ENIAC

APOSTILA

GIT

SUMÁRIO

3 ..... Introdução ao Git e controle de versão

6 ..... Git VS Github

7 ..... Estrutura do Git

8 ..... Fluxo de trabalho

9 ..... Principais comandos

10 ..... Trabalhando com ramificações

11 ..... Pull Requests

11 ..... Padrões de desenvolvimento

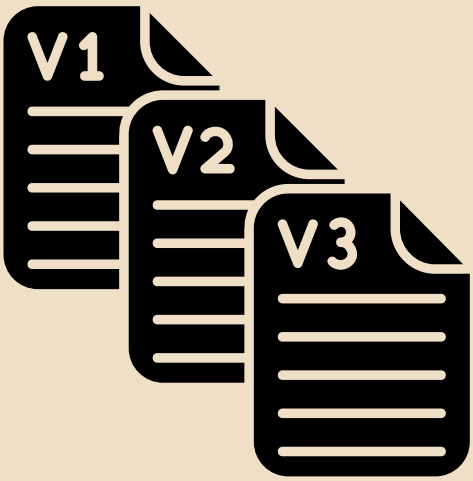
15 ..... Guia de Referência Rápida - Git

22 ..... Dicas rápidas e Aliases Úteis

INTRODUÇÃO AO GIT E CONTROLE DE VERSÃO

Por que usar o GIT?

No desenvolvimento de qualquer projeto, seja um código-fonte, um documento ou um site, o **gerenciamento de alterações** é um desafio constante. É comum recorrer a métodos manuais, como salvar múltiplas cópias de um arquivo.



Exemplo:

projeto_v1.cpp;
projeto_v2_final.cpp;
projeto_v2_final_agora_sim.cpp;

No entanto, essa abordagem, além de ineficiente, é suscetível a erros, como a perda de alterações importantes ou a dificuldade em identificar qual é a versão mais recente e estável. Para solucionar esse problema de forma profissional, utilizamos um **Sistema de Controle de Versão** (VCS - Version Control System).

O **Git** é o VCS mais utilizado no mundo atualmente. Ele é uma ferramenta que **gerencia e armazena o histórico de modificações de um projeto**. A cada alteração significativa, o Git salva um "snapshot" (um registro instantâneo) do estado de todos os arquivos, criando um histórico detalhado e permitindo a navegação entre as diferentes versões.

ORIGEM DO GIT

O Git foi criado em 2005 por **Linus Torvalds**, o mesmo criador do kernel do **Linux**. O desenvolvimento do Linux envolvia (e ainda envolve) a **colaboração de milhares de desenvolvedores ao redor do mundo**, o que representava um desafio logístico e técnico gigantesco.

A ferramenta de Version Control System (VCS) que utilizavam na época, o **BitKeeper**, não atendia mais às necessidades do projeto por razões cruciais que iam além da tecnologia. Essencialmente, o **uso de uma ferramenta restritiva** e proprietária ia contra o próprio ethos do software livre que o Linux representava. Esse conflito filosófico se materializou quando a licença de uso gratuito foi revogada pela empresa, um incidente que não apenas deixou o projeto sem sua ferramenta principal, mas também expôs a grande vulnerabilidade de um projeto de código aberto depender de uma tecnologia de código-fechado.

BitKeeper
Scalable Version Control

Linus então projetou o Git com **três objetivos principais** em mente:

1 - **Velocidade**: Ser extremamente rápido para lidar com projetos de grande escala.

2 - **Modelo Distribuído**: Diferente de sistemas centralizados, no Git cada desenvolvedor possui uma cópia completa do repositório, incluindo todo o seu histórico. Isso permite trabalho offline e maior redundância.

3 - **Integridade de Dados**: Garantir que o histórico do projeto não possa ser corrompido, usando mecanismos criptográficos para registrar cada alteração.

Embora tenha sido concebido para o Linux, sua **eficiência e flexibilidade** o tornaram o padrão de fato para o desenvolvimento de software em toda a indústria.

A partir disso, é possível notar a **importância de adotar o Git em projetos**, desde os acadêmicos até os profissionais, tendo em vista que ele oferece vantagens estruturais significativas. **Os conceitos a seguir são a base do seu poder.**

- **Histórico e Rastreabilidade (Controle de Versão)**: O Git mantém um log detalhado de todas as alterações. Para cada modificação salva (um processo chamado commit), ele armazena o que foi alterado, quem fez a alteração e quando. Isso funciona, na prática, como uma "máquina do tempo" para o seu código, permitindo reverter o projeto para qualquer ponto estável do passado com total segurança.

- **Trabalho em Equipe e Colaboração (Desenvolvimento Paralelo)**: O Git foi projetado para o trabalho em equipe. Ele permite que múltiplos desenvolvedores trabalhem simultaneamente na mesma base de código sem que um sobrescreva o trabalho do outro. Através de um processo chamado merge (mesclagem), o Git auxilia na combinação das diferentes linhas de trabalho, destacando e ajudando a resolver eventuais conflitos.

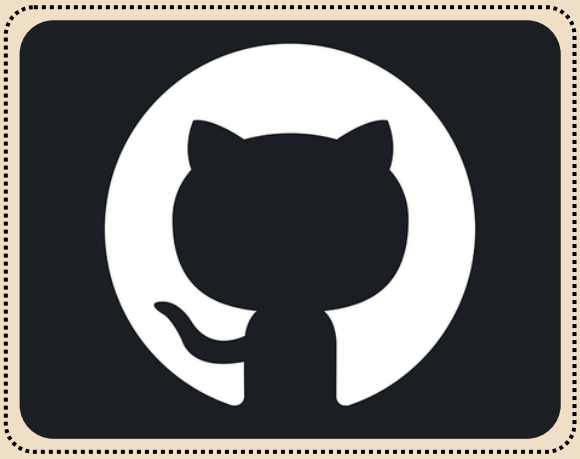
- **Ramificações para Desenvolvimento Isolado (Branches)**: Uma das funcionalidades mais poderosas do Git é a capacidade de criar branches (ramificações). Uma branch é essencialmente uma linha de desenvolvimento independente. Você pode criar uma nova branch para trabalhar em uma funcionalidade específica, corrigir um bug ou experimentar uma nova ideia, tudo isso em isolamento, sem afetar a linha principal do projeto (comumente chamada de main ou master). Uma vez que o trabalho na branch esteja concluído e testado, ele pode ser reintegrado à branch principal.

GIT VS GITHUB

A **diferença** entre **git** e **github** é uma das dúvidas mais comuns para iniciantes, por isso é crucial esclarecer a distinção desde o início. Embora os nomes sejam parecidos e trabalhem juntos, eles são coisas fundamentalmente diferentes.

Git é a ferramenta, é o Sistema de **Controle de Versão (VCS) propriamente dito**. É o software que você instala em seu computador para rastrear as alterações no seu código, gerenciar o histórico de versões e trabalhar com branches. O Git funciona localmente, na sua máquina, e não precisa de internet para operar no seu projeto.

GitHub é a plataforma, é um **serviço de hospedagem na nuvem para repositórios Git**. Ele utiliza o Git como sua tecnologia central e adiciona uma interface web e uma série de funcionalidades colaborativas e sociais sobre ele. Pense no GitHub como uma rede social para desenvolvedores.



O **GitHub** permite que você **pegue seu repositório Git local e o publique na internet**, criando um repositório remoto. Isso serve como um backup seguro e, mais importante, como um ponto central para o trabalho em equipe.

Na plataforma do **GitHub**, você pode:

- Visualizar projetos de outros desenvolvedores.
- Sugerir alterações no código de outras pessoas (através de Pull Requests).
- Discutir problemas e novas funcionalidades (Issues).
- Marcar seus repositórios favoritos com uma estrela (Star), de forma semelhante a uma "curtida".

Por fim, é importante saber que o GitHub não é o único. Existem outras plataformas populares que oferecem serviços de hospedagem para repositórios Git, como o **GitLab, Azure/AWS/GCP e o Bitbucket.** Todas elas usam a mesma ferramenta por baixo dos panos: o Git.

ESTRUTURA DO GIT

Para **rastrear as alterações de forma eficiente**, o Git divide seu projeto em **três "árvores"** ou ambientes principais que existem localmente na sua máquina. Entender o fluxo de trabalho entre eles é a chave para dominar o Git.

1. Working Directory (Área de Trabalho)

Este é o ambiente mais simples: é a pasta do seu projeto no seu computador. São os arquivos que você pode ver, editar, criar e deletar diretamente com seu editor de código ou gerenciador de arquivos. É a sua "mesa de trabalho", onde a versão atual dos arquivos está disponível para você modificar.

Estado: Contém arquivos modificados, novos ou não rastreados.

Analogia: Sua mesa de trabalho onde você espalha os documentos e os edita livremente.

2. Staging Area (Área de Preparação ou "Index")

Este é um ambiente intermediário e um dos conceitos mais poderosos do Git. A Staging Area é como uma "área de preparação". Antes de salvar permanentemente um conjunto de alterações (fazer um commit), você primeiro precisa adicioná-las aqui. Isso permite que você escolha exatamente quais modificações farão parte do próximo "snapshot" do projeto, ignorando outras que ainda estão em andamento.

Comando chave: git add <arquivo> move as alterações do Working Directory para a Staging Area.

Analogia: Uma caixa que você está preparando para envio. Você seleciona cuidadosamente os documentos finalizados da sua mesa (Working Directory) e os coloca na caixa (Staging Area), prontos para serem despachados.

3. Local Repository (Repositório Local - a pasta .git)

Este é o destino final e seguro das suas alterações. O repositório local é o "banco de dados" do Git, onde todo o histórico de commits (os snapshots salvos) é armazenado permanentemente. Fica contido em uma subpasta oculta chamada .git dentro do seu projeto. Uma vez que uma alteração é "commitada", ela fica registrada de forma segura no histórico.

Comando chave: git commit pega tudo o que está na Staging Area e salva como um novo snapshot permanente no Local Repository.

Analogia: O arquivo permanente ou o cofre da sua empresa. Depois de despachar a caixa (Staging Area), seu conteúdo é arquivado com data, carimbo e uma descrição, ficando guardado para sempre no histórico.

O FLUXO DE TRABALHO VISUALIZADO

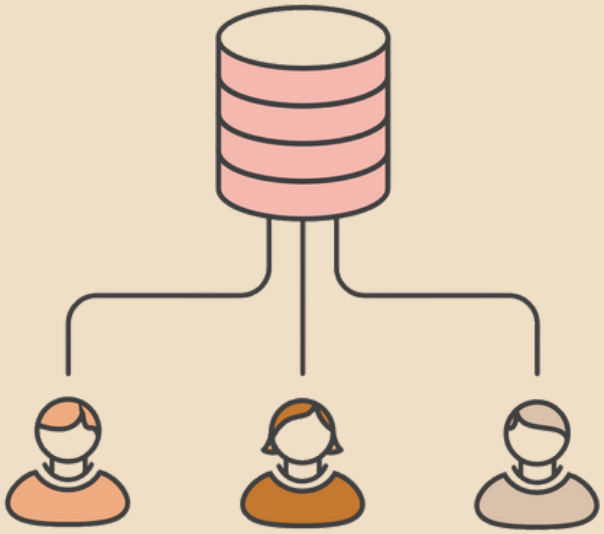
O **fluxo básico de trabalho** para salvar uma alteração no Git segue esta jornada:

Working Directory → Staging Area → Local Repository

E os comandos que realizam essa **transição** são:

[Seus Arquivos] --- git add ---› [Arquivos Preparados] --- git commit --- › [Histórico Permanente]

Depois que as alterações estão seguras no seu Repositório Local, você pode então sincronizá-las com um Repositório Remoto (como o GitHub) usando os comandos **git push** (para enviar) e **git pull** (para receber).



PRINCIPAIS COMANDOS

Agora que entendemos a estrutura do Git, vamos conhecer os **comandos** que nos permitem interagir com ele. Estes são os comandos que você usará no dia a dia para gerenciar seus projetos.

Configuração inicial

Antes de mais nada, você precisa se apresentar ao Git. Isso é feito apenas uma vez por computador.

- **git config --global user.name "Seu Nome":** Define o seu nome, que será associado a cada *commit* que você fizer.

- **git config --global user.email "seuemail@exemplo.com":** Define o seu e-mail, que também fica registrado no histórico.

Iniciar o projeto

Existem duas maneiras de começar a usar o Git em um projeto.

- **git init:** Transforma uma pasta existente em um repositório Git. Basta navegar até a pasta do seu projeto no terminal e executar este comando. Ele cria a subpasta oculta .git, que armazena todo o histórico.

- **git clone [URL]** Cria uma cópia local de um repositório remoto que já existe (por exemplo, um projeto do GitHub). Este comando baixa todo o projeto e seu histórico para a sua máquina.

Depois de salvar as alterações, você pode querer **visualizá-las**, use o comando:

- git log: Mostra o histórico de todos os commits feitos no projeto, do mais recente ao mais antigo. Cada entrada no log mostra o autor, a data e a mensagem do commit.

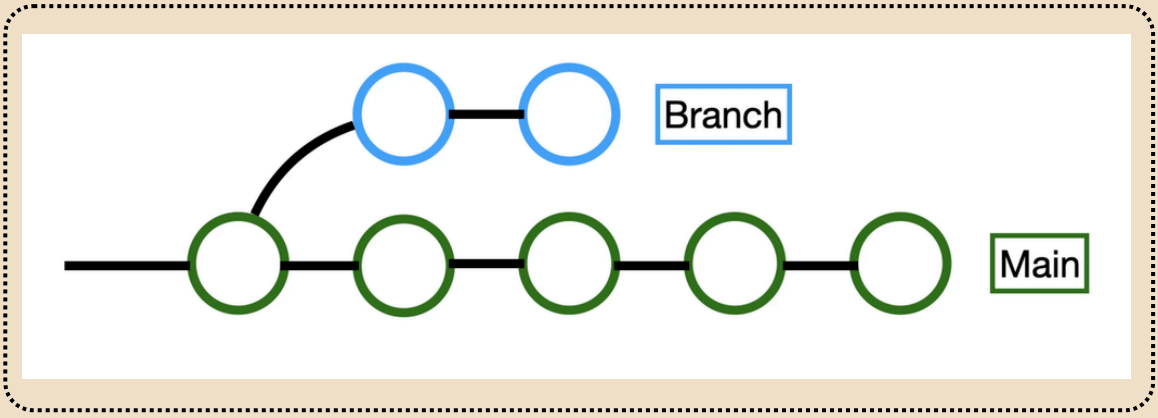
TRABALHANDO COM RAMIFICAÇÕES

Branches permitem trabalhar em **diferentes versões** do projeto simultaneamente.

- **git branch**: Lista, cria ou deleta branches. git branch novo-nome cria uma nova branch.
- **git checkout [branch] ou git switch [branch]**: Muda para a branch especificada.
- **git merge [branch]**: Une o histórico da branch especificada à sua branch atual, combinando as alterações.

Sincronizando com Repositórios Remotos

- **git remote add <nome> <URL>**: Conecta seu repositório local a um remoto. O nome padrão é origin.
- **git fetch**: Baixa as novidades do remoto, mas não as integra ao seu trabalho local.
- **git pull**: Baixa as novidades do remoto e tenta mesclá-las (merge) automaticamente. É um fetch + merge.
- **git push**: Envia seus commits locais para o repositório remoto.



PULL REQUESTS (PRS) - COLABORAÇÃO EM EQUIPE

Pull Request não é um comando do Git, mas uma **funcionalidade** de plataformas como **GitHub e GitLab**. O Pull Request é um pedido formal para que suas alterações sejam revisadas e integradas à branch principal do projeto.

Como funciona:

1 - Você trabalha em uma branch separada;
2 - Faz commits das suas alterações;
3 - Envia a branch para o repositório remoto com git push;
4 - Abre um Pull Request na plataforma (GitHub, por exemplo);
5 - Outros desenvolvedores revisam seu código;
6 - Após aprovação, as alterações são integradas (merged) à branch principal.

Vantagens:

- Permite revisão de código
- Facilita discussões sobre as alterações
- Mantém a qualidade do código
- Documenta decisões técnicas

PADRÕES DE DESENVOLVIMENTO: COMMITS E BRANCHES

1. Padrão de Commits (Conventional Commits)

1.1 Estrutura Básica

Um commit bem estruturado segue o formato:

```
<tipo>(<escopo>): <descrição>
[corpo opcional]
[rodapé opcional]
```

1.2 Tipos de Commit

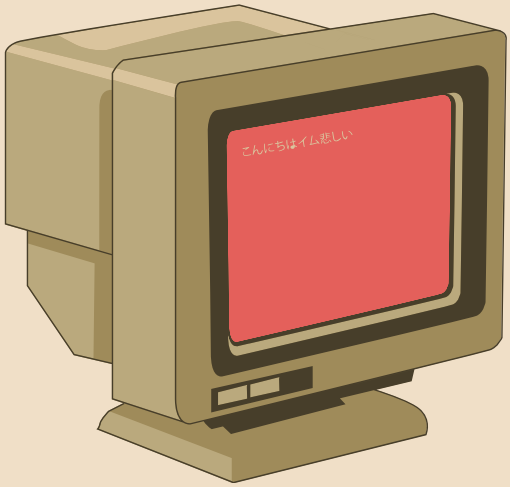
- **feat**: Nova funcionalidade visual que adiciona algo novo.
- **fix**: Correção de bug.
- **docs**: Mudanças na documentação.
- **style**: Mudanças de formatação de **código** (indentação, espaços, ponto e vírgula) ou de aparência visual (CSS, cores, espaçamentos).
- **refactor**: Reestruturação de código que muda a lógica interna.
- **test**: Adição ou correção de testes.
- **chore**: Tarefas de manutenção, configurações, dependências.

1.3 Exemplos Práticos

- feat(autenticacao): adiciona login com Google;
- fix(carrinho): corrige cálculo de frete para CEPs especiais;
- docs(readme): atualiza instruções de instalação;
- refactor(api): melhora estrutura dos controllers;
- test(usuario): adiciona testes unitários para cadastro.

1.4 Boas Práticas

- Use verbos no imperativo ("adiciona" ao invés de "adicionado")
- Mantenha a primeira linha com no máximo 50-72 caracteres
- Seja específico e descritivo
- Um commit deve representar uma unidade lógica de mudança
- Commits pequenos e frequentes são preferíveis a commits grandes



2. Padrão de Branches (Git Flow Simplificado)

2.1 Branches Principais

main/master
- Código em produção
- Sempre estável e deployável
- Protegida contra commits diretos

develop
- Branch de integração
- Contém as últimas funcionalidades desenvolvidas
- Base para criar novas features

2.2 Branches de Suporte

feature/*
- Para desenvolvimento de novas funcionalidades
- Criada a partir de: develop
- Merge de volta para: develop
- Nomenclatura: feature/nome-da-funcionalidade

```
git checkout develop
git checkout -b feature/sistema-pagamento
```

bugfix/*
- Para correção de bugs no ambiente de desenvolvimento
- Criada a partir de: develop
- Merge de volta para: develop
- Nomenclatura: bugfix/descricao-do-bug

```
git checkout develop
git checkout -b bugfix/erro-validacao-email
```

hotfix/*
- Para correções urgentes em produção
- Criada a partir de: main
- Merge de volta para: main e develop
- Nomenclatura: hotfix/descricao-do-problema

```
git checkout main
git checkout -b hotfix/falha-pagamento-critico
```

2.3 Nomenclatura de Branches

Regras gerais:

- Use letras minúsculas;
- Separe palavras com hífen (-);
- Seja descritivo mas conciso;
- Inclua o tipo de branch como prefixo.

Exemplos:

feature/autenticacao-dois-fatores
feature/relatorio-vendas
bugfix/corrige-layout-mobile
hotfix/vulnerabilidade-sql-injection

2.4 Boas Práticas

- Mantenha branches de vida curta (máximo 1-2 semanas);
- Delete branches após o merge;
- Faça rebase regularmente para evitar conflitos grandes;
- Uma branch deve ter um propósito claro e único;
- Use Pull Requests para revisão de código antes do merge.

2.5 Leia e aprenda mais

- Conventional Commits
- GitHub - iuricode/padroes-de-commits
- Conventional Commits Pattern. O que é, para que serve e como... | by Victor Ribeiro | LinkApi Solutions | Medium
- **Como fazer um COMMIT padronizado**

Glossário

- Repositório: Pasta do projeto gerenciada pelo Git
- Commit: Um ponto salvo no histórico do projeto
- Branch: Uma linha independente de desenvolvimento
- Merge: Unir alterações de diferentes branches
- Remote: Repositório hospedado online
- Clone: Cópia de um repositório remoto
- Pull Request (PR): Solicitação de revisão e integração de código

GUIA DE REFERÊNCIA RÁPIDA - GIT

Configuração Inicial

```
# Configurar nome de usuário
git config --global user.name "Seu Nome"

# Configurar email
git config --global user.email "seu@email.com"

# Ver todas as configurações
git config --list

# Configurar editor padrão
git config --global core.editor "code --wait"
```

Criar e Clonar Repositórios

```
# Inicializar repositório local
git init

# Clonar repositório remoto
git clone <url>

# Clonar para pasta específica
git clone <url> <nome-da-pasta>
```

Comandos Básicos

```
# Ver status dos arquivos
git status

# Adicionar arquivo específico
git add <arquivo>

# Adicionar todos os arquivos modificados
git add .

# Adicionar arquivos por extensão
git add *.js

# Remover arquivo do stage (antes do commit)
git reset <arquivo>
```

```
# Fazer commit
git commit -m "mensagem do commit"

# Adicionar e commitar em um comando
git commit -am "mensagem"

# Modificar último commit (mensagem ou arquivos)
git commit --amend
```

Branches (Ramificações)

```
# Listar todas as branches
git branch

# Listar branches remotas
git branch -r

# Criar nova branch
git branch <nome-branch>

# Mudar para uma branch
git checkout <nome-branch>

# Criar e mudar para nova branch
git checkout -b <nome-branch>

# Renomear branch atual
git branch -m <novo-nome>

# Deletar branch local
git branch -d <nome-branch>

# Forçar exclusão de branch
git branch -D <nome-branch>

# Deletar branch remota
git push origin --delete <nome-branch>
```

GUIA DE REFERÊNCIA RÁPIDA - GIT

Atualizar e Mesclar

```
# Buscar mudanças do remoto (sem aplicar)
git fetch

# Buscar e aplicar mudanças do remoto
git pull

# Fazer merge de uma branch na atual
git merge <nome-branch>

# Fazer rebase
git rebase <nome-branch>

# Cancelar merge em conflito
git merge --abort

# Cancelar rebase
git rebase --abort
```

Enviar para Repositório Remoto

```
# Enviar commits para o remoto
git push

# Enviar branch específica
git push origin <nome-branch>

# Enviar e criar branch no remoto
git push -u origin <nome-branch>

# Forçar push (cuidado!)
git push --force
```

Histórico e Logs

```
# Ver histórico de commits
git log
```

```
# Ver log resumido (uma linha por commit)
git log --oneline

# Ver log com gráfico de branches
git log --graph --oneline --all

# Ver últimos N commits
git log -n 5

# Ver commits de um autor
git log --author="Nome"

# Ver mudanças de um arquivo
git log -p <arquivo>

# Ver quem modificou cada linha
git blame <arquivo>
```

Desfazer Mudanças

```
# Descartar mudanças em arquivo (antes do add)
git checkout -- <arquivo>

# Descartar todas as mudanças locais
git checkout .

# Remover arquivo do stage
git reset HEAD <arquivo>

# Desfazer último commit (mantém mudanças)
git reset --soft HEAD~1

# Desfazer último commit (descarta mudanças)
git reset --hard HEAD~1

# Reverter commit específico (cria novo commit)
git revert <hash-do-commit>

# Voltar para um commit específico
git reset --hard <hash-do-commit>
```

Tags

```
# Listar todas as tags
git tag

# Criar tag
git tag <nome-tag>

# Criar tag anotada
git tag -a v1.0 -m "versão 1.0"

# Enviar tag para remoto
git push origin <nome-tag>

# Enviar todas as tags
git push --tags

# Deletar tag local
git tag -d <nome-tag>

# Deletar tag remota
git push origin --delete <nome-tag>
```

Stash (Guardar Temporariamente)

```
# Guardar mudanças temporariamente
git stash

# Guardar com mensagem descritiva
git stash save "mensagem"

# Listar todos os stashes
git stash list

# Aplicar último stash
git stash apply

# Aplicar e remover último stash
git stash pop
```

```
# Aplicar stash específico
git stash apply stash@{n}

# Remover último stash
git stash drop

# Limpar todos os stashes
git stash clear
```

Remotos

```
# Ver remotos configurados
git remote -v

# Adicionar remoto
git remote add origin <url>

# Alterar URL do remoto
git remote set-url origin <nova-url>

# Remover remoto
git remote remove <nome>

# Renomear remoto
git remote rename <nome-antigo> <nome-novo>
```

Pesquisa e Diferenças

```
# Ver diferenças não commitadas
git diff

# Ver diferenças no stage
git diff --staged

# Ver diferenças entre branches
git diff <branch1> <branch2>

# Pesquisar no código
git grep "texto"

# Pesquisar nos commits
git log -S "texto"
```

GUIA DE REFERÊNCIA RÁPIDA - GIT

Limpeza

```
# Remover arquivos não rastreados
git clean -f

# Remover arquivos e diretórios não rastreados
git clean -fd

# Ver o que será removido (dry run)
git clean -n

# Otimizar repositório local
git gc
```

Comandos de Emergência

```
# Ver referência de todos os commits (incluindo perdidos)
git reflog

# Recuperar commit "perdido"
git checkout <hash-do-reflog>

# Criar branch do commit recuperado
git checkout -b <nome-branch> <hash>

# Ver configuração de um comando
git help <comando>
```

DICAS RÁPIDAS

Situação	Comando
Esqueci de adicionar arquivo no último commit	git add <arquivo> → git commit --amend --no-edit
Commitei na branch errada	git reset HEAD~1 → mudar de branch → commitar
Quero descartar todas as mudanças	git reset --hard HEAD
Quero ver o que mudou antes de pull	git fetch → git diff origin/main
Conflito de merge e quero recomeçar	git merge --abort

Aliases Úteis

Adicione no seu .gitconfig:
git config --global alias.st status
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.unstage 'reset HEAD --'
git config --global alias.last 'log -1 HEAD'
git config --global alias.visual 'log --graph --oneline --all'

Uso: git st ao invés de git status