualize o reflog para encontrar o commit perdido:

```
git reflog
```

Exemplo de saída:

```
8b3dac5 (HEAD, dev) HEAD@{2}: commit: git checkout
4134ad2 HEAD@{3}: reset: moving to HEAD~1
73ac997 HEAD@{4}: commit: git reset example
4134ad2 HEAD@{5}: reset: moving to HEAD~1
bed2cc3 HEAD@{6}: commit: git reset example
4134ad2 HEAD@{7}: revert: Revert "revert"
56c9845 HEAD@{8}: commit: revert
3d87987 HEAD@{9}: merge vv1.0: Merge made by the 'ort' strategy.
```

Recupere o commit fazendo checkout dele:

```
git checkout 73ac997
```

Ou, se você quiser redefinir sua branch para este commit, use:

```
git reset --hard 56c9845
```

Se você executar **git reflog** novamente, verá que todas as alterações são feitas como um novo commit:

```
56c9845 (HEAD) HEAD@{0}: reset: moving to 56c9845
8b3dac5 (dev) HEAD@{1}: reset: moving to 8b3dac5
73ac997 HEAD@{2}: checkout: moving from dev to 73ac997
8b3dac5 (dev) HEAD@{3}: commit: git checkout
4134ad2 HEAD@{4}: reset: moving to HEAD~1
73ac997 HEAD@{5}: commit: git reset example
4134ad2 HEAD@{6}: reset: moving to HEAD~1
bed2cc3 HEAD@{7}: commit: git reset example
4134ad2 HEAD@{8}: revert: Revert "revert"
56c9845 (HEAD) HEAD@{9}: commit: revert
```

Encontrando Commits Pendurados com `git fsck`

Se você não conseguir encontrar o commit no relog, ele pode estar "pendurado", o que significa que não é referenciado por nenhuma branch ou tag, mas ainda existe no banco de dados do Git. Você pode procurar por esses commits pendurados usando `git fsck`.

Imagine que exista um commit perdido no repositório, você pode encontrar com o comando `git fsck`. Os commits perdidos podem acontecer quando esquecemos de fazer merge de branches e deletamos o trabalho.

1. Execute `git fsck` para encontrar commits pendurados:

```
git fsck --lost-found
```

- Isso listará objetos que não fazem parte de nenhuma referência, incluindo commits pendurados.

2. Inspeção os commits:

Procure por entradas rotuladas como "dangling commit" na saída e use `git show` para inspecioná-las.

```
git show hash-do-commit
```

Exemplo:

Encontre commits pendurados com `git fsck`:

```
git fsck --lost-found
```

Exemplo de saída:

```
dangling commit 047885dbe5a76825fa1780cecb741f76eeb9ec35
dangling commit 18baee9d1b898ee1d82504174672b80d63aafa9a
dangling commit 227a6260d4d24c359aa5d4bcf667326c5cf10724
dangling tree 36e529c29f83cad089c8ddc57b7a1fa329564e87
dangling commit 413e484a74bef6b89674ecc6108f5f4364b9395f
dangling commit 726c4249600cb31b5bc8b52238faf3c3e31b4f68
dangling commit 73ac997c040ffda224e4a387614e2f860a8e0905
dangling commit 74a4b8ab890536d2ab8dacf076eaa64a5b24e486
dangling tree 96fc3be44051489ebc3303a66c07108bbcb563da
dangling commit a2d081673eb8bdd9f8e3159d787050c77116e901
dangling commit bed2cc3d74faff05c84ba21f84b29ab37357af63
dangling commit c2df1cab9005791812dd7efb690f9ce5f799bf70
dangling commit c8a104a4a311e3ee8cc2a696aeae5e4dc46cbeea
dangling commit fcde575ac34212c1187cee90604d35f0f6b00202
```

Inspecione o commit pendurado:

```
git show 047885dbe5a76825fa1780cecb741f76eeb9ec35
```

O retorno:

```
commit 047885dbe5a76825fa1780cecb741f76eeb9ec35
Author: Lorenzo Uriel
Date: Sun Feb 16 21:01:01 2025 -0300
```

Updated commit message

```
diff --git a/filename.txt b/filename.txt
index 8d1c8b6..91b27f5 100644
--- a/filename.txt
+++ b/filename.txt
@@ -1,2 @@

+Lorenzo Uriel
\ No newline at end of file
```

- Exibe todos os detalhes do commit, permitindo que você decida se é o commit que deseja recuperar.

Mais Sobre `git status`, `git log`, `git show` e `git diff`

Sim... nós já os usamos acima.

Compreender os comandos `git status`, `git log`, `git show` e `git diff` pode ajudá-lo a rastrear as alterações e gerenciar seu repositório Git de uma forma mais inteligente.

Com esses comandos, você pode verificar o estado atual do seu repositório, ajudá-lo a explorar os históricos de commit e permitir que você veja as diferenças entre vários estados de seus arquivos.

É como um monte de instruções de consulta.

`git status`

`git status` fornece uma visão geral do estado atual do diretório de trabalho e da área de staging. Ele ajuda você a rastrear quais alterações foram adicionadas ao stage, quais não foram e quais arquivos não estão sendo rastreados.

```
git status
```

Exemplo de saída:

```
HEAD detached from 73ac997
Changes not staged for commit:
  (use "git add ..." to update what will be committed)
  (use "git restore ..." to discard changes in working directory)
    modified:   README.md
    modified:   filename.txt
    modified:   filename2.txt
    modified:   filename3.txt

Untracked files:
  (use "git add ..." to include in what will be committed)
    filename3 copy.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

- Changes to be committed: Esses arquivos foram adicionados ao stage e estão prontos para serem commitados.

- Untracked files: Esses arquivos ainda não estão sendo rastreados pelo Git.

git log

git log mostra o histórico de commits do repositório, permitindo que você explore commits anteriores, suas mensagens, autores e timestamps. Você também pode usar várias opções para filtrar e formatar a saída do log.

```
git log
```

Exemplo de saída:

```
commit 71333c330b93ffe20e02e6bac2b9a57ce15e355c (HEAD)
Author: Lorenzo Uriel
Date:   Mon Mar 10 21:56:23 2025 -0300

    oh yeah

commit 56c9845e9e0ff4298def3b693456f8857d82824b
Author: Lorenzo Uriel
Date:   Sun Mar 9 21:33:00 2025 -0300

    revert

commit 3d87987f33e66c859aa25f6a950995815c8caf50
Merge: 28b146b d207431
Author: Lorenzo Uriel
Date:   Wed Mar 5 23:14:00 2025 -0300

    show
```

- Commit hash: O identificador exclusivo para o commit.
- Author: Quem fez o commit.
- Date: Quando o commit foi feito.
- Commit message: Uma breve descrição das alterações feitas nesse commit.

Você também pode usar **git log --online** para detalhes mais resumidos:

```
git log --online
```

Exemplo de saída:


```
5b07142 (HEAD) html example for git show
71333c3 oh yeah
56c9845 revert
3d87987 herge tag 'vv1.0' into dev Showw show
d207431 (tag: vv1.0, main) Merge branch 'release/v1.0'
28b146b Yes
0a91ea2 Updated commit message
ca382f6 (origin/main) filename3
db14708 new file
5a26f39 (tag: v1.0.0) first commit
```

git show

git show permite que você veja informações detalhadas sobre um commit específico, incluindo a mensagem do commit, autor, data e as alterações feitas.

Foi o comando que usamos no tópico acima.

```
git show 5b07142
```

Exemplo de saída:

```
commit 5b071427e8369b85a56a5ee6b5388a2e2586306c (HEAD)
Author: Lorenzo Uriel
Date:   Mon Mar 10 22:02:09 2025 -0300
```

```
html example for git show
```

```
diff --git a/README.md b/README.md
index 49d8b13..a71ae21 100644
--- a/README.md
+++ b/README.md
@@ -1,3 +1,3 @@
- "# git-github-ebook-test"

- git revert test
+ git revert testfcfdefd
diff --git a/filename.txt b/filename.txt
index 8d1c8b6..8ebae4c 100644
--- a/filename.txt
+++ b/filename.txt
@@ -1 +1,2 @@

+ sdfdfdfadf
\ No newline at end of file
diff --git a/filename2.txt b/filename2.txt
```

- Isso mostrará as alterações feitas no commit (o diff) junto com outros metadados.

git diff

git diff mostra as diferenças entre vários estados do repositório, como alterações entre o diretório de trabalho e o último commit, entre dois commits ou entre branches.

Compare o diretório de trabalho com o último commit:

```
git diff
```

Exemplo de saída:

```
diff --git a/index.html b/index.html
index 64b907a..a24d290 100644
--- a/index.html
+++ b/index.html
@@ -3,7 +3,7 @@

-   My First HTML Page
+   My Second HTML Page

Welcome to My First HTML Page
```

- Isso mostra as alterações entre o diretório de trabalho e o último commit. O sinal **+** indica uma linha adicionada e o sinal **-** indica uma linha removida.

Comparar dois commits:

Você também pode comparar as alterações entre dois commits especificando seus hashes:

```
git diff 5b07142 e0f5e31
```

- Isso mostra as diferenças entre os dois commits especificados.

Exemplo de saída:

```
diff --git a/index.html b/index.html
index a24d290..685af61 100644
--- a/index.html
+++ b/index.html
@@ -3,7 +3,7 @@

-   My Second HTML Page
+   My Third Example HTML Page

Welcome to My First HTML Page
```

Comparar branches:

Para ver as diferenças entre branches, use:

```
git diff main dev
```

- Isso mostra as diferenças entre as duas branches, ajudando você a ver quais alterações estão em uma branch, mas não na outra.

Exemplo de saída:

```
diff --git a/README.md b/README.md
index f3e1357..f70444d 100644
--- a/README.md
+++ b/README.md
@@ -1,3 @@
- "# git-github-ebook-test"
+
+git reset example
\ No newline at end of file
diff --git a/filename3.txt b/filename3.txt
index 8d1c8b6..3ad7dfd 100644
--- a/filename3.txt
+++ b/filename3.txt
@@ -1,2 @@

+Changes on files
\ No newline at end of file
```

Mais Sobre `git commit --amend`

Um dia você fez uma mensagem de commit grande e bonita e 5 minutos depois você encontra uma mudança de código e precisa commitar novamente?

Por que não apenas commitar para o mesmo último commit se for apenas uma pequena mudança? `--amend` pode ajudar com isso.

O comando `git commit --amend` é usado para modificar o commit mais recente no Git. Ele permite que você corrija a mensagem do commit, adicione ou remova alterações do commit ou até mesmo atualize as alterações e a mensagem de uma só vez.

Este comando é particularmente útil quando:

- Você percebe que a mensagem do commit precisa ser aprimorada.
- Você esqueceu de incluir algumas alterações no commit.

Uso:

```
git commit --amend
```

Por padrão, executar este comando abrirá o editor de commit para permitir que você modifique a mensagem do commit. Você também pode passar a opção `-m` para atualizar diretamente a mensagem do commit.

Exemplo 1: Modificar a Mensagem do Commit

Se você quiser atualizar a mensagem do último commit, pode usar:

```
git commit --amend -m "Mensagem de commit atualizada"
```

- Isso substituirá a mensagem do último commit pela nova.

Exemplo 2: Adicionar Alterações Perdidas

Se você perceber que esqueceu de adicionar algumas alterações ao stage para o último commit, você pode:

1. Adicionar as alterações que você perdeu ao stage:

```
git add .
```

2. Emendar o commit com as alterações adicionadas ao stage:

```
git commit --amend
```

- Isso abrirá o editor de commit novamente, onde você pode modificar a mensagem do commit, se desejar. Se você não quiser alterar a mensagem, simplesmente salve e feche o editor.

O commit emendado agora incluirá as alterações originais e as alterações recém-adicionadas ao stage.

Exemplo 3: Substituir Completamente o Commit (Mensagem + Alterações)

Se você quiser alterar a mensagem do commit e as alterações, adicione as alterações ao stage primeiro e, em seguida, execute o comando **--amend**:

```
git add .  
git commit --amend -m "Nova mensagem de commit com alterações atualizadas"
```

- Isso substituirá o último commit pelas novas alterações e a nova mensagem de commit.

git commit --amend reescreve o último commit, então ele muda seu hash de commit. Isso pode causar problemas se o commit já tiver sido enviado para um repositório compartilhado. Recomenda-se emendar apenas commits que ainda não foram enviados ou usá-lo com cautela se você já tiver compartilhado suas alterações.

Um Guia para Boas Práticas

Você precisa aprender algumas práticas recomendadas antes de fazer seu próximo commit.

Qual é o melhor tipo de mensagem? Ou o melhor nome para os branches?

Sempre buscamos aplicar as melhores práticas em nossos projetos. Se você ainda não faz isso, é hora de começar.

Ainda mais quando trabalhamos em equipe, compartilhamos repositórios e fazemos revisões de código.

Sabe aquele commit que quebrou todo o código e a mensagem só dizia: "Agora vai!"? Pois é disso que estou falando.

Então o que fazer?

Quais são as melhores práticas quando se trata de Git e GitHub?

Commits Claros e Frequentes

Com certeza o mais importante e o primeiro da lista: Escreva de uma forma que as pessoas possam entender!

Faça commits pequenos e frequentes para manter um histórico de alterações claro.

Use mensagens de commit descritivas e padronizadas explicando o que foi alterado e o porquê.

Exemplo:

```
git commit -m "fix: login issue by correcting password validation"
```

Evite commits genéricos como apenas "fix" ou "update". Prefira algo com mais detalhes.

Faça commits pequenos, onde cada um aborda uma única alteração. Pare de fazer commits e escrever um texto inteiro explicando o que mudou, isso é péssimo quando precisamos reverter.

Dicas para Mensagens de Commits

O que escrever na sua mensagem de commit?

Eu sei que você já teve essa dúvida, principalmente quando fazemos várias alterações e precisamos especificar tudo em um único commit.

Eu sigo algumas práticas que considero boas nos meus projetos. Eles são:

- feat: adicionado suporte para backup automático
- fix: ajustado normalização de dados
- revert: revertida alteração no esquema do banco
- delete: removida tabela [old_logs]
- update: atualizada estrutura do banco de dados
- config: ajustada conexão do banco de dados em .env
- ci: adicionada etapa de migração no pipeline
- docs: documentada estrutura da tabela
- test: adicionados testes para funções ETL

A ideia é colocar no início uma descrição do que é o commit, seguido por um detalhe claro do que foi realizado.

Commits frequentes são sempre melhores! Não espere para fazer tudo de uma vez.

Gerenciamento de Branch

Use branches para gerenciar o desenvolvimento de recursos, implantações, lançamentos e outras tarefas. Adote um fluxo de trabalho que seja bom para sua equipe, como:

- **Git Flow:** A maneira mais comum e popular de trabalhar, dividida por branches principais (**dev**, **main**) e secundários (**feature**, **release**, **hotfix**).
- **Trunk-Based Development:** O projeto é centralizado em apenas um branch exclusivo para toda a equipe, o **main** e temos o conceito de **feature-branches** para alterações.

Dicas para nomes de branches

E sempre nos encontramos com a mesma pergunta, qual título colocar neste branch?

Branches principais

O padrão que vejo na indústria para as branches principais é:

- **dev** - development
- **qa** - test
- **main**

Branches de alteração

Para branches de alterações, sempre especifique o objetivo e adicione uma mensagem descritiva:

- **feature/add-backup-automation**
- **fix/fix-null-values-in-report**
- **hotfix/fix-production-database-issue**
- **refactor/normalize-user-table**
- **release/1.0.0**

Uma boa dica é relacionar o ID da tarefa e o título da tarefa, se seu backlog suportar:

- **feature/task544-add-backup-automation**
- **fix/task17-fix-null-values-in-report**

Rebase

Outra boa dica ao trabalhar com branches é o comando **git rebase**. Você pode reordenar ou editar commits antes de mesclar. Isso ajuda a manter um histórico linear e legível:

```
git rebase -i HEAD~3 # Reordene ou edite os últimos três commits interativamente
```

*Evite rebasear branches compartilhados como **main** ou **dev** para evitar conflitos e confusão para os outros.*

Delete

Exclua regularmente branches antigos ou desatualizados para manter o repositório limpo:

```
git branch -d feature/task544-add-backup-automation # Exclua um branch mesclado  
git branch -D feature/task544-add-backup-automation # Exclua um branch não mesclado à força
```

Use .gitignore

Sempre configure um arquivo **.gitignore** apropriado para seu projeto para evitar versionamento de arquivos desnecessários.

Exclua arquivos de build, dependências, credenciais e todas as configurações de ambiente possíveis.

Recomendo que você use o [site toptal](#), ele gera os arquivos **.gitignore** mais completos para cada linguagem. Este é um [exemplo python gerado com toptal](#).

Você também pode verificar os modelos prontos feitos pelo github no [repositório github/gitignore](#).

Git Hooks

Com Git Hooks você pode automatizar regras e padrões que precisam ser aplicados em vários estágios do seu fluxo de trabalho.

Pre-commit Hooks

Use pre-commit hooks para executar verificações automatizadas antes que as alterações sejam confirmadas. Esses hooks podem impor ao código seus padrões de qualidade predefinidos, alguns exemplos:

- Executar linters para capturar erros de sintaxe ou estilo.
- Executar testes de unidade para garantir que as alterações não quebrem o código.
- Validar mensagens de confirmação para seguir um formato consistente.

Aqui você pode detectar erros antecipadamente e impedir que o desenvolvedor envie qualquer código que possa afetar o projeto principal.

Post-receive Hooks

Use post-receive hooks para automatizar tarefas após as alterações serem enviadas para um repositório remoto. Esses hooks são ideais para:

- Atualizar ambientes de preparação ou produção automaticamente.
- Acionar pipelines de CI/CD para implantar ou testar alterações.
- Enviar notificações para sua equipe sobre novas atualizações.

Claro, minimizando erros e criando uma implantação inicial de forma automática.

Um Guia para Tipos de Workflows

O que é um workflow? E por que você deveria usar um?

É isso que quero explicar hoje. Tenho certeza de que você começará a usá-lo ainda hoje.

Um workflow define como você e sua equipe podem colaborar usando Git e GitHub. Escolher o workflow correto dependerá de como sua equipe está estruturada, da complexidade do projeto e de suas estratégias para lançar uma nova versão.

Vou explorar os workflows mais comuns e seus casos de uso.

Trunk-Based Development

O mais utilizado, com certeza. Você provavelmente já usou esse modelo e nem sabia que ele tinha um nome.

Mas como ele funciona?

Os desenvolvedores trabalham diretamente na branch **main** ou criam **feature branches de curta duração**. Todas as alterações são mescladas na **main** várias vezes ao dia.

Isso resulta em integração frequente, minimizando conflitos de merge. Com isso, temos entrega rápida, colaboração em tempo real e menos conflitos de merge.

Passos do Workflow

1. Iniciar o trabalho:

```
git checkout -b feature-article
```

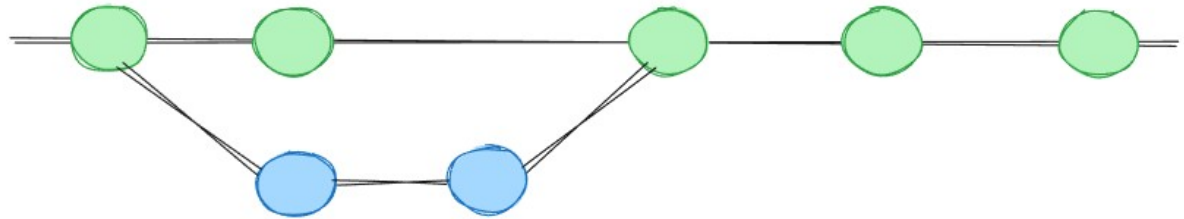
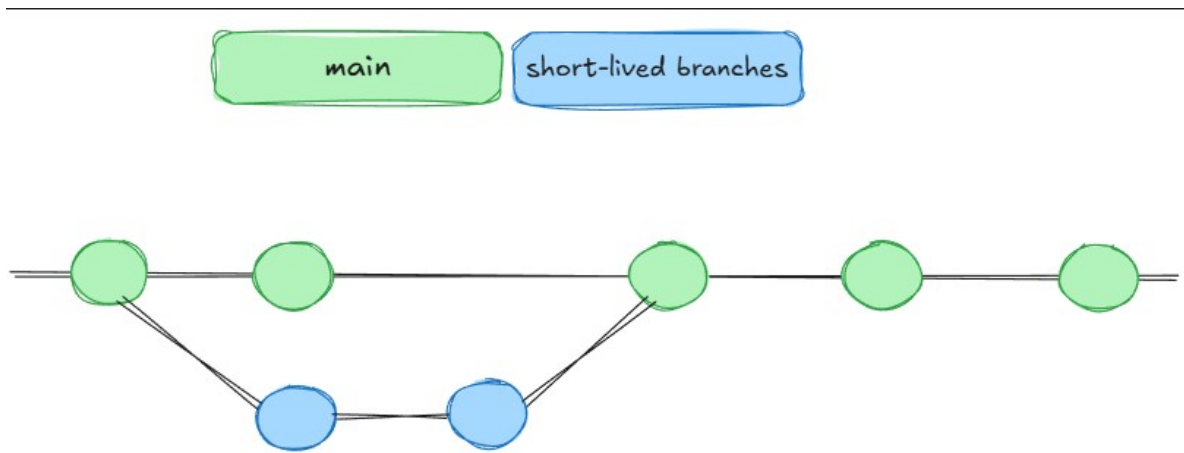
2. Fazer alterações e enviar frequentemente:

```
git commit -m "Finish Article"
```

3. Mesclar a branch:

```
git checkout main  
git merge feature-article  
git push origin main
```

Diagrama



GitFlow

Em vez de ter apenas uma branch principal, sempre temos duas: **main** e **dev**.

Com base em **dev**, podemos criar outras branches para diferentes propósitos, como:

- **feature branches**: Criadas a partir da **dev**, mescladas de volta quando concluídas.
- **release branches**: Criadas a partir da **dev** ao preparar um lançamento.
- **hotfix branches**: Criadas a partir da **main** para corrigir problemas urgentes e mescladas tanto na **main** quanto na **dev**.

A branch **main** sempre armazena o código de produção, enquanto **dev** integra todas as feature branches. Desenvolvemos na branch **dev** para evitar mexer no código de produção.

GitFlow é a melhor opção para propósitos de CI/CD.

Passos do Workflow

1. Inicializar o GitFlow:

```
git flow init
```

Durante a inicialização, será feito um pequeno questionário para configurar o projeto.

Which branch should be used for bringing forth production releases?

- dev
- main

Branch name for production releases: [main]

Which branch should be used for integration of the "next release"?

- dev

Branch name for "next release" development: [dev]

How to name your supporting branch prefixes?

Feature branches? [feature/]

Bugfix branches? [bugfix/]

Release branches? [release/]

Hotfix branches? [hotfix/]

Support branches? [support/]

Version tag prefix? [v]

Hooks and filters directory? [C:/ebooks/git-github-ebook-test/.git/hooks]

O próprio `git flow init` criará toda a estrutura.

2. Iniciar uma nova feature:

```
git flow feature start feature-article
```

- Esta ação cria um novo branch de recurso baseado em `dev` e alterna para ele

3. Finalizar a feature:

```
git flow feature finish feature-article
```

- Mescla `feature-article` em `dev`
- Remove o branch feature
- Volta para o branch `dev`

4. Publicar a feature:

```
git flow feature publish feature-article
```

- Este comando publica o próprio recurso no repositório remoto para outros usuários.

5. Criar uma branch de release:

```
git flow release start v1.0
```

- Cria uma branch de lançamento criada a partir da branch **dev**.

6. Finalizar & deploy da release:

```
git flow release finish 'v1.0'
```

- Mescla o branch **release** em **main**
- Faz a Tag do **release** com o nome.
- Mescla o branch **release** em **dev** novamente
- Então, remove o branch **release**

A mensagem de prompt é a seguinte:

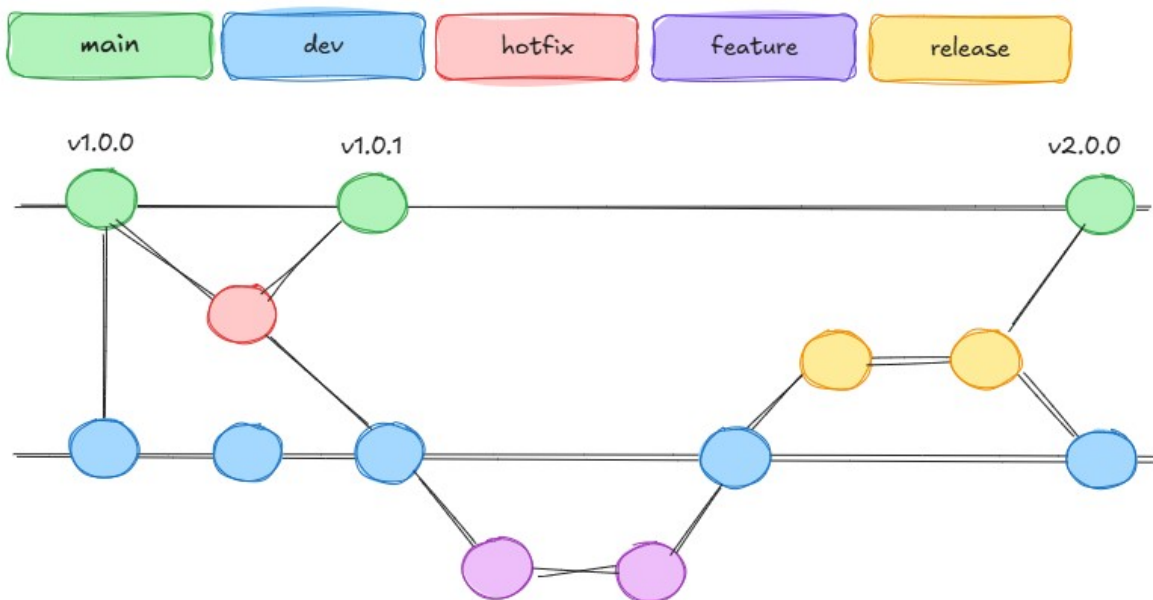
```
C:\ebooks\git-github-ebook-test>git flow release finish 'v1.0'
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
Merge made by the 'ort' strategy.
Already on 'main'
Your branch is ahead of 'origin/main' by 3 commits.
  (use "git push" to publish your local commits)
Switched to branch 'dev'
Merge made by the 'ort' strategy.
Deleted branch release/v1.0 (was 28b146b).
```

Summary of actions:

- Release branch 'release/v1.0' has been merged into 'main'
- The release was tagged 'v1.0'
- Release tag 'v1.0' has been back-merged into 'dev'
- Release branch 'release/v1.0' has been locally deleted
- You are now on branch 'dev'

Se quiser saber mais sobre os comandos, recomendo este [GitFlow cheatsheet](#).

Diagrama



Feature Branch

Ideal para equipes que trabalham em múltiplas features simultaneamente. Cada feature é desenvolvida em uma branch separada e só é mesclada na **main** após revisão.

Passos do Workflow

1. Criar uma feature branch:

```
git checkout -b feature-article
```

2. Trabalhar e realizar commits:

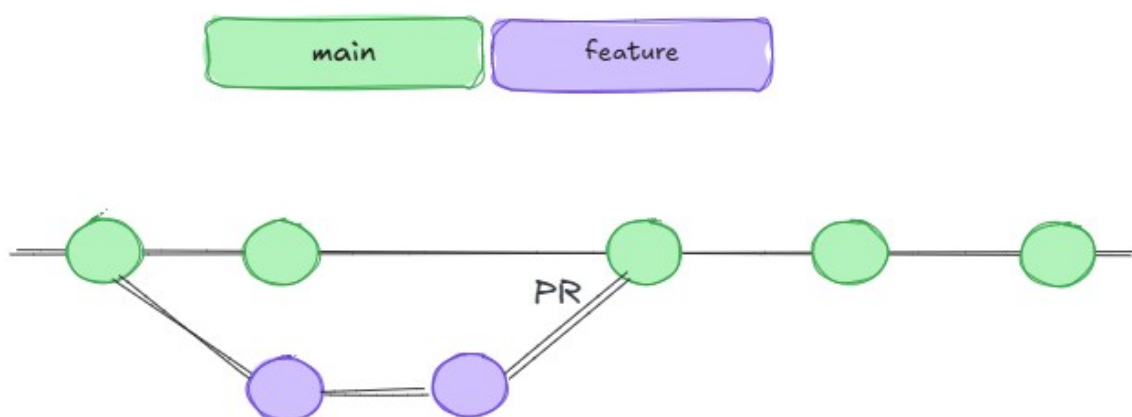
```
git add .  
git commit -m "Added Feature Branch Topic"
```

3. Enviar a branch e abrir um Pull Request:

```
git push origin feature-article
```

4. Após a aprovação, mesclar na **main** no repositório remoto e local.

Diagrama



Forking Workflow

Os desenvolvedores fazem um fork do repositório em vez de trabalhar diretamente nele. Este modelo é recomendado e amplamente utilizado em projetos Open-Source.

Os desenvolvedores clonam o repositório, criam uma branch, fazem alterações e enviam suas mudanças para seu próprio fork. Depois disso, submetem as alterações como um **Pull Request (PR)**.

Passos do Workflow

1. Fazer um fork do repositório no GitHub.
2. Clonar o repositório forkado:

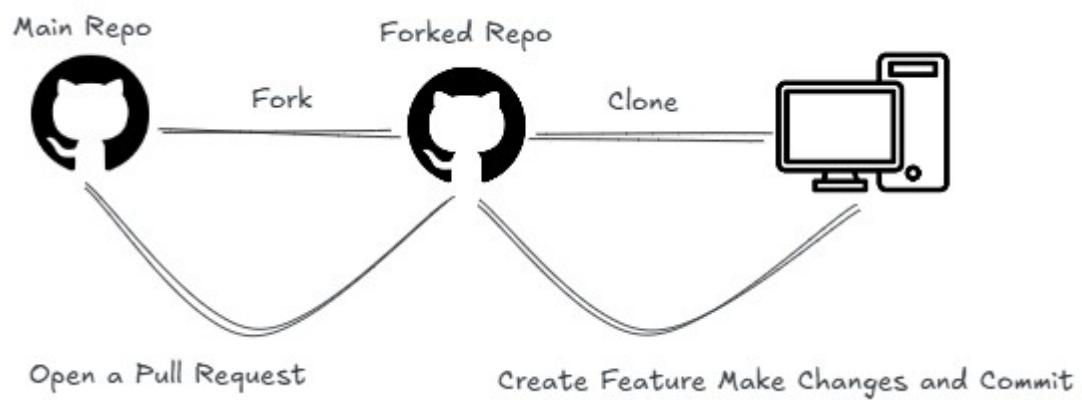
```
git clone https://github.com/lorenzouriel/ebook-git-github.git
```

3. Criar uma branch e fazer alterações:

```
git checkout -b feature-article  
git add .  
git commit -m "Added feature article"  
git push origin feature-article
```

4. Submeter um Pull Request para o repositório original.

Diagrama



Ao aprender isso, você poderá decidir qual workflow mais se adapta a você e sua equipe, lembrando que em muitos casos já utilizamos um sem nem perceber.

O Fim

Obrigado.