

# guia-git

## Sumário:

[Sumário](#):

[Introdução](#)

[1- Introdução ao Git](#)

[2. Instalação e Configuração do Git](#)

[Instalação do Git](#)

[Windows](#)

[macOS](#)

[Linux](#)

[Configuração Inicial do Git](#)

[Configurando Nome de Usuário e Email](#)

[Configurando Editor Padrão](#)

[Configurando Ferramentas de Diff](#)

[Verificando a Configuração](#)

[3. Conceitos Básicos do Git](#)

[Repositório \(Local e Remoto\)](#)

[Branches \(Ramificações\)](#)

[Commits](#)

[SHA-1 Hash](#)

[Árvore de Diretórios](#)

[4. Comandos Básicos do Git](#)

Inicialização de um Repositório ( `git init` )

Clonagem de Reppositórios ( `git clone` )

Status do Repositório ( `git status` )

Adição de Arquivos ( `git add` )

Commit de Alterações ( `git commit` )

Visualização de Histórico ( `git log` )

Exercícios Práticos

## 5. Trabalhando com Branches

Conceito de Branches

Criação de Branches

Navegação entre Branches

Mesclagem de Branches

Resolução de Conflitos

Branches Remotas e Rastreamento

Exercícios Práticos

## 7. Manipulação e Reescrita de Histórico

Revertendo Commits ( `git revert` )

Resetando Commits ( `git reset` )

Rebasing ( `git rebase` )

Stashing ( `git stash` )

Exercícios Práticos

## 8. Trabalho Colaborativo

Forks e Pull Requests

Revisão de Código

Estratégias de Branching

Exercícios Práticos

## 9. Boas Práticas e Convenções

Mensagens de Commit Claras e Descritivas

Padrões de Branches

Versionamento Semântico

Revisão de Código

Gerenciamento de Conflitos

## 10. Recursos Adicionais e Referências

Documentação Oficial do Git

Livros Recomendados

Tutoriais e Cursos Online

Ferramentas e Plugins

Comunidades e Fóruns de Discussão

---

# Introdução

Olá, muito obrigado por ter acessado esse guia, ele foi pensado e desenvolvido com muito carinho com o objetivo de ajudar você que está começando na programação. ❤

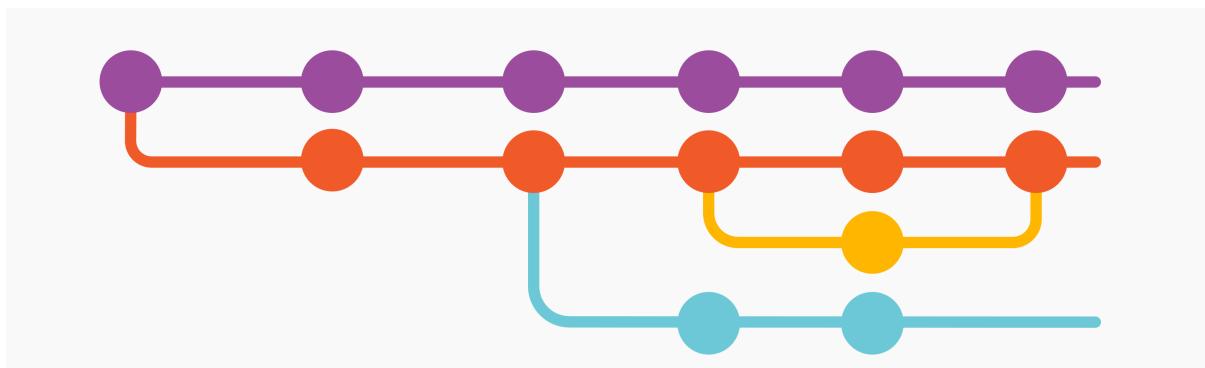
Não esquece de me seguir no Instagram [@umporcentoprog](#) que lá eu trago conteúdo educacional de programação todos os dias, e é por lá que eu solto esses guias gratuitos, como esse que você está lendo!

---

## 1- Introdução ao Git

### O que é o Git?

Git é um sistema de controle de versão distribuído. Mas o que isso realmente significa? 🤔 Em termos simples, o Git é uma ferramenta que ajuda você a acompanhar as alterações no seu código ao longo do tempo. Com ele, você pode voltar no tempo e ver versões antigas do seu código, comparar mudanças e até colaborar com outras pessoas de maneira eficiente.



### História e Evolução do Git

O Git foi criado em 2005 por Linus Torvalds, o mesmo cara que criou o Linux. Naquela época, os desenvolvedores do kernel Linux precisavam de uma ferramenta poderosa para gerenciar as versões do código. Daí nasceu o Git, uma ferramenta que rapidamente se tornou popular pela sua velocidade e eficiência.

### Vantagens e Desvantagens do Uso do Git

#### Vantagens:

1. **Distribuído:** Diferente de outros sistemas, o Git permite que cada desenvolvedor tenha uma cópia completa do repositório, incluindo o histórico de todas as mudanças. Isso significa que você pode trabalhar offline e ainda ter acesso a todo o histórico de versões.
2. **Velocidade:** As operações no Git são extremamente rápidas, o que facilita a vida quando você está lidando com grandes projetos.
3. **Flexibilidade:** O Git oferece várias formas de trabalhar, adaptando-se ao seu fluxo de trabalho, seja ele mais simples ou mais complexo.
4. **Integridade:** O Git utiliza SHA-1, um algoritmo de hashing, para garantir a integridade dos dados. Cada commit é identificado por um hash único, tornando qualquer tentativa de alteração muito fácil de ser detectada.

#### **Desvantagens:**

1. **Curva de aprendizado:** Para quem está começando, o Git pode parecer um pouco complicado, especialmente por conta dos muitos comandos e conceitos novos.
  2. **Complexidade:** Em grandes equipes ou projetos complexos, gerenciar branches e merges pode se tornar um pouco desafiador.
- 

## **2. Instalação e Configuração do Git**

### **Instalação do Git**

#### **Windows**

Para instalar o Git no Windows, siga estes passos:

1. Acesse o site oficial do Git e baixe o instalador: [git-scm.com](https://git-scm.com)
2. Execute o instalador e siga as instruções. Durante a instalação, você pode optar por usar Git Bash ou a linha de comando padrão do Windows (recomendado: Git Bash).
3. Após a instalação, abra o Git Bash e digite `git --version` para verificar se o Git foi instalado corretamente.

## macOS

No macOS, você pode instalar o Git de várias maneiras, mas a mais comum é usando o Homebrew:

1. Se ainda não tiver o Homebrew instalado, instale-o executando o seguinte comando no Terminal:

```
/bin/bash -c "$(curl -fsSL <https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh>)"
```

2. Depois, instale o Git com o Homebrew:

```
brew install git
```

3. Verifique a instalação com:

```
git --version
```

## Linux

A instalação do Git no Linux pode variar dependendo da distribuição que você está usando. Aqui estão os comandos para as distribuições mais populares:

- **Ubuntu/Debian:**

```
sudo apt update  
sudo apt install git
```

- **Fedorá:**

```
sudo dnf install git
```

- **Arch Linux:**

```
sudo pacman -S git
```

Verifique a instalação com:

```
git --version
```

## Configuração Inicial do Git

Depois de instalar o Git, você precisa configurá-lo com suas informações pessoais. Isso é importante para que os commits que você fizer sejam corretamente atribuídos a você.

### Configurando Nome de Usuário e Email

Execute os seguintes comandos no terminal (ou Git Bash, se estiver no Windows):

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

Isso configura seu nome de usuário e email globalmente, ou seja, para todos os repositórios que você criar ou clonar.

### Configurando Editor Padrão

O Git usa um editor de texto para algumas operações, como editar mensagens de commit. Você pode configurar seu editor preferido com:

- **VS Code:**

```
git config --global core.editor "code --wait"
```

- **Vim:**

```
git config --global core.editor "vim"
```

- **Nano:**

```
git config --global core.editor "nano"
```

## Configurando Ferramentas de Diff

O Git também permite configurar ferramentas de diffs e merges personalizadas. Por exemplo, para usar o `meld` como ferramenta de diff, use:

```
git config --global diff.tool meld  
git config --global difftool.prompt false
```

## Verificando a Configuração

Para verificar todas as configurações atuais do Git, execute:

```
git config --list
```

Isso mostrará uma lista de todas as configurações que você definiu.

Pronto! Agora você deve estar com o Git instalado e configurado corretamente no seu sistema. No próximo tópico, vamos explorar os conceitos básicos do Git, para que você entenda como ele funciona por baixo dos panos. 

---

## 3. Conceitos Básicos do Git

Para dominar o Git, é crucial entender seus conceitos básicos. Vamos mergulhar nos principais componentes que fazem o Git funcionar de maneira eficiente e poderosa.

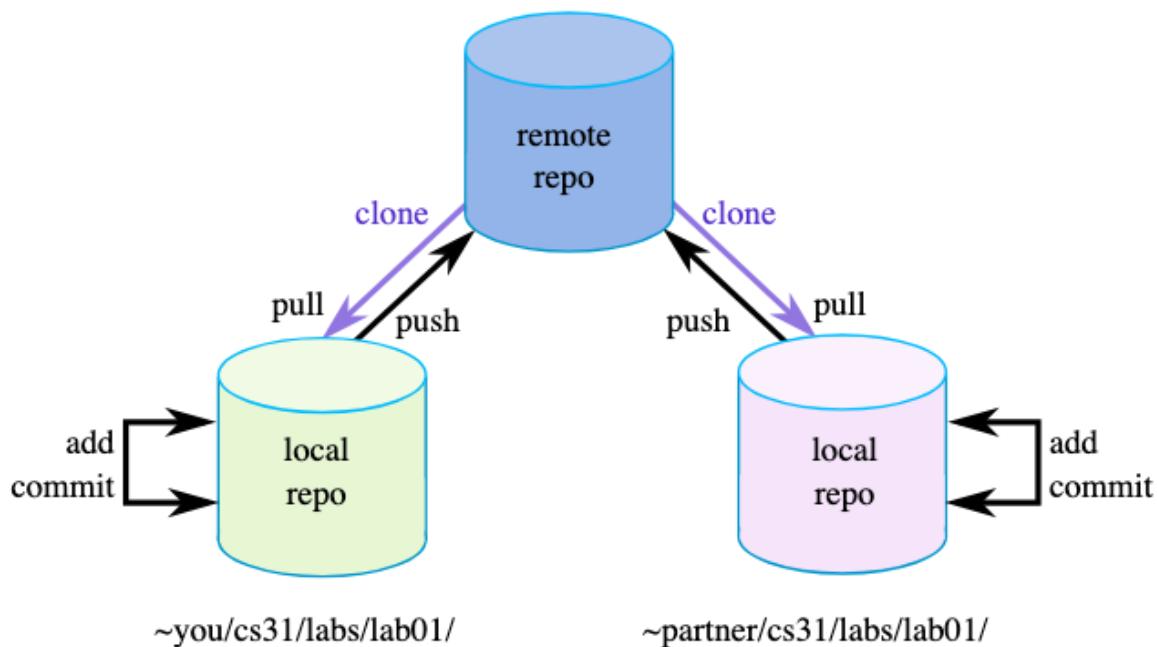
### Repositório (Local e Remoto)

Um repositório Git é onde todo o histórico do seu projeto é armazenado. Existem dois tipos de repositórios:

- 1. Repositório Local:** É o repositório que está no seu computador. Ele contém todo o histórico do projeto e permite que você faça operações como commits, branches e merges sem precisar de uma conexão com a internet.

Em um repositório local, você pode criar branches, fazer commits e visualizar o histórico das alterações.

2. **Repositório Remoto:** É uma versão do seu repositório que está hospedada em um servidor (como GitHub, GitLab ou Bitbucket). Ele permite a colaboração entre diferentes desenvolvedores. Você pode sincronizar seu repositório local com o remoto através de operações como push, pull e fetch. O repositório remoto é essencial para o trabalho colaborativo, permitindo que várias pessoas trabalhem no mesmo projeto simultaneamente.



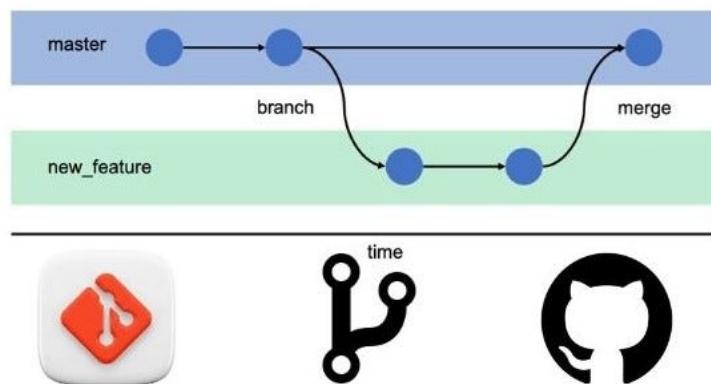
## Branches (Ramificações)

Branches são uma das funcionalidades mais poderosas do Git. Elas permitem que você crie "linhas do tempo" alternativas no seu projeto. Cada branch é uma linha separada de desenvolvimento, e você pode alternar entre elas facilmente.

- **Branch Principal (main ou master):** É a linha principal de desenvolvimento. Geralmente, contém a versão mais estável do seu projeto. Todas as outras branches geralmente derivam dessa branch principal.
- **Branches de Funcionalidade:** São branches criadas para desenvolver novas funcionalidades ou corrigir bugs sem afetar a branch principal. Isso

permite que você trabalhe em novas features ou fixes de maneira isolada, sem interromper o desenvolvimento principal.

## GIT BRANCHES



## Commits

Um commit é uma "fotografia" do seu projeto em um momento específico no tempo. Ele contém uma mensagem que descreve as alterações feitas e um identificador único (SHA-1 Hash). Os commits permitem que você acompanhe a evolução do seu projeto e reverta para versões anteriores, se necessário. Cada commit aponta para um ou mais commits anteriores, formando um histórico linear ou ramificado das alterações.

- **Histórico de Commits:** O histórico de commits é uma lista sequencial de todos os commits feitos em um repositório. Ele permite que você veja a evolução do projeto e entenda quem fez quais mudanças e por quê.

## SHA-1 Hash

Cada commit no Git é identificado por um hash SHA-1, um identificador único de 40 caracteres. Esse hash é gerado com base no conteúdo do commit, garantindo a integridade dos dados. Isso significa que, se o conteúdo de um

commit for alterado, o hash também será alterado, tornando fácil detectar alterações não autorizadas.

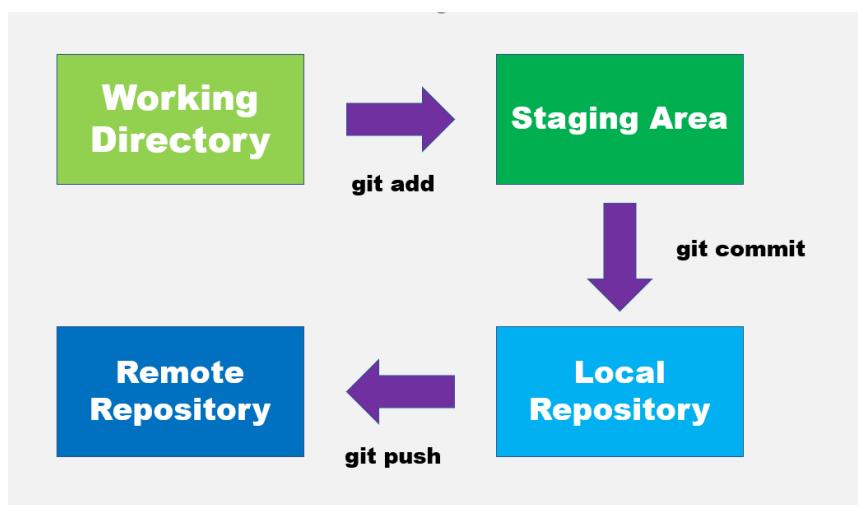
### Exemplo de SHA-1 Hash:

```
e83c5163316f89bfbd7d9ab23ca2e25604af290
```

## Árvore de Diretórios

O Git trabalha com três áreas principais:

1. **Working Directory (Diretório de Trabalho)**: Onde você faz alterações nos arquivos. É o estado atual do seu projeto no sistema de arquivos.
2. **Staging Area (Área de Preparação)**: Onde você adiciona os arquivos que deseja incluir no próximo commit. Os arquivos na staging area são preparados para serem commitados. Pense na staging area como um local onde você pode revisar e organizar suas mudanças antes de fazer um commit.
3. **Repository (Repositório)**: Onde os commits são armazenados. É o histórico completo do seu projeto. No repositório, todos os commits são armazenados de maneira segura e eficiente.



Esses conceitos formam a base do funcionamento do Git. Entender como cada componente interage com os outros é essencial para usar o Git de maneira

eficaz e eficiente. No próximo tópico, vamos explorar os comandos básicos do Git para que você possa colocar esses conceitos em prática.

---

## 4. Comandos Básicos do Git

Agora que você já entende os conceitos básicos do Git, é hora de colocar a mão na massa e aprender os comandos essenciais para trabalhar com o Git no dia a dia. Vamos explorar os comandos mais usados e como eles se encaixam nos conceitos que discutimos anteriormente.

### Inicialização de um Repositório ( `git init` )

O comando `git init` é usado para criar um novo repositório Git. Quando você executa esse comando, ele inicializa um repositório vazio no diretório atual, criando um subdiretório `.git` que contém todos os arquivos necessários do repositório.

```
git init
```

#### Exemplo:

```
mkdir meu-projeto  
cd meu-projeto  
git init
```

### Clonagem de Repositórios ( `git clone` )

O comando `git clone` cria uma cópia local de um repositório remoto. É muito usado para começar a trabalhar em um projeto existente.

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

#### Exemplo:

```
git clone <https://github.com/usuario/meu-repositorio.git>
```

## Status do Repositório ( `git status` )

O comando `git status` mostra o estado atual do diretório de trabalho e da staging area. Ele informa quais arquivos foram modificados, quais estão prontos para serem commitados e quais não estão sendo rastreados pelo Git.

```
git status
```

## Adição de Arquivos ( `git add` )

O comando `git add` adiciona mudanças do diretório de trabalho à staging area. Isso permite que você prepare suas alterações para serem commitadas. Você pode adicionar arquivos individuais, múltiplos arquivos ou todos de uma vez.

```
git add <nome-do-arquivo>
git add .
```

### Exemplo:

```
git add meu-arquivo.txt
git add .
```

## Commit de Alterações ( `git commit` )

O comando `git commit` grava as alterações da staging area no repositório. Cada commit deve ter uma mensagem descritiva que explique as alterações feitas.

```
git commit -m "Mensagem do commit"
```

### Exemplo:

```
git commit -m "Adiciona nova funcionalidade X"
```

## Visualização de Histórico ( `git log` )

O comando `git log` mostra o histórico de commits do repositório. Ele exibe uma lista de commits em ordem cronológica inversa, começando pelo commit mais recente.

```
git log
```

### Exemplo:

```
git log --oneline --graph --all
```

## Exercícios Práticos

### 1. Inicialize um Re却ritório:

- Crie um novo diretório chamado `projeto-exemplo` no seu computador.
- Navegue até esse diretório e inicialize um repositório Git com `git init`.
- Verifique o conteúdo do diretório para ver o subdiretório `.git`.

### 2. Clone um Re却ritório:

- Escolha um repositório público no GitHub (ou qualquer outra plataforma) e clone-o para o seu computador usando `git clone <url-do-repositorio>`.

### 3. Adicione e Committe Arquivos:

- Crie um arquivo chamado `hello.txt` no seu repositório.
- Adicione algum conteúdo a esse arquivo e salve.
- Use `git add hello.txt` para adicionar o arquivo à staging area.
- Faça um commit com a mensagem "Adiciona arquivo hello.txt".

### 4. Verifique o Status e o Histórico:

- Use `git status` para verificar o estado atual do seu repositório.
- Faça algumas alterações no arquivo `hello.txt` e verifique novamente o status.

- Adicione e committe as alterações com uma mensagem apropriada.
- Use `git log` para visualizar o histórico de commits do seu repositório.

## 5. Experimente os Comandos Avançados do `git log`:

- Use `git log --oneline --graph --all` para visualizar o histórico de commits em uma forma mais condensada e visual.
- 

# 5. Trabalhando com Branches

Branches (ramificações) são uma das funcionalidades mais poderosas e flexíveis do Git. Elas permitem que você desenvolva novas funcionalidades, corrija bugs ou experimente ideias sem afetar o código principal do projeto. Vamos explorar como criar, gerenciar e mesclar branches de maneira eficaz.

## Conceito de Branches

Uma branch em Git é essencialmente uma linha do tempo separada de commits. A branch principal, geralmente chamada de `main` ou `master`, é a linha de desenvolvimento principal. As branches secundárias podem ser criadas a partir de qualquer commit para isolar trabalhos específicos, como desenvolvimento de novas funcionalidades ou correção de bugs.

### Por que usar branches?

- **Isolamento:** Permite que você trabalhe em novas funcionalidades sem interferir com o código estável na branch principal.
- **Colaboração:** Facilita o trabalho colaborativo, permitindo que vários desenvolvedores trabalhem em diferentes funcionalidades simultaneamente.
- **Organização:** Ajuda a manter o histórico do projeto organizado e fácil de entender.

## Criação de Branches

Para criar uma nova branch, você usa o comando `git branch <nome-da-branch>`. Isso cria uma nova branch, mas não muda para ela automaticamente. Para alternar para a nova branch, use `git checkout <nome-da-branch>` ou combine os dois comandos com `git checkout -b <nome-da-branch>`.

#### Exemplo:

```
git branch nova-funcionalidade  
git checkout nova-funcionalidade
```

ou

```
git checkout -b nova-funcionalidade
```

## Navegação entre Branches

Você pode alternar entre branches a qualquer momento usando `git checkout <nome-da-branch>`. Quando você muda de branch, o Git atualiza o diretório de trabalho para refletir o estado da branch selecionada.

#### Exemplo:

```
git checkout main  
git checkout nova-funcionalidade
```

## Mesclagem de Branches

Mesclar branches é o processo de integrar as mudanças de uma branch em outra. O comando `git merge <nome-da-branch>` é usado para isso. Quando você faz um merge, o Git tenta combinar as alterações das duas branches.

#### Exemplo:

```
git checkout main  
git merge nova-funcionalidade
```

## Resolução de Conflitos

Às vezes, quando você tenta mesclar duas branches, podem ocorrer conflitos se as mesmas linhas de código foram alteradas em ambas as branches. O Git marca esses conflitos e permite que você os resolva manualmente.

### Passos para resolver conflitos:

1. Identifique os arquivos com conflitos (marcados com `<<<<<`, `=====` e `>>>>>`).
2. Edite os arquivos para resolver os conflitos.
3. Adicione os arquivos resolvidos à staging area (`git add <arquivo>`).
4. Complete o merge com um commit (`git commit`).

## Branches Remotas e Rastreamento

Branches remotas são branches que existem no repositório remoto. Para sincronizar suas branches locais com as remotas, você pode usar `git fetch` e `git pull`. O Git cria branches de rastreamento remoto, que são cópias locais das branches remotas.

### Exemplo:

```
git fetch origin
git checkout -b nova-funcionalidade origin/nova-funcionalidade
```

## Exercícios Práticos

### 1. Criar e Alternar Entre Branches:

- Crie uma nova branch chamada `feature-a`.
- Altere para a branch `feature-a` e faça algumas alterações em um arquivo.
- Volte para a branch `main` e verifique se suas alterações não estão lá.

### 2. Mesclar Branches:

- Na branch `feature-a`, faça um commit com suas alterações.

- Volte para a branch `main` e faça algumas alterações diferentes no mesmo arquivo.
- Tente mesclar a branch `feature-a` na `main` e resolva qualquer conflito que ocorrer.

### 3. Criar e Trabalhar com Branches Remotas:

- Crie uma nova branch chamada `feature-b` e faça um commit.
- Faça o push da branch `feature-b` para o repositório remoto.
- No repositório remoto, faça algumas alterações e comite-as.
- No seu repositório local, faça o pull das mudanças na branch `feature-b`.

### 4. Resolução de Conflitos:

- Crie duas novas branches, `conflict-1` e `conflict-2`, a partir da `main`.
  - Em `conflict-1`, edite uma linha específica em um arquivo e comite.
  - Em `conflict-2`, edite a mesma linha de maneira diferente e comite.
  - Tente mesclar `conflict-2` em `conflict-1` e resolva o conflito manualmente.
- 

## 6. Trabalho com Repositórios Remotos

Trabalhar com repositórios remotos é uma parte fundamental do uso do Git, especialmente quando você está colaborando com outras pessoas.

Repositórios remotos são versões do seu projeto que estão hospedadas em servidores na internet ou em uma rede interna. Vamos explorar como adicionar repositórios remotos, enviar (push) e receber (pull) alterações e manter seus repositórios locais e remotos sincronizados.

### Adição de Re却itórios Remotos (`git remote add`)

Para começar a trabalhar com um repositório remoto, você primeiro precisa adicioná-lo ao seu repositório local. O comando `git remote add` é usado para isso.

### Sintaxe:

```
git remote add <nome-do-remoto> <url-do-repositorio-remoto>
```

### Exemplo:

```
git remote add origin <https://github.com/usuario/meu-repositorio.git>
```

Aqui, `origin` é o nome padrão dado ao repositório remoto principal. Você pode usar qualquer nome, mas `origin` é uma convenção amplamente seguida.

## Push para Re却tórios Remotos ( `git push` )

O comando `git push` é usado para enviar suas alterações locais para um repositório remoto. Esse comando atualiza o repositório remoto com os commits que você fez localmente.

### Sintaxe:

```
git push <nome-do-remoto> <nome-da-branch>
```

### Exemplo:

```
git push origin main
```

## Pull de Re却tórios Remotos ( `git pull` )

O comando `git pull` é usado para buscar (fetch) e integrar (merge) alterações do repositório remoto para o seu repositório local. É uma combinação de dois comandos: `git fetch` e `git merge`.

### Sintaxe:

```
git pull <nome-do-remoto> <nome-da-branch>
```

### **Exemplo:**

```
git pull origin main
```

## **Fetch de Atualizações Remotas ( `git fetch` )**

O comando `git fetch` busca todas as atualizações do repositório remoto, mas não as integra automaticamente no seu repositório local. Ele atualiza suas referências remotas, permitindo que você veja as mudanças sem afetar sua branch atual.

### **Sintaxe:**

```
git fetch <nome-do-remoto>
```

### **Exemplo:**

```
git fetch origin
```

## **Reposições e Clonagem de Repositórios**

Além de adicionar, enviar e receber alterações, você também pode clonar um repositório existente para começar a trabalhar nele localmente.

### **Clonagem de Repositórios:**

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

### **Exemplo:**

```
git clone <https://github.com/usuario/meu-repositorio.git>
```

## **Gerenciamento de Múltiplos Remotos**

Você pode adicionar múltiplos repositórios remotos a um único repositório local. Isso é útil quando você está colaborando em diferentes forks do mesmo projeto ou quando está lidando com diferentes servidores remotos.

#### **Adicionar um Segundo Remoto:**

```
git remote add upstream <https://github.com/outro-usuário/outro-repositorio.git>
```

#### **Listar Remotos:**

```
git remote -v
```

## **Exercícios Práticos**

### **1. Adicionar um Re却itório Remoto:**

- Crie um novo repositório no GitHub.
- No seu repositório local, adicione esse novo repositório remoto usando `git remote add origin <url-do-repositorio>`.
- Verifique se o repositório remoto foi adicionado corretamente com `git remote -v`.

### **2. Enviar (Push) Alterações para o Re却itório Remoto:**

- Faça algumas alterações no seu repositório local e comite-as.
- Use `git push origin main` para enviar essas alterações para o repositório remoto.
- Verifique no GitHub (ou outro servidor) se as alterações foram aplicadas.

### **3. Buscar (Fetch) e Integrar (Pull) Alterações:**

- Peça a um colaborador para fazer alterações e comitar no repositório remoto.
- No seu repositório local, use `git fetch origin` para buscar as atualizações.
- Verifique as alterações usando `git log` ou `git diff`.

- Use `git pull origin main` para integrar as alterações ao seu repositório local.

#### 4. Gerenciar Múltiplos Remotos:

- Adicione um segundo repositório remoto ao seu projeto com `git remote add upstream <url-do-outro-repositorio>`.
  - Faça `fetch` das alterações desse segundo repositório com `git fetch upstream`.
  - Tente mesclar uma branch do segundo repositório no seu repositório local.
- 

## 7. Manipulação e Reescrita de Histórico

Manipular e reescrever o histórico no Git pode parecer assustador no início, mas é uma habilidade poderosa que pode ajudar a manter seu repositório limpo e compreensível. Vamos explorar as principais técnicas e comandos para manipular e reescrever o histórico de commits.

### Revertendo Commits (`git revert`)

O comando `git revert` cria um novo commit que desfaz as alterações introduzidas por um commit anterior. Isso é útil quando você deseja desfazer uma alteração sem remover o histórico.

#### Sintaxe:

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

#### Exemplo:

```
git revert e83c5163316f89bfbde7d9ab23ca2e25604af290
```

Esse comando cria um novo commit que reverte as mudanças do commit especificado, preservando o histórico original.

## Resetando Commits ( `git reset` )

O comando `git reset` pode ser usado para redefinir a cabeça do branch atual para um commit anterior, removendo commits subsequentes do histórico.

Existem três modos principais de reset:

1. **Soft**: Move a cabeça do branch para o commit especificado, mas mantém as alterações na staging area.

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

2. **Mixed (padrão)**: Move a cabeça do branch para o commit especificado e limpa a staging area, mantendo as alterações no diretório de trabalho.

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

3. **Hard**: Move a cabeça do branch para o commit especificado e remove todas as alterações na staging area e no diretório de trabalho.

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

### Exemplo:

```
git reset --hard e83c5163316f89bfbde7d9ab23ca2e25604af290
```

## Rebasing ( `git rebase` )

O comando `git rebase` é usado para reaplicar commits de uma branch sobre outra. Isso pode ser útil para manter um histórico de commits mais linear e limpo.

Existem dois tipos principais de rebase:

1. **Rebase Interativo**: Permite reordenar, editar, combinar ou remover commits.

```
git rebase -i <ponto-de-partida>
```

**2. Rebase Normal:** Aplica todos os commits da branch atual em cima de outra branch.

```
git rebase <branch-base>
```

### Exemplo de Rebase Interativo:

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

Esse comando abre um editor onde você pode escolher como manipular os últimos três commits.

## Stashing (`git stash`)

O comando `git stash` é usado para salvar temporariamente mudanças não commitadas no diretório de trabalho, permitindo que você trabalhe em outra coisa e depois volte ao estado salvo.

### Sintaxe:

```
git stash
```

Para aplicar o stash mais recente:

```
git stash apply
```

Para listar todos os stashes:

```
git stash list
```

### Exemplo:

```
git stash
# Faz algumas outras operações
git stash apply
```

## Exercícios Práticos

## **1. Reverter um Commit:**

- Crie um commit no seu repositório local.
- Use `git revert <hash-do-commit>` para criar um novo commit que desfaz as alterações do commit anterior.
- Verifique o histórico de commits para confirmar que o revert foi aplicado.

## **2. Resetar Commits:**

- Faça vários commits no seu repositório local.
- Use `git reset --soft <hash-do-commit>` para redefinir a cabeça do branch para um commit anterior, mantendo as alterações na staging area.
- Experimente também os modos `-mixed` e `-hard` e observe as diferenças.

## **3. Rebase Interativo:**

- Crie uma sequência de commits no seu repositório local.
- Use `git rebase -i HEAD~3` para abrir o rebase interativo.
- Reordene, edite ou combine alguns commits e complete o rebase.
- Verifique o histórico para ver como ele foi alterado.

## **4. Usar Stash:**

- Faça algumas alterações no seu diretório de trabalho sem comitar.
- Use `git stash` para salvar temporariamente essas mudanças.
- Faça outras alterações e comite-as.
- Use `git stash apply` para restaurar o stash anterior.
- Verifique se todas as mudanças foram restauradas corretamente.

## **5. Combinar Stash e Rebase:**

- Crie várias alterações e stashes.
- Use `git stash list` para ver todos os stashes salvos.
- Aplique um stash específico e faça um rebase interativo em seguida.

# 8. Trabalho Colaborativo

Colaborar eficientemente é uma das grandes vantagens de usar o Git. Em projetos de software, é comum trabalhar em equipes, contribuir para projetos open-source e gerenciar revisões de código. Vamos explorar como usar o Git para facilitar o trabalho colaborativo, incluindo forks, pull requests, revisão de código e estratégias de branching.

## Forks e Pull Requests

Forks e pull requests são conceitos fundamentais para colaborar em projetos de código aberto.

**Fork:** Quando você "forka" um repositório, você cria uma cópia do repositório original em sua própria conta no GitHub ou outra plataforma. Isso permite que você faça alterações no seu fork sem afetar o repositório original.

**Pull Request (PR):** Um pull request é uma solicitação para mesclar suas alterações (do seu fork ou de uma branch) de volta ao repositório original. É uma forma de propor mudanças e iniciar discussões sobre elas.

### Fluxo de Trabalho Típico com Forks e PRs:

- 1. Fork o Repositório:** Faça um fork do repositório original para a sua conta.
- 2. Clone o Fork:** Clone o repositório forkado para o seu ambiente local.

```
git clone <https://github.com/usuario/seu-fork.git>
```

- 3. Crie uma Branch:** Crie uma nova branch para suas alterações.

```
git checkout -b minha-feature
```

- 4. Faça Suas Alterações e Commit:** Faça as alterações necessárias e commit.

```
git add .  
git commit -m "Implementa minha feature"
```

- 5. Push para o Fork:** Envie suas alterações para o repositório forkado.

```
git push origin minha-feature
```

**6. Abra um Pull Request:** No GitHub (ou outra plataforma), abra um pull request para mesclar sua branch no repositório original.



## Revisão de Código

A revisão de código é um processo onde outros desenvolvedores revisam seu código antes de ser mesclado na branch principal. Isso ajuda a garantir a qualidade do código e a encontrar bugs ou problemas antes de integrar as mudanças.

### Práticas Recomendadas para Revisão de Código:

- **Clareza:** Escreva mensagens de commit claras e detalhadas.
- **Pequenos Commits:** Mantenha os commits pequenos e focados em uma única tarefa ou correção.
- **Comentários:** Comente seu código para explicar partes complexas ou decisões de design.
- **Testes:** Inclua testes para suas alterações, sempre que possível.

## Estratégias de Branching

Estratégias de branching ajudam a gerenciar o fluxo de trabalho de desenvolvimento, especialmente em equipes grandes. Duas estratégias populares são:

1. **Git Flow**: Uma estratégia de branching robusta que utiliza várias branches permanentes (como `main`, `develop`, `feature`, `release`, `hotfix`). Cada tipo de branch tem um propósito específico e regras de mesclagem.

- `main` : Contém a versão de produção.
- `develop` : Contém o último desenvolvimento entregue.
- `feature` : Usada para desenvolver novas funcionalidades.
- `release` : Preparação para a nova versão de produção.
- `hotfix` : Correções urgentes em produção.

2. **GitHub Flow**: Uma estratégia mais simples e linear, adequada para desenvolvimento contínuo e deploy frequente. Utiliza uma única branch de produção (`main`) e branches de feature.

- `main` : Contém a versão de produção.
- `feature` : Branches curtas criadas para novas funcionalidades ou correções, que são mescladas diretamente na `main`.

## Exercícios Práticos

### 1. Criar e Gerenciar um Fork:

- Faça um fork de um repositório público no GitHub.
- Clone o repositório forkado no seu ambiente local.
- Crie uma branch nova, faça algumas alterações e comite-as.
- Faça o push das suas alterações para o seu fork e abra um pull request para o repositório original.

### 2. Participar de Revisões de Código:

- Revise o código em um pull request de um colega.
- Deixe comentários sobre partes que você acha que podem ser melhoradas ou que precisam de correções.

- Faça sugestões de melhoria ou correção e aprove ou rejeite o pull request baseado na qualidade do código.

### 3. Implementar Git Flow:

- Em um projeto local, implemente a estratégia Git Flow.
- Crie branches `develop`, `feature`, `release` e `hotfix` conforme necessário.
- Siga o fluxo de trabalho Git Flow, mesclando mudanças das branches `feature` para `develop` e depois para `main`.

### 4. Implementar GitHub Flow:

- Em outro projeto, implemente a estratégia GitHub Flow.
- Crie uma branch `feature` para uma nova funcionalidade, faça alterações e comite-as.
- Faça o push para o repositório remoto e abra um pull request para a branch `main`.
- Mescle o pull request e verifique se as mudanças foram aplicadas corretamente.

### 5. Simulação de Colaboração:

- Em colaboração com um colega, faça um fork de um repositório e trabalhem em diferentes branches de feature.
  - Realizem pull requests entre si e pratiquem revisões de código.
  - Usem estratégias de branching para organizar e integrar suas alterações de forma eficiente.
- 

## 9. Boas Práticas e Convenções

Manter um repositório Git bem organizado e fácil de entender é essencial para qualquer equipe de desenvolvimento. Seguir boas práticas e convenções ajuda a garantir que o histórico de commits seja limpo, as branches sejam gerenciáveis e as colaborações sejam eficazes. Vamos explorar algumas das melhores práticas e convenções que você deve seguir ao usar Git.

## Mensagens de Commit Claras e Descritivas

Mensagens de commit são essenciais para entender o que foi alterado e por que. Boas mensagens de commit facilitam a navegação pelo histórico do projeto e a compreensão das mudanças feitas.

### Estrutura de uma Boa Mensagem de Commit:

1. **Linha de Assunto:** Uma breve descrição da mudança (máximo de 50 caracteres).
2. **Linha em Branco:** Separação visual.
3. **Corpo:** Descrição detalhada da mudança, explicando o que foi alterado e por que (máximo de 72 caracteres por linha).

### Exemplo:

```
Corrige bug na função de login
```

A função de login estava falhando ao processar senhas contendo caracteres especiais.

Foi adicionada uma verificação para tratar corretamente esses casos.

## Padrões de Branches

Seguir uma convenção clara para nomear e gerenciar branches facilita a colaboração e o gerenciamento do projeto.

### Convencional:

- **main:** Branch principal, contendo a versão estável do código.
- **develop:** Branch de desenvolvimento, onde a integração de novas funcionalidades ocorre antes de serem lançadas.
- **feature/nome-da-feature:** Branches de funcionalidades, derivadas de `develop`.
- **hotfix/nome-do-hotfix:** Branches de correção urgente, derivadas de `main`.
- **release/nome-da-release:** Branches de preparação para lançamento, derivadas de `develop`.

### **Exemplo:**

```
feature/adicionar-login-social  
hotfix/corrigir-bug-login  
release/v1.2.0
```

## **Versionamento Semântico**

Versionamento semântico é uma convenção para atribuir números de versão que transmitem significado sobre as mudanças feitas no software.

### **Formato:**

```
MAJOR.MINOR.PATCH
```

- **MAJOR**: Incrementado para mudanças incompatíveis com versões anteriores.
- **MINOR**: Incrementado para novas funcionalidades compatíveis com versões anteriores.
- **PATCH**: Incrementado para correções de bugs compatíveis com versões anteriores.

### **Exemplo:**

```
1.0.0 -> 2.0.0: Mudança incompatível  
1.0.0 -> 1.1.0: Nova funcionalidade  
1.0.0 -> 1.0.1: Correção de bug
```

## **Revisão de Código**

A revisão de código é um processo fundamental para garantir a qualidade e a segurança do código antes de ele ser mesclado na branch principal.

### **Práticas Recomendadas:**

- **Faça Revisões Frequentes**: Revisar código regularmente evita que grandes volumes de mudanças se acumulem.

- **Seja Construtivo:** Faça comentários úteis e construtivos, focando na melhoria do código.
- **Teste o Código:** Sempre teste o código revisado para garantir que ele funcione conforme o esperado.

## Gerenciamento de Conflitos

Conflitos de merge são inevitáveis em qualquer projeto colaborativo. Saber como resolvê-los eficientemente é crucial.

### Passos para Resolver Conflitos:

1. **Identificar Conflitos:** Use `git status` para ver quais arquivos estão em conflito.
  2. **Editar Arquivos:** Resolva manualmente os conflitos nos arquivos marcados com `<<<<<`, `=====` e `>>>>>`.
  3. **Adicionar Arquivos Resolvidos:** Adicione os arquivos resolvidos à staging area com `git add <arquivo>`.
  4. **Finalizar Merge:** Complete o merge com um commit (`git commit`).
- 

## 10. Recursos Adicionais e Referências

Para se tornar um mestre em Git, é importante continuar aprendendo e se atualizando sobre as melhores práticas e novos recursos. Há uma vasta quantidade de recursos disponíveis que podem ajudá-lo a aprofundar seus conhecimentos e resolver problemas específicos. Aqui estão alguns dos melhores recursos e referências para você consultar.

### Documentação Oficial do Git

A documentação oficial do Git é a fonte mais confiável e abrangente de informações sobre Git. Ela cobre todos os aspectos do uso do Git, desde os comandos básicos até as funcionalidades mais avançadas.

### Links Úteis:

- [Documentação Oficial do Git](#)
- [Pro Git Book](#): Um livro completo e gratuito sobre Git, escrito por Scott Chacon e Ben Straub.

## Livros Recomendados

Ler livros sobre Git pode fornecer uma compreensão mais profunda e estruturada dos conceitos e práticas.

1. **"Aprendendo Git"** por O'Reilly
  - [Adquira aqui.](#)
2. **"Controlando Versões com Git e GitHub"** por Casa do Código
  - [Adquira aqui.](#)

## Tutoriais e Cursos Online

Cursos online e tutoriais em vídeo podem ser extremamente úteis para aprender Git de maneira prática e visual.

1. **Udemy**
  - [Git Complete: The Definitive, Step-by-Step Guide to Git](#)
  - Um curso completo que cobre todos os aspectos do Git, desde a configuração inicial até as funcionalidades avançadas.
2. **Coursera**
  - [Version Control with Git](#)
  - Oferecido pela Atlassian, este curso abrange os fundamentos do controle de versão com Git e práticas colaborativas.
3. **YouTube**
  - Canais como [The Net Ninja](#) e [Traversy Media](#) oferecem tutoriais gratuitos e de alta qualidade sobre Git.

## Ferramentas e Plugins

Ferramentas adicionais e plugins podem melhorar sua experiência com Git, tornando-o mais produtivo e eficiente.

### 1. **GitLens (para Visual Studio Code)**

- Extensão que melhora a visualização do histórico de commits, autores e alterações de código diretamente no VS Code.
- [GitLens - Git supercharged](#)

### 2. **Oh My Zsh**

- Um framework para a configuração do Zsh, que inclui plugins Git para melhorar a produtividade na linha de comando.
- [Oh My Zsh](#)

### 3. **Git Extensions**

- Uma ferramenta GUI poderosa para Git, que facilita o uso de comandos avançados e visualizações de histórico.
- [Git Extensions](#)

## Comunidades e Fóruns de Discussão

Participar de comunidades e fóruns de discussão pode ajudá-lo a resolver problemas, compartilhar conhecimento e aprender com outros desenvolvedores.

### 1. **Stack Overflow**

- [Git Tag](#)
- Um dos melhores lugares para fazer perguntas e encontrar respostas para problemas específicos relacionados ao Git.

### 2. **Reddit**

- [r/git](#)
- Uma comunidade ativa onde você pode discutir práticas e soluções com outros usuários de Git.

### 3. **GitHub Community**

- [GitHub Community Forum](#)

- Um fórum oficial para usuários do GitHub, onde você pode encontrar dicas, truques e resolver problemas com a ajuda da comunidade.