

**[Uma imagem com símbolo, Tipo de letra, Gráficos, captura de ecrã

Os conteúdos gerados por IA poderão estar incorretos.](https://creativecommons.org/licenses/by-nc/4.0/)**

**Luís Simões da Cunha (2025)**

Índice

[Parte 1: Introdução ao Git e à Gestão de Versões 🚀 8](#_Toc196779838)

[🎯 Objetivos de Aprendizagem 8](#_Toc196779839)

[1. O que é o Git? Por que é tão poderoso? 🛠️ 8](#_Toc196779840)

[2. Git vs Outros Sistemas de Controlo de Versões ⚡ 8](#_Toc196779841)

[3. Conceitos Fundamentais 📚 9](#_Toc196779842)

[4. Instalação e Configuração Básica 🛠️ 10](#_Toc196779843)

[Instalar o Git 10](#_Toc196779844)

[Configurar o Git (Essencial!) 10](#_Toc196779845)

[Confirmar a Instalação 10](#_Toc196779846)

[🚀 Conclusão da Parte 1 10](#_Toc196779847)

[Parte 2: Primeiro Contacto: Criar e Configurar Repositórios 🛠️ 11](#_Toc196779848)

[🎯 Objetivos de Aprendizagem 11](#_Toc196779849)

[1. Criar um Novo Repositório 🏗️ 11](#_Toc196779850)

[Passos: 11](#_Toc196779851)

[2. Clonar um Repositório Existente 🔍 11](#_Toc196779852)

[Exemplo: 11](#_Toc196779853)

[Opcional: Clonar para uma pasta com nome diferente 12](#_Toc196779854)

[3. Ignorar Ficheiros com .gitignore 🚫 12](#_Toc196779855)

[Como criar: 12](#_Toc196779856)

[Exemplos: 12](#_Toc196779857)

[Tornar o .gitignore ativo: 12](#_Toc196779858)

[4. Configurar Utilizador e Identidade 🧑‍💻 13](#_Toc196779859)

[Verificar a configuração: 13](#_Toc196779860)

[🚀 Conclusão da Parte 2 13](#_Toc196779861)

[Mini-Missão da Parte 2 🎯 14](#_Toc196779862)

[Parte 3: Trabalhar com Commits 📝✨ 15](#_Toc196779863)

[🎯 Objetivos de Aprendizagem 15](#_Toc196779864)

[1. Compreender o Fluxo de Trabalho: Working Directory, Staging Area e Repository 🔄 15](#_Toc196779865)

[Visual do processo: 15](#_Toc196779866)

[2. Adicionar Alterações (git add) ➕ 15](#_Toc196779867)

[Exemplos: 15](#_Toc196779868)

[Boas práticas: 16](#_Toc196779869)

[3. Fazer Commits (git commit) 🗂️ 16](#_Toc196779870)

[Comando básico: 16](#_Toc196779871)

[O que acontece num commit? 16](#_Toc196779872)

[4. Boas Práticas na Escrita de Mensagens de Commit ✍️ 17](#_Toc196779873)

[Exemplo: 17](#_Toc196779874)

[Dicas práticas: 17](#_Toc196779875)

[🚀 Conclusão da Parte 3 17](#_Toc196779876)

[Mini-Missão da Parte 3 🎯 18](#_Toc196779877)

[Parte 3: Trabalhar com Commits 📝✨ 19](#_Toc196779878)

[🎯 Objetivos de Aprendizagem 19](#_Toc196779879)

[1. O Fluxo de Trabalho no Git 🔄 19](#_Toc196779880)

[Fluxo Visual: 19](#_Toc196779881)

[2. Adicionar Alterações com git add ➕ 20](#_Toc196779882)

[Comando básico: 20](#_Toc196779883)

[3. Fazer Commits com git commit 🗂️ 20](#_Toc196779884)

[Comando: 20](#_Toc196779885)

[4. Boas Práticas na Escrita de Mensagens de Commit ✍️ 21](#_Toc196779886)

[Estrutura ideal: 21](#_Toc196779887)

[Exemplos bons: 21](#_Toc196779888)

[Dicas rápidas: 21](#_Toc196779889)

[🚀 Conclusão da Parte 3 21](#_Toc196779890)

[Mini-Missão da Parte 3 🎯 22](#_Toc196779891)

[Parte 4: Corrigir e Editar Commits 🔧 23](#_Toc196779892)

[🎯 Objetivos de Aprendizagem 23](#_Toc196779893)

[1. Corrigir o Último Commit (git commit --amend) ✍️ 23](#_Toc196779894)

[2. Desfazer Alterações Locais 🔄 23](#_Toc196779895)

[a) Desfazer alterações num ficheiro (não staged): 23](#_Toc196779896)

[b) Remover ficheiros do Staging (sem apagar alterações): 24](#_Toc196779897)

[3. Reverter Commits Indesejados (git revert) 🔥 24](#_Toc196779898)

[Exemplo: 24](#_Toc196779899)

[4. Reescrever Histórico Localmente ✨ 24](#_Toc196779900)

[Ferramenta: git rebase -i (interativo) 24](#_Toc196779901)

[🚀 Conclusão da Parte 4 25](#_Toc196779902)

[Mini-Missão da Parte 4 🎯 26](#_Toc196779903)

[Parte 5: Gestão de Branches 🌿✨ 27](#_Toc196779904)

[🎯 Objetivos de Aprendizagem 27](#_Toc196779905)

[1. Criar, Mudar, Eliminar e Renomear Branches 🛠️ 27](#_Toc196779906)

[Criar uma nova branch 27](#_Toc196779907)

[Mudar para uma branch 27](#_Toc196779908)

[Criar e mudar ao mesmo tempo 27](#_Toc196779909)

[Eliminar uma branch 27](#_Toc196779910)

[Renomear uma branch 28](#_Toc196779911)

[2. Entender o que é o HEAD 🎯 28](#_Toc196779912)

[Exemplo prático: 28](#_Toc196779913)

[3. Trabalhar em Múltiplas Linhas de Desenvolvimento 🔀 28](#_Toc196779914)

[4. Estratégias de Naming para Branches 🏷️ 29](#_Toc196779915)

[Exemplo visual: 29](#_Toc196779916)

[🚀 Conclusão da Parte 5 30](#_Toc196779917)

[Mini-Missão da Parte 5 🎯 30](#_Toc196779918)

[Parte 6: Sincronizar com Repositórios Remotos 🌎🔄 31](#_Toc196779919)

[🎯 Objetivos de Aprendizagem 31](#_Toc196779920)

[1. Clonar, Fetch, Pull e Push 🧩 31](#_Toc196779921)

[Clonar um Repositório (git clone) 31](#_Toc196779922)

[Buscar Atualizações (git fetch) 31](#_Toc196779923)

[Atualizar e Fundir (git pull) 31](#_Toc196779924)

[Enviar Alterações (git push) 32](#_Toc196779925)

[2. Criar e Acompanhar Tracking Branches 📡 32](#_Toc196779926)

[Confirmar o acompanhamento 32](#_Toc196779927)

[3. Gerir Remotes: Adicionar, Alterar e Remover 🔧 32](#_Toc196779928)

[Ver todos os remotes 32](#_Toc196779929)

[Adicionar um novo remote 33](#_Toc196779930)

[Alterar o URL de um remote 33](#_Toc196779931)

[Remover um remote 33](#_Toc196779932)

[4. Resolver Conflitos de Sincronização 🛠️ 33](#_Toc196779933)

[Como resolver: 33](#_Toc196779934)

[🚀 Conclusão da Parte 6 34](#_Toc196779935)

[Mini-Missão da Parte 6 🎯 34](#_Toc196779936)

[Parte 7: Merging e Resolução de Conflitos 🔀🧠 35](#_Toc196779937)

[🎯 Objetivos de Aprendizagem 35](#_Toc196779938)

[1. Como o Git Faz Merges 🔬 35](#_Toc196779939)

[2. Resolver Conflitos Manualmente e com Ferramentas Gráficas 🛠️ 35](#_Toc196779940)

[Quando ocorre um conflito: 35](#_Toc196779941)

[Resolver manualmente: 36](#_Toc196779942)

[Usar ferramentas gráficas para resolver 36](#_Toc196779943)

[3. Estratégias Avançadas de Merge 🧠🔬 36](#_Toc196779944)

[Principais estratégias: 37](#_Toc196779945)

[4. Boas Práticas para Evitar Conflitos 🚀 37](#_Toc196779946)

[🚀 Conclusão da Parte 7 38](#_Toc196779947)

[Mini-Missão da Parte 7 🎯 38](#_Toc196779948)

[Parte 8: Explorar e Analisar o Histórico de Commits 🔍📜 39](#_Toc196779949)

[🎯 Objetivos de Aprendizagem 39](#_Toc196779950)

[1. Ver Histórico de Alterações (git log) 🕰️ 39](#_Toc196779951)

[Exemplo básico: 39](#_Toc196779952)

[2. Limitar e Formatar Saídas 📏🖌️ 39](#_Toc196779953)

[3. Pesquisar Alterações Específicas 🔎 40](#_Toc196779954)

[a) Procurar palavras no histórico (git grep) 40](#_Toc196779955)

[b) Procurar alterações específicas no histórico (git log -S) 40](#_Toc196779956)

[4. Utilizar reflog para Recuperar Alterações Perdidas 🚑 41](#_Toc196779957)

[Como recuperar um commit perdido 41](#_Toc196779958)

[🚀 Conclusão da Parte 8 42](#_Toc196779959)

[Mini-Missão da Parte 8 🎯 42](#_Toc196779960)

[Parte 9: Manipular e Editar Histórico ✍️⏳ 43](#_Toc196779961)

[🎯 Objetivos de Aprendizagem 43](#_Toc196779962)

[1. Uso Seguro de git rebase 🎢 43](#_Toc196779963)

[Como funciona? 43](#_Toc196779964)

[Exemplo típico: 43](#_Toc196779965)

[Benefícios do rebase 43](#_Toc196779966)

[Comandos úteis: 43](#_Toc196779967)

[⚠️ Atenção: 44](#_Toc196779968)

[2. Ferramentas para Edição Pesada de Histórico 🛠️ 44](#_Toc196779969)

[a) git filter-branch 44](#_Toc196779970)

[b) git replace 44](#_Toc196779971)

[3. Dicas para Reescrever Histórico de Forma Segura 🛡️ 45](#_Toc196779972)

[🚀 Conclusão da Parte 9 45](#_Toc196779973)

[Mini-Missão da Parte 9 🎯 45](#_Toc196779974)

[Parte 10: Funcionalidades Avançadas e Boas Práticas 🚀🧠 46](#_Toc196779975)

[🎯 Objetivos de Aprendizagem 46](#_Toc196779976)

[1. Trabalhar com Tags 🏷️ 46](#_Toc196779977)

[Criar uma tag simples 46](#_Toc196779978)

[Criar uma tag anotada (com metadados e opção de assinatura) 46](#_Toc196779979)

[Ver e gerir tags 46](#_Toc196779980)

[2. Submódulos: Incluir Outros Repositórios 🔗 47](#_Toc196779981)

[Adicionar um submódulo 47](#_Toc196779982)

[Clonar um repositório com submódulos 47](#_Toc196779983)

[3. Hooks: Automatizar Tarefas no Ciclo de Vida do Git ⚙️ 47](#_Toc196779984)

[Exemplo de utilização: 47](#_Toc196779985)

[4. Estratégias de Colaboração Profissional 🤝 48](#_Toc196779986)

[a) **Forking Workflow** (ideal para projetos open-source) 48](#_Toc196779987)

[b) **Feature Branch Workflow** 48](#_Toc196779988)

[c) **Gitflow Workflow** 48](#_Toc196779989)

[5. Dicas de Segurança no Git 🔒 49](#_Toc196779990)

[a) Usar SSH para autenticação 49](#_Toc196779991)

[b) Assinar commits com GPG 49](#_Toc196779992)

[c) Gerir credenciais de forma segura 49](#_Toc196779993)

[🚀 Conclusão da Parte 10 50](#_Toc196779994)

[Super-Missão Final 🎯🚀 50](#_Toc196779995)

[🎓 Desafio Final: Missão Mestre do Git 🚀 51](#_Toc196779996)

[🎯 Objetivo 51](#_Toc196779997)

[🛠️ Descrição do Projeto 51](#_Toc196779998)

[🔥 Desafios que tens de cumprir 51](#_Toc196779999)

[🎖️ Critérios para considerar a missão concluída: 53](#_Toc196780000)

[🏆 Bónus (Opcional) 53](#_Toc196780001)

[🎉 Resultado final 53](#_Toc196780002)

# Parte 1: Introdução ao Git e à Gestão de Versões 🚀

## 🎯 Objetivos de Aprendizagem

* Entender o que é o Git e a sua importância.
* Comparar Git com outros sistemas de controlo de versões.
* Compreender conceitos fundamentais: repositórios, commits, branches e merges.
* Instalar e configurar o Git para começar a usar.

## 1. O que é o Git? Por que é tão poderoso? 🛠️

Imagina um **"cofre mágico"** onde todas as versões do teu projeto ficam guardadas, organizadas e acessíveis a qualquer momento.  
O Git é esse cofre! 🔒✨

🔹 **Definição simples**:  
Git é um sistema de controlo de versões distribuído que regista todas as alterações feitas a um conjunto de ficheiros ao longo do tempo.

🔹 **Principais superpoderes**:

* Voltar atrás no tempo e recuperar versões anteriores.
* Comparar diferentes versões facilmente.
* Trabalhar em várias versões do projeto ao mesmo tempo (branches).
* Combinar (merger) alterações feitas por diferentes pessoas de forma estruturada.

🔹 **Porque é tão popular**:

* Cada utilizador tem uma cópia completa do projeto (não depende sempre de uma ligação à internet).
* Os branches são leves e rápidos.
* Excelente suporte para colaboração (com ferramentas como o GitHub).

🎓 **Curiosidade**: O Git foi inicialmente criado por Linus Torvalds (criador do Linux) para gerir o gigantesco projeto do próprio kernel Linux!

## 2. Git vs Outros Sistemas de Controlo de Versões ⚡

| Git (Distribuído) | CVS/Subversion (Centralizado) |
| --- | --- |
| Cada pessoa tem cópia total do projeto. | Existe um servidor central único. |
| Permite trabalhar offline. | Precisa de ligação contínua ao servidor. |
| Branches e merges super rápidos. | Branches lentos e complicados. |
| Trabalhos independentes e flexíveis. | Depende de coordenação centralizada. |

**Resumo visual:**

Com Git, o **teu repositório é o teu castelo**. Com sistemas antigos, precisas sempre **pedir permissão no castelo principal**! 🏰

## 3. Conceitos Fundamentais 📚

Vamos criar o nosso "vocabulário Git" essencial:

| Termo | Significado |
| --- | --- |
| **Repositório** | Coleção de ficheiros e historial completo de alterações. |
| **Commit** | Um "snapshot" ou fotografia do projeto num dado momento. |
| **Branch** | Uma linha independente de desenvolvimento. |
| **Merge** | Combinar alterações de diferentes branches num só. |

🔹 **Repositórios**:

* Cada repositório contém **todos os commits** e **todas as branches**.

🔹 **Commits**:

* São **instantâneos** completos do projeto — não apenas diferenças parciais.
* Cada commit é identificado de forma única através de um código chamado **SHA-1**.

🔹 **Branches**:

* Permitem desenvolver novas funcionalidades ou corrigir erros **sem afetar o projeto principal**.

🔹 **Merges**:

* Juntam linhas de desenvolvimento separadas, **resolvendo diferenças** se necessário.

🎯 **Dica de memória**:  
Repositório = "caixa de histórias";  
Commit = "página da história";  
Branch = "linha alternativa da história";  
Merge = "juntar linhas para fazer uma história única".

## 4. Instalação e Configuração Básica 🛠️

### Instalar o Git

Escolhe a tua plataforma:

* **Windows**: [Git for Windows](https://gitforwindows.org/)
* **MacOS**: Usar brew install git (se tiveres Homebrew).
* **Linux**: Usa o gestor de pacotes (ex: sudo apt install git).

### Configurar o Git (Essencial!)

Abre o terminal ou linha de comandos:

git config --global user.name "O Teu Nome"  
git config --global user.email "o.teu.email@exemplo.com"

Estes dois comandos definem o teu nome e email, que aparecerão nos commits. 📝

### Confirmar a Instalação

Verifica se tudo está bem instalado:

git --version

E vê a tua configuração:

git config --list

🔥 **Mini-missão 1**:  
Instala o Git no teu computador, configura o teu nome e email, e verifica a versão instalada.

# 🚀 Conclusão da Parte 1

Parabéns! 🎉 Já tens uma base sólida sobre o que é o Git, porque é tão especial, e estás pronto para **começar a usá-lo de verdade** no teu próprio computador!

# Parte 2: Primeiro Contacto: Criar e Configurar Repositórios 🛠️

## 🎯 Objetivos de Aprendizagem

* Criar um repositório do zero.
* Clonar um projeto existente.
* Ignorar ficheiros que não devem ser versionados.
* Reforçar a configuração da tua identidade no Git.

## 1. Criar um Novo Repositório 🏗️

Queres começar um projeto novo e guardar o seu histórico? É aqui que entra o **git init**! 🚀

### Passos:

1. **Cria uma pasta** para o teu projeto (se ainda não existir):

mkdir meu-projeto  
cd meu-projeto

1. **Inicializa o repositório Git**:

git init

🔹 O comando cria uma pasta oculta chamada .git, onde o Git guarda toda a magia dos históricos, branches, merges, etc.

🧠 **Importante lembrar**: só com git init o Git começa a acompanhar as alterações no teu projeto. Antes disso, é só uma pasta "normal"!

## 2. Clonar um Repositório Existente 🔍

Se já existe um projeto online (ex: GitHub, GitLab), podes fazer uma **cópia completa** para o teu computador com **git clone**.

### Exemplo:

git clone https://github.com/utilizador/projeto.git

🔹 Este comando:

* Cria uma nova pasta com o projeto.
* Copia **todos os ficheiros** e **todo o histórico de alterações**.
* Liga o teu repositório local ao remoto para poderes fazer *pull* e *push* depois.

### Opcional: Clonar para uma pasta com nome diferente

git clone https://github.com/utilizador/projeto.git nome-que-quiseres

🎯 **Dica visual**:  
Clonar é como "descarregar uma cápsula do tempo" completa: não trazes só os ficheiros atuais, mas **todas as versões anteriores também!** 📦

## 3. Ignorar Ficheiros com .gitignore 🚫

Nem tudo o que está na tua pasta precisa (ou deve) ser guardado no repositório: ficheiros temporários, senhas, configurações locais, etc.

Para isso existe o **ficheiro mágico .gitignore**!

### Como criar:

1. Cria um ficheiro chamado .gitignore na raiz do teu projeto.
2. Adiciona lá os ficheiros ou pastas que queres ignorar.

### Exemplos:

# Ignorar todos os ficheiros .log  
\*.log  
# Ignorar a pasta temporaria/  
temporaria/  
# Ignorar ficheiro de configuração local  
config-local.yaml

### Tornar o .gitignore ativo:

Depois de criar/modificar o .gitignore, adiciona-o ao Git como qualquer outro ficheiro:

git add .gitignore  
git commit -m "Adicionar ficheiro .gitignore para ignorar ficheiros desnecessários"

🎯 **Dica para reter**:  
.gitignore = "Lista VIP invertida" – só quem NÃO está na lista é que entra no repositório! 🎟️

## 4. Configurar Utilizador e Identidade 🧑‍💻

⚡ **Este passo é essencial**: define quem tu és nas alterações que vais fazer!

Se ainda não o fizeste, usa:

git config --global user.name "O Teu Nome"  
git config --global user.email "o.teu.email@exemplo.com"

### Verificar a configuração:

git config --list

🔹 Se quiseres configurar **diferentes nomes/emails para projetos específicos**, podes correr o comando **sem o --global** dentro da pasta do projeto:

git config user.name "Outro Nome"  
git config user.email "outro.email@exemplo.com"

📜 **Resumo Mental**:  
**Global** → aplica-se a todos os projetos.  
**Local** → aplica-se só àquele projeto.

# 🚀 Conclusão da Parte 2

Excelente! 🎉 Agora já sabes:

* Criar repositórios novos,
* Clonar projetos que já existem,
* Ignorar ficheiros que não deves guardar,
* E garantir que todas as tuas alterações têm a tua assinatura digital correta! 🖋️

# Mini-Missão da Parte 2 🎯

1. Cria uma pasta chamada primeiro-repo.
2. Inicializa um repositório com git init.
3. Cria um ficheiro chamado README.md, escreve algo dentro.
4. Cria um .gitignore e adiciona lá a linha \*.tmp.
5. Configura o teu nome e email.
6. Faz o teu primeiro commit!

# Parte 3: Trabalhar com Commits 📝✨

## 🎯 Objetivos de Aprendizagem

* Entender como o Git organiza e controla as alterações.
* Aprender o ciclo completo: adicionar ➔ confirmar ➔ guardar.
* Escrever mensagens de commit claras e profissionais.

## 1. Compreender o Fluxo de Trabalho: Working Directory, Staging Area e Repository 🔄

Imagina que o teu projeto tem **3 níveis** como numa cozinha:

| Área | O que é no Git? | Analogia culinária 🍳 |
| --- | --- | --- |
| **Working Directory** | A tua pasta de trabalho, onde editas ficheiros. | Onde estás a cozinhar. |
| **Staging Area** | Onde preparas alterações que queres guardar. | Prato pronto, mas ainda na bancada. |
| **Repository** | Onde ficam os commits, ou seja, a história oficial. | O livro de receitas, onde registaste o prato! 📖 |

### Visual do processo:

(editar ficheiros) → git add → (staging area) → git commit → (repository)

🔹 **Working Directory**: Onde as alterações ainda estão *soltas*.  
🔹 **Staging Area (Índice)**: Escolhes quais alterações queres guardar.  
🔹 **Repository**: Onde os commits ficam guardados para sempre.

🎯 **Dica**: Só o que passa pelo *Staging* com git add é que vai parar ao *Repository* com git commit!

## 2. Adicionar Alterações (git add) ➕

Antes de fazer um commit, tens de dizer ao Git **quais alterações** queres guardar.  
É aqui que entra o git add.

### Exemplos:

* Adicionar um ficheiro específico:

git add ficheiro.txt

* Adicionar todos os ficheiros alterados:

git add .

⚡ **Atenção**: git add . adiciona **tudo** o que mudou — mesmo ficheiros que, por distração, poderias não querer guardar.

### Boas práticas:

* Usa git status antes de git add para ver o que mudou.
* Adiciona apenas o que realmente deves versionar.

## 3. Fazer Commits (git commit) 🗂️

Agora que já tens alterações no Staging Area, podes fazer um **commit** — ou seja, guardar essas alterações no historial do projeto.

### Comando básico:

git commit -m "Mensagem descritiva aqui"

🔹 O -m permite escrever a mensagem diretamente na linha de comandos.  
🔹 Sem -m, o Git abre o editor de texto padrão para escrever a mensagem.

### O que acontece num commit?

* O Git tira uma "fotografia" do estado atual dos ficheiros *no Staging*.
* Regista essa fotografia no historial do projeto, com:
  + Um identificador único (SHA-1).
  + A mensagem que explicaste.
  + O teu nome, email e data.

## 4. Boas Práticas na Escrita de Mensagens de Commit ✍️

Uma boa mensagem de commit é como uma **nota para o futuro**: ajuda-te a ti e aos outros a entender *o que mudou* e *porquê*.

🔹 **Formato recomendado**:

Linha 1: Resumo breve e imperativo (máx. 50 caracteres)  
Linha 2: (em branco)  
Linhas 3+: Explicação mais detalhada (se necessário)

### Exemplo:

Corrigir erro de login no formulário  
Corrigia um erro que impedia o utilizador de iniciar sessão  
quando o campo de email era deixado em branco.

### Dicas práticas:

* Usa sempre **tempo verbal imperativo** (ex: "Corrigir", "Adicionar", "Remover") → como se desses uma ordem ao projeto.
* Sê **específico**, mas evita descrições demasiado longas.
* Se a alteração for complexa, usa o corpo da mensagem para explicar *porquê* a alteração foi necessária.
* Pequenos commits bem descritos são melhores que grandes commits confusos.

# 🚀 Conclusão da Parte 3

Parabéns! 🎉 Agora já sabes:

* Como preparar as alterações (staging),
* Como gravá-las oficialmente (commit),
* E como escrever mensagens que realmente ajudam a entender a evolução do projeto! 🛤️

# Mini-Missão da Parte 3 🎯

1. Cria um ficheiro ideias.txt no teu repositório.
2. Escreve 3 ideias no ficheiro.
3. Usa git status para ver o estado.
4. Adiciona o ficheiro ao staging (git add).
5. Faz um commit com uma boa mensagem (git commit -m "Adicionar lista inicial de ideias").
6. Verifica o historial com git log.

# Parte 3: Trabalhar com Commits 📝✨

## 🎯 Objetivos de Aprendizagem

* Compreender o ciclo natural de trabalho no Git.
* Saber adicionar alterações ao *Staging Area* (git add).
* Saber criar commits (git commit) de forma correta.
* Aprender a escrever mensagens de commit claras e úteis.

## 1. O Fluxo de Trabalho no Git 🔄

Imagina o Git como uma cozinha onde tens:

* Bancada de trabalho 🍽️ (*Working Directory*)
* Prato de preparação 🍛 (*Staging Area*)
* Livro de receitas 📖 (*Repository*)

Cada nível tem o seu papel:

| Área | Definição | Analogia |
| --- | --- | --- |
| **Working Directory** | Onde editas os ficheiros. | A cozinha onde trabalhas. |
| **Staging Area (Index)** | Onde preparas as alterações que queres guardar. | O prato montado antes de ir para a fotografia. |
| **Repository** | Onde ficam guardadas as alterações oficialmente. | O livro de receitas com fotos e instruções. |

### Fluxo Visual:

(Editar ficheiros) → git add → (Staging) → git commit → (Repository)

🔹 Primeiro, editas ficheiros no teu projeto.  
🔹 Depois, selecionas quais as alterações a guardar (git add).  
🔹 Finalmente, guardas essas alterações como um "snapshot" permanente (git commit).

🎯 **Nota Importante**:  
O Git não guarda apenas diferenças entre versões — guarda *fotos* completas do projeto no momento do commit!

## 2. Adicionar Alterações com git add ➕

Antes de fazer um commit, tens de dizer explicitamente ao Git quais as alterações que queres guardar.

### Comando básico:

* Adicionar um ficheiro específico:

git add nome-do-ficheiro.txt

* Adicionar todos os ficheiros modificados:

git add .

⚡ **Atenção**:  
git add . adiciona **tudo** o que está alterado. Convém sempre confirmar o que vai ser adicionado (git status)!

## 3. Fazer Commits com git commit 🗂️

Depois de adicionares as alterações ao *Staging Area*, estás pronto para criar um commit.

### Comando:

git commit -m "Mensagem de commit descritiva"

Este comando:

* Cria um novo snapshot do projeto no estado atual do *Staging*.
* Guarda essa versão para sempre no repositório.
* Liga automaticamente este commit ao anterior, criando uma linha de história.

🧠 **Lembra-te**:  
O commit regista exatamente o que estava no *Staging* na altura. Se alteraste ficheiros depois do git add, essas alterações não entram no commit!

## 4. Boas Práticas na Escrita de Mensagens de Commit ✍️

Uma mensagem de commit é como uma **mensagem para o teu "eu do futuro"** — e para qualquer outra pessoa que venha trabalhar no projeto!

### Estrutura ideal:

* **Linha 1**: Resumo curto (até 50 caracteres), no imperativo.
* **Linha 2**: Em branco.
* **Linhas 3+**: Detalhes adicionais, se necessário.

### Exemplos bons:

Corrigir bug de validação no formulário de registo  
Corrige a verificação do campo "email", que não detetava endereços  
inválidos introduzidos manualmente.

### Dicas rápidas:

* Usa verbos no **imperativo**: "Adicionar", "Corrigir", "Remover", "Atualizar"...
* Sê **claro e direto**: explica o "o quê" e, se necessário, "porquê".
* Evita mensagens vagas como "alterações" ou "update final".

🎯 **Dica extra para grandes equipas**:  
Mensagens de commit bem escritas poupam horas a tentar perceber alterações antigas!

# 🚀 Conclusão da Parte 3

Parabéns! 🎉  
Agora sabes:

* Como preparar ficheiros para commit,
* Como criar commits de forma organizada,
* E como escrever mensagens de qualidade que contam a história do teu projeto!

# Mini-Missão da Parte 3 🎯

1. Cria um ficheiro chamado plano.txt no teu repositório.
2. Escreve 3 objetivos que tenhas para aprender Git.
3. Usa git add plano.txt para o preparar para commit.
4. Faz um commit com a mensagem: Adicionar plano inicial de estudo de Git.
5. Vê o histórico com git log.

# Parte 4: Corrigir e Editar Commits 🔧

## 🎯 Objetivos de Aprendizagem

* Corrigir o último commit sem criar um novo.
* Desfazer alterações locais antes de fazer commit.
* Reverter commits que já foram registados no repositório.
* Reescrever o histórico local de commits com segurança.

## 1. Corrigir o Último Commit (git commit --amend) ✍️

Às vezes cometemos pequenos erros:

* Esquecemo-nos de adicionar um ficheiro.
* Escrevemos uma mensagem de commit errada.

**Boa notícia:** não precisas fazer um novo commit para corrigir o anterior — podes usar:

git commit --amend

🔹 Se apenas quiseres corrigir a mensagem:

git commit --amend -m "Nova mensagem mais clara"

🔹 Se esqueceste de adicionar ficheiros:

git add ficheiro-esquecido.txt  
git commit --amend

🧠 **Importante**:  
Isto **reescreve** o último commit. Só o deves fazer se ainda **não o enviaste (push)** para um repositório remoto!

## 2. Desfazer Alterações Locais 🔄

Nem todas as alterações merecem ser registadas. Às vezes queremos **reverter o ficheiro ao estado anterior**.

### a) Desfazer alterações num ficheiro (não staged):

git restore nome-do-ficheiro

ou na versão mais clássica:

git checkout -- nome-do-ficheiro

🔹 Isto **descarta** as alterações locais **não guardadas** no Staging Area.

### b) Remover ficheiros do Staging (sem apagar alterações):

Se fizeste git add mas ainda queres corrigir:

git restore --staged nome-do-ficheiro

🔹 O ficheiro volta ao estado de "modificado", mas não vai ser commitado.

🎯 **Resumo Visual**:

* git restore ficheiro → Desfaz alterações no disco.
* git restore --staged ficheiro → Tira do Staging sem apagar alterações no disco.

## 3. Reverter Commits Indesejados (git revert) 🔥

Se já fizeste um commit e queres **anular** o que foi feito **sem apagar o histórico**, usa:

git revert <id-do-commit>

🔹 Isto cria um novo commit que **inverte** as alterações do commit original.

### Exemplo:

git revert 1a2b3c4d

* Cria um commit novo que desfaz as alterações do commit 1a2b3c4d.

🛡️ **Nota**:  
**git revert é seguro** para projetos partilhados porque preserva todo o historial!

## 4. Reescrever Histórico Localmente ✨

**Às vezes** queremos limpar o nosso histórico **antes** de partilhar com outros:

* Combinar vários commits pequenos.
* Melhorar mensagens antigas.
* Corrigir pequenos enganos.

### Ferramenta: git rebase -i (interativo)

Para reescrever os últimos N commits:

git rebase -i HEAD~N

Exemplo para editar os últimos 3 commits:

git rebase -i HEAD~3

🔹 O Git abre um editor com algo como:

pick a1b2c3d Mensagem do commit 1  
pick d4e5f6g Mensagem do commit 2  
pick h7i8j9k Mensagem do commit 3

Podes então:

* Mudar pick para reword (editar mensagem).
* Mudar pick para squash (juntar commits).
* Mudar a ordem dos commits.

Quando terminares, o Git orienta-te passo-a-passo até a limpeza estar feita.

⚡ **Aviso importante**:  
Só uses git rebase em commits que **AINDA NÃO enviaste para o repositório remoto**!

# 🚀 Conclusão da Parte 4

Parabéns! 🎉  
Agora sabes:

* Corrigir o último commit facilmente.
* Desfazer alterações locais com segurança.
* Reverter commits sem estragar o histórico.
* Melhorar o histórico local antes de partilhar.

# Mini-Missão da Parte 4 🎯

1. Altera um ficheiro do teu repositório.
2. Comete com uma mensagem errada.
3. Usa git commit --amend para corrigir a mensagem.
4. Experimenta modificar outro ficheiro e usa git restore para cancelar a alteração.
5. Faz dois commits experimentais e depois junta-os com git rebase -i HEAD~2.

# Parte 5: Gestão de Branches 🌿✨

## 🎯 Objetivos de Aprendizagem

* Criar, mudar, eliminar e renomear branches no Git.
* Compreender o que é o HEAD.
* Trabalhar com múltiplas linhas de desenvolvimento.
* Aplicar boas práticas na escolha de nomes para branches.

## 1. Criar, Mudar, Eliminar e Renomear Branches 🛠️

### Criar uma nova branch

Queres desenvolver uma nova funcionalidade sem afetar o projeto principal? Cria uma branch:

git branch nome-da-branch

🔹 Isto cria a branch **mas não muda** para ela.

### Mudar para uma branch

Para mudar de linha de trabalho:

git checkout nome-da-branch

Ou, mais moderno e prático (tudo num só comando):

git switch nome-da-branch

🔹 Pensa em git checkout como "trocar de mundo" e git switch como "trocar de pista" 🚂.

### Criar e mudar ao mesmo tempo

Podes criar e entrar numa branch diretamente:

git checkout -b nome-da-branch

ou com o novo comando:

git switch -c nome-da-branch

### Eliminar uma branch

Quando já não precisas de uma branch:

git branch -d nome-da-branch

🔹 Usa -d para apagar de forma segura (só se a branch já tiver sido fundida). Se quiseres forçar:

git branch -D nome-da-branch

### Renomear uma branch

Se quiseres mudar o nome de uma branch:

git branch -m nome-antigo nome-novo

🎯 **Nota prática**: Se já fizeste "push" da branch antiga, terás de eliminar a antiga no remoto e fazer push da nova.

## 2. Entender o que é o HEAD 🎯

🔹 O **HEAD** é como um **marcador** que indica **onde estás** no teu projeto.

* Se estiveres numa branch chamada main, o HEAD aponta para refs/heads/main.
* Se fizeres checkout de um commit antigo diretamente, o HEAD "descola-se" (chamamos a isso **detached HEAD**).

### Exemplo prático:

git symbolic-ref HEAD  
# refs/heads/main

Significa que estás na branch main!

🧠 **Resumo mental**:  
HEAD = "Onde estou agora?" no teu projeto.

## 3. Trabalhar em Múltiplas Linhas de Desenvolvimento 🔀

Branches permitem desenvolver diferentes funcionalidades, corrigir erros, experimentar ideias, **tudo em paralelo**!

🔹 Exemplo típico:

* main ou master: código de produção.
* develop: linha de desenvolvimento principal.
* feature/nova-funcionalidade: novas funcionalidades.
* bugfix/corrigir-erro-login: correções específicas.
* hotfix/urgente-corrigir: correções críticas para produção.

**Cenário de trabalho**:

1. Cria uma branch para a nova funcionalidade.
2. Desenvolve sem medo de estragar a produção.
3. Faz merge da branch para develop ou main quando tudo estiver testado!

## 4. Estratégias de Naming para Branches 🏷️

Dar bons nomes às branches ajuda **toda a equipa** (e a ti no futuro!) a perceber o que cada linha de trabalho representa.

🔹 **Boas práticas**:

* Usa um prefixo que indique o tipo de trabalho: feature/, bugfix/, hotfix/, release/, etc.
* Usa nomes **descritivos** e **curtos**:  
  Exemplo: feature/adicionar-pagamento-cartao
* Usa hífen (-) para separar palavras.
* Evita espaços, acentos ou caracteres especiais.
* Se aplicável, inclui o número da tarefa/ticket:  
  Exemplo: feature/1234-login-google

### Exemplo visual:

feature/adicionar-autenticacao-google  
bugfix/corrigir-erro-sessao-expirada  
hotfix/corrigir-vulnerabilidade-xss  
release/versao-2.0

🎯 **Resumo mental**:  
Nome da branch = "tipo/trabalho-descritivo"

# 🚀 Conclusão da Parte 5

Parabéns! 🎉  
Agora já sabes:

* Criar e mudar branches como um profissional.
* O que é o HEAD e como ele controla onde estás.
* Gerir múltiplas linhas de desenvolvimento de forma organizada.
* Nomear branches de maneira clara e profissional.

# Mini-Missão da Parte 5 🎯

1. Cria uma branch feature/hello-world.
2. Muda para ela.
3. Cria um ficheiro hello.txt com o texto "Olá Mundo".
4. Faz um commit.
5. Volta à branch main.
6. Verifica se o ficheiro hello.txt ainda não está lá!

# Parte 6: Sincronizar com Repositórios Remotos 🌎🔄

## 🎯 Objetivos de Aprendizagem

* Dominar os comandos clone, fetch, pull e push.
* Entender como funcionam as tracking branches.
* Gerir remotes de forma eficaz (adicionar, alterar, remover).
* Resolver conflitos de sincronização com confiança.

## 1. Clonar, Fetch, Pull e Push 🧩

### Clonar um Repositório (git clone)

O ponto de partida para trabalhar com um projeto já existente é **clonar** o repositório:

git clone url-do-repositorio

🔹 Isto cria uma cópia **completa** do projeto, incluindo todo o histórico.  
🔹 Também cria um **remote** chamado origin que aponta para o repositório original.

### Buscar Atualizações (git fetch)

O fetch é como "pedir novidades" ao repositório remoto:

git fetch origin

🔹 Traz **todas as atualizações** (commits, branches, etc.) do repositório remoto.  
🔹 **Não altera** automaticamente a tua branch atual — apenas atualiza as tracking branches (ex.: origin/main).

### Atualizar e Fundir (git pull)

Se quiseres **buscar atualizações e fundi-las automaticamente** com o que tens localmente:

git pull origin nome-da-branch

🔹 Combina fetch + merge num só passo.  
🔹 Pode originar conflitos se houver alterações incompatíveis.

### Enviar Alterações (git push)

Depois de fazer commits locais, queres **enviar** as alterações para o repositório remoto:

git push origin nome-da-branch

🔹 Isto publica oficialmente o teu trabalho!  
🔹 O repositório remoto é atualizado para refletir os teus novos commits.

🎯 **Resumo visual**:

fetch = buscar novidades 📥  
pull = buscar e misturar 📥➕  
push = enviar novidades 📤

## 2. Criar e Acompanhar Tracking Branches 📡

Uma **tracking branch** é uma branch local que **acompanha** uma branch remota.

Quando fazes:

git checkout nome-da-branch

e essa branch só existe no remoto (origin/nome-da-branch), o Git:

* Cria automaticamente uma **branch local**.
* Liga-a à correspondente branch no origin.

🔹 Isto permite fazer git pull e git push sem precisar de especificar a origem e o destino — o Git já sabe!

### Confirmar o acompanhamento

Para veres qual branch está a acompanhar qual:

git branch -vv

## 3. Gerir Remotes: Adicionar, Alterar e Remover 🔧

### Ver todos os remotes

git remote -v

🔹 Mostra os remotes e os seus URLs associados.

### Adicionar um novo remote

git remote add nome-remoto url-do-repositorio

🔹 Exemplo:

git remote add upstream https://github.com/autor/original.git

(Útil, por exemplo, para acompanhar o repositório original de um fork.)

### Alterar o URL de um remote

git remote set-url nome-remoto novo-url

### Remover um remote

git remote remove nome-remoto

🛡️ **Nota**:  
Se removeste um remote, a tua configuração de tracking branches relacionada pode precisar ser atualizada!

## 4. Resolver Conflitos de Sincronização 🛠️

Conflitos surgem quando:

* Tu modificaste uma parte do projeto,
* E alguém também modificou a mesma parte,
* E tentas unir ambas as versões (git pull ou git merge).

### Como resolver:

1. O Git indica quais ficheiros têm conflitos (git status).
2. Abre os ficheiros e procura as marcações especiais:

<<<<<<< HEAD  
Versão local  
=======  
Versão remota  
>>>>>>> origin/main

1. Escolhe, combina ou adapta o conteúdo.
2. Marca o conflito como resolvido:

git add nome-do-ficheiro

1. Finaliza com um commit de merge, se necessário:

git commit

# 🚀 Conclusão da Parte 6

Parabéns! 🎉  
Agora sabes:

* Trabalhar eficientemente com repositórios remotos.
* Entender e controlar as tracking branches.
* Adicionar, alterar e remover remotes.
* Resolver conflitos de sincronização de forma organizada.

# Mini-Missão da Parte 6 🎯

1. Clona um repositório de teste.
2. Cria uma nova branch local chamada experiencia-sync.
3. Faz uma alteração num ficheiro e commita.
4. Faz push dessa branch para o remoto.
5. Pede a alguém para alterar o mesmo ficheiro no repositório remoto (ou altera tu diretamente via GitHub, por exemplo).
6. Faz git pull e resolve o conflito manualmente!

# Parte 7: Merging e Resolução de Conflitos 🔀🧠

## 🎯 Objetivos de Aprendizagem

* Compreender como o Git faz merges de branches.
* Resolver conflitos manualmente e usando ferramentas gráficas.
* Dominar estratégias avançadas de merge (recursive, ours, theirs).
* Conhecer boas práticas para evitar conflitos desnecessários.

## 1. Como o Git Faz Merges 🔬

Quando fazes um **merge** no Git (por exemplo, git merge feature-x), o Git:

1. **Identifica o ponto de partida comum** entre as duas branches (o *merge base*).
2. **Compara** o que mudou em cada branch desde o ponto comum.
3. **Aplica automaticamente** as mudanças quando possível.
4. **Assinala conflitos** caso alterações incompatíveis sejam detetadas.

🔹 Se não houver conflitos, o Git faz um **commit automático** de merge.

🧠 **Conceito-chave**:  
O Git é altamente otimizado para **texto estruturado por linhas** (como código-fonte).

## 2. Resolver Conflitos Manualmente e com Ferramentas Gráficas 🛠️

### Quando ocorre um conflito:

* O Git interrompe o merge.
* Marca os ficheiros problemáticos com secções especiais:

<<<<<<< HEAD  
Conteúdo da tua branch atual  
=======  
Conteúdo da branch que estás a fundir  
>>>>>>> feature-x

🔹 A tua missão: **escolher, combinar ou reescrever** o conteúdo.

### Resolver manualmente:

1. Editar o(s) ficheiro(s).
2. Remover os marcadores (<<<<<<<, =======, >>>>>>>).
3. Guardar o resultado final.
4. Marcar como resolvido:

git add nome-do-ficheiro

1. Finalizar o merge:

git commit

### Usar ferramentas gráficas para resolver

O Git integra-se com muitas ferramentas visuais de merge como:

* **VS Code** (abre automaticamente conflitos graficamente)
* **Meld**
* **kdiff3**
* **Beyond Compare**
* **P4Merge**

Para iniciar uma ferramenta gráfica manualmente:

git mergetool

🔹 O Git abrirá a tua ferramenta preferida para te ajudar a resolver os conflitos de forma visual.

🎯 **Dica prática**:  
Podes configurar a tua ferramenta favorita com:

git config --global merge.tool nome-da-ferramenta

## 3. Estratégias Avançadas de Merge 🧠🔬

Por vezes, precisamos de indicar ao Git **como preferimos resolver conflitos** automaticamente.

### Principais estratégias:

| Estratégia | Descrição |
| --- | --- |
| recursive | **Padrão**. Usa o *merge base* para aplicar alterações. Resolve conflitos simples automaticamente. |
| ours | Durante conflitos, **mantém a tua versão** (da branch atual). A versão da branch a fundir é ignorada. |
| theirs | Durante conflitos, **aceita a versão da outra branch** e descarta a tua versão local. |

🔹 Exemplo: fazer merge preferindo sempre a tua versão (ours):

git merge -s ours nome-da-branch

🔹 Exemplo: ao usar recursive, podes especificar preferências:

git merge -X ours nome-da-branch  
git merge -X theirs nome-da-branch

⚡ **Nota Importante**:  
Usar ours ou theirs **não impede o merge completo** — apenas afeta **resolução de conflitos específicos**!

## 4. Boas Práticas para Evitar Conflitos 🚀

✔️ **Comunica frequentemente** com os teus colegas de equipa.  
✔️ **Puxa (pull) antes de começar** a trabalhar no projeto.  
✔️ **Evita grandes alterações em simultâneo** em ficheiros críticos.  
✔️ **Divide o trabalho** em múltiplas pequenas branches, se possível.  
✔️ **Faz commits pequenos e frequentes** — ajudam a isolar problemas.  
✔️ **Nomeia branches de forma clara** para evitar misturas confusas.

# 🚀 Conclusão da Parte 7

Parabéns! 🎉  
Agora já sabes:

* Como o Git funde alterações automaticamente.
* Como resolver conflitos à mão ou usando ferramentas gráficas.
* Como aplicar estratégias avançadas de merge.
* Como evitar a maior parte dos conflitos antes mesmo que aconteçam!

# Mini-Missão da Parte 7 🎯

1. Cria duas branches a partir da main: feature-A e feature-B.
2. Em feature-A, altera a primeira linha de projeto.txt.
3. Em feature-B, altera a mesma linha de projeto.txt de forma diferente.
4. Tenta fazer git merge feature-B a partir da feature-A.
5. Resolve o conflito manualmente ou com uma ferramenta gráfica!

# Parte 8: Explorar e Analisar o Histórico de Commits 🔍📜

## 🎯 Objetivos de Aprendizagem

* Explorar o histórico de alterações usando git log.
* Personalizar a visualização do histórico para facilitar a análise.
* Pesquisar alterações específicas dentro do repositório.
* Recuperar alterações aparentemente perdidas usando git reflog.

## 1. Ver Histórico de Alterações (git log) 🕰️

O comando mais usado para "voltar atrás no tempo" é:

git log

🔹 Mostra a lista de commits, do mais recente para o mais antigo.

Cada commit exibido inclui:

* ID do commit (SHA-1).
* Autor.
* Data.
* Mensagem do commit.

### Exemplo básico:

commit a2c1234...  
Author: Luís Cunha  
Date: 29 April 2025  
 Corrigir erro no login

## 2. Limitar e Formatar Saídas 📏🖌️

🔹 **Limitar número de commits**:

git log -n 5

Mostra apenas os 5 commits mais recentes.

🔹 **Visualizar apenas resumo (one-liner)**:

git log --oneline

Mostra uma linha por commit — perfeito para ter uma visão rápida!

🔹 **Ver o gráfico de branches**:

git log --graph --oneline --all

Mostra um gráfico com a estrutura das branches e merges!  
(Ótimo para visualizar o histórico de um projeto mais complexo.)

🔹 **Personalizar a saída**:

git log --pretty=format:"%h - %an: %s"

Mostra apenas:

* Abreviação do SHA (%h),
* Nome do autor (%an),
* Mensagem (%s).

🎯 **Dica prática**: Cria os teus próprios "formatos" para ver exatamente o que queres!

## 3. Pesquisar Alterações Específicas 🔎

### a) Procurar palavras no histórico (git grep)

Queremos encontrar commits que alteraram uma linha contendo a palavra "senha":

git grep "senha"

🔹 Pesquisa nos ficheiros **atuais**.

### b) Procurar alterações específicas no histórico (git log -S)

Se quiseres encontrar commits que **adicionaram ou removeram** uma linha específica:

git log -S"senha"

🔹 Mostra commits onde houve uma alteração no número de ocorrências da palavra "senha" no projeto.

🔹 Outra opção poderosa:

git log -G"regex"

Pesquisa alterações no código que **correspondem a uma expressão regular**!

## 4. Utilizar reflog para Recuperar Alterações Perdidas 🚑

Já apagaste uma branch ou commit sem querer? 😨  
O Git salva os teus movimentos recentes no **reflog**!

🔹 Ver o histórico de posições do HEAD:

git reflog

Irás ver algo assim:

a2c1234 HEAD@{0}: commit: Corrigir erro no login  
d9b5678 HEAD@{1}: checkout: moving from main to feature-x  
...

Cada linha mostra:

* A posição antiga do HEAD,
* A ação realizada,
* O ID do commit.

### Como recuperar um commit perdido

Se encontras o commit no reflog, podes:

git checkout a2c1234

Ou, para restaurá-lo para a tua branch:

git cherry-pick a2c1234

🚨 **Nota**: O reflog é uma ferramenta poderosa mas é **local** — cada máquina mantém o seu próprio histórico!

# 🚀 Conclusão da Parte 8

Parabéns! 🎉  
Agora sabes:

* Explorar o histórico de commits de forma poderosa e flexível.
* Limitar, formatar e personalizar as pesquisas no Git.
* Localizar alterações específicas no passado.
* Recuperar alterações perdidas com segurança usando reflog.

# Mini-Missão da Parte 8 🎯

1. Executa git log --graph --oneline --all no teu repositório para ver o gráfico.
2. Usa git log -S"senha" ou outra palavra para encontrar alterações relevantes.
3. Experimenta git reflog e identifica o teu último checkout ou commit!

# Parte 9: Manipular e Editar Histórico ✍️⏳

## 🎯 Objetivos de Aprendizagem

* Utilizar git rebase de forma segura para limpar o histórico.
* Conhecer ferramentas de edição pesada de histórico (git filter-branch, git replace).
* Aplicar boas práticas ao reescrever histórico em ambientes partilhados.

## 1. Uso Seguro de git rebase 🎢

O git rebase permite **reorganizar commits** para criar um histórico mais limpo e linear.

### Como funciona?

* Em vez de fazer um merge tradicional (com múltiplos pais num commit),
* o rebase **pega nos teus commits** e **"reaplica-os"** sobre o topo de outra branch.

### Exemplo típico:

git checkout feature  
git rebase main

🔹 Isto faz com que a branch feature pareça que foi criada **diretamente** a partir do ponto mais recente da main.

### Benefícios do rebase

✔️ Um histórico mais **linear** e fácil de ler.  
✔️ Facilita a revisão de alterações.  
✔️ Evita merges confusos durante desenvolvimento.

### Comandos úteis:

* **Abortar** um rebase em andamento:
* git rebase --abort
* **Interativo**: editar, reordenar ou combinar commits:
* git rebase -i HEAD~3
* 🔹 Permite:
  + pick: manter commit como está.
  + reword: alterar a mensagem de commit.
  + squash: combinar commits.

### ⚠️ Atenção:

Nunca faças rebase em **branches já partilhadas** com outros utilizadores!  
Isso altera a história partilhada e pode causar conflitos difíceis de resolver.

## 2. Ferramentas para Edição Pesada de Histórico 🛠️

Quando queres **reescrever grandes partes do histórico**, existem ferramentas específicas:

### a) git filter-branch

Permite aplicar transformações a muitos commits antigos.

Exemplo: mudar o autor de todos os commits de uma branch:

git filter-branch --env-filter '  
if [ "$GIT\_AUTHOR\_EMAIL" = "email\_antigo@example.com" ];  
then  
 GIT\_AUTHOR\_EMAIL="novo\_email@example.com";  
 GIT\_AUTHOR\_NAME="Novo Nome";  
fi  
' -- --all

🔹 **Atenção**: filter-branch é poderoso mas lento e deve ser usado com muito cuidado.

### b) git replace

Cria **substituições temporárias** de commits, sem alterar o repositório original.

Exemplo:

git replace commit-antigo novo-commit

🔹 Ideal para testes ou correções sem destruir o histórico original.

🚨 **Nota Importante**: Tanto filter-branch como replace **não devem ser usados** em repositórios partilhados sem alinhamento claro com a equipa!

## 3. Dicas para Reescrever Histórico de Forma Segura 🛡️

✔️ **Evita reescrever branches públicas**: altera apenas o teu histórico local.  
✔️ **Faz sempre backup** antes de qualquer operação destrutiva (git clone, git branch backup).  
✔️ **Comunica** com a equipa antes de forçar push após rebases ou filtros (git push --force-with-lease).  
✔️ **Usa --force-with-lease em vez de --force** para evitar sobrescrever o trabalho dos outros:

git push --force-with-lease

# 🚀 Conclusão da Parte 9

Parabéns! 🎉  
Agora sabes:

* Usar git rebase de forma segura para manter um histórico limpo.
* Utilizar git filter-branch e git replace para manipulações mais profundas.
* Aplicar boas práticas para manter a colaboração segura e eficaz.

# Mini-Missão da Parte 9 🎯

1. Cria três commits simples (commit1, commit2, commit3).
2. Executa git rebase -i HEAD~3 para:
   * Alterar a mensagem de commit2.
   * Combinar (squash) commit3 com commit2.
3. Faz git log --oneline para ver o novo histórico.
4. (Opcional) Cria uma substituição de commit usando git replace!

# Parte 10: Funcionalidades Avançadas e Boas Práticas 🚀🧠

## 🎯 Objetivos de Aprendizagem

* Utilizar tags para marcar versões importantes.
* Trabalhar com submódulos para integrar outros repositórios.
* Automatizar tarefas usando Git hooks.
* Conhecer workflows de colaboração profissionais.
* Aumentar a segurança no Git com SSH, GPG e boas práticas de credenciais.

## 1. Trabalhar com Tags 🏷️

As **tags** servem para assinalar commits importantes, como versões de produção (v1.0, v2.1-beta, etc.).

### Criar uma tag simples

git tag nome-da-tag

🔹 Cria uma tag "leve" apontando para o commit atual.

### Criar uma tag anotada (com metadados e opção de assinatura)

git tag -a nome-da-tag -m "Mensagem da tag"

🔹 Estas são recomendadas para versões públicas, porque permitem assinatura e mensagem associada.

### Ver e gerir tags

* Listar tags:
* git tag
* Mostrar detalhes de uma tag:
* git show nome-da-tag
* Enviar tags para o repositório remoto:
* git push origin nome-da-tag

🎯 **Dica prática**:  
Para publicar todas as tags de uma vez:

git push --tags

## 2. Submódulos: Incluir Outros Repositórios 🔗

Um **submódulo** é um repositório Git embutido noutro repositório principal.

### Adicionar um submódulo

git submodule add url-do-repositorio caminho/para/submodulo

🔹 Isto cria uma ligação ao repositório externo, sem misturar os conteúdos diretamente.

### Clonar um repositório com submódulos

git clone --recurse-submodules url-do-repositorio

🔹 Se já clonaste o repositório:

git submodule update --init --recursive

🧠 **Nota Importante**:  
Submódulos são ótimos para dependências externas fixas, mas adicionam complexidade na gestão de atualizações!

## 3. Hooks: Automatizar Tarefas no Ciclo de Vida do Git ⚙️

**Hooks** são scripts automáticos que o Git executa em momentos chave, como antes de um commit (pre-commit) ou depois de um push (post-push).

### Exemplo de utilização:

* Prevenir commits sem mensagens:
  + Criar .git/hooks/commit-msg
  + Tornar o script executável:
  + chmod +x .git/hooks/commit-msg

🔹 Alguns hooks comuns:

| Hook | Quando é acionado |
| --- | --- |
| pre-commit | Antes de fazer commit |
| commit-msg | Para validar a mensagem |
| pre-push | Antes de fazer push |
| pre-receive | Antes do servidor aceitar push |

🎯 **Dica prática**: Usa ferramentas como **Husky** para gerir hooks de forma moderna e multiplataforma.

## 4. Estratégias de Colaboração Profissional 🤝

### a) **Forking Workflow** (ideal para projetos open-source)

* Faz um fork do projeto principal.
* Trabalha no teu repositório pessoal.
* Envia alterações via **Pull Request**.

### b) **Feature Branch Workflow**

* Cada nova funcionalidade é feita numa nova branch:
* git checkout -b feature/nome-da-funcionalidade
* Facilita o isolamento de novas funcionalidades.

### c) **Gitflow Workflow**

Estratégia formal para grandes projetos:

* main → branch de produção.
* develop → branch de integração de funcionalidades.
* Branches especiais para feature/, release/ e hotfix/.

🔹 Automatizável com ferramentas como **git-flow**.

🧠 **Resumo visual**:

Forking → colaboração aberta.  
Feature Branch → isolamento de funcionalidades.  
Gitflow → gestão rigorosa de ciclos de vida.

## 5. Dicas de Segurança no Git 🔒

### a) Usar SSH para autenticação

* Gera uma chave SSH:
* ssh-keygen -t ed25519 -C "teu-email@example.com"
* Adiciona a chave pública na tua conta de repositórios (GitHub, GitLab, etc.).

### b) Assinar commits com GPG

* Instalar GPG.
* Gerar uma chave:
* gpg --full-generate-key
* Configurar Git para usar a chave:
* git config --global user.signingkey ID-da-chave  
  git config --global commit.gpgsign true

### c) Gerir credenciais de forma segura

* Usa o **credential helper** do Git:
* git config --global credential.helper cache
* Ou preferencialmente:
* git config --global credential.helper store
* Em ambientes corporativos, usa **credential managers** integrados (como o Git Credential Manager).

# 🚀 Conclusão da Parte 10

Parabéns! 🎉  
Agora sabes:

* Trabalhar com tags de forma profissional.
* Integrar subprojetos usando submódulos.
* Automatizar tarefas no ciclo de vida do Git com hooks.
* Escolher e aplicar workflows adequados ao teu projeto.
* Tornar o teu uso do Git mais seguro e fiável.

# Super-Missão Final 🎯🚀

1. Cria uma tag anotada para o teu projeto atual.
2. Adiciona um pequeno projeto externo como submódulo.
3. Cria um hook pre-commit que bloqueie commits vazios.
4. Escolhe um workflow (Forking, Feature Branch ou Gitflow) e aplica-o no teu próximo projeto.
5. Configura SSH e, se quiseres, começa a assinar os teus commits com GPG!

# 🎓 Desafio Final: Missão Mestre do Git 🚀

## 🎯 Objetivo

Simular **um projeto real**, aplicando todas as técnicas, boas práticas e funcionalidades que aprendeste nas 10 partes.

## 🛠️ Descrição do Projeto

Imagina que estás a liderar um pequeno projeto chamado **"Git-Explorer"** — um programa fictício simples (ex: um ficheiro README.md e alguns scripts .py ou .sh), que vais construir e gerir com Git da forma mais profissional possível.

## 🔥 Desafios que tens de cumprir

1. **Criar o repositório**
   * Inicia um novo repositório Git chamado git-explorer.
   * Configura o teu nome e email de utilizador.
2. **Organizar o trabalho em branches**
   * Cria uma branch develop a partir da main.
   * Cria uma branch feature/iniciar-projeto para adicionar o README.md.
3. **Trabalhar com Commits**
   * Escreve commits claros, pequenos e bem descritos.
   * Aplica git add seletivo e confirma o que vai em cada commit.
4. **Simular Erros e Corrigir**
   * Escreve uma má mensagem de commit de propósito.
   * Usa git commit --amend para corrigir.
   * Simula alterações locais e desfaz com git restore.
5. **Usar Merges e Resolver Conflitos**
   * Cria uma branch feature/adicionar-script.
   * Em develop, altera também o README.md.
   * Faz merge da feature/adicionar-script → vai gerar conflito → resolve-o manualmente!
6. **Navegar no Histórico**
   * Usa git log, git log --graph, git log -S"palavra-chave".
   * Simula "perder" um commit e recupera-o com git reflog.
7. **Limpar o Histórico**
   * Usa git rebase -i para combinar dois commits pequenos numa única melhoria do projeto.
8. **Trabalhar com Tags**
   * Quando a primeira versão mínima estiver funcional, cria uma tag v1.0.
9. **Adicionar Submódulos**
   * Simula adicionar uma biblioteca externa como submódulo.
10. **Automatizar Tarefas**
    * Cria um hook pre-commit que:
      + Valide que todos os ficheiros .md têm conteúdo (não estão vazios).
11. **Implementar Estratégias de Workflow**
    * Trabalha sempre numa branch de funcionalidade (feature/) e faz merges para develop.
    * Faz um "simulado" de Pull Request interno (simular com merge manual).
12. **Garantir Segurança**
    * Configura autenticação SSH.
    * Se quiseres ir além, gera uma chave GPG e assina commits importantes.

## 🎖️ Critérios para considerar a missão concluída:

| Critério | Descrição |
| --- | --- |
| Organização | Repositório limpo, com histórico linear e claro. |
| Boas práticas | Mensagens de commit úteis, branches bem nomeadas. |
| Gestão de conflitos | Resolver conflitos de merge de forma segura. |
| Segurança | Uso de SSH e (opcional) GPG. |
| Uso de funcionalidades avançadas | Tags, submódulos, hooks, workflows. |

# 🏆 Bónus (Opcional)

Se quiseres mesmo ir **nível ninja**, podes:

* Criar um segundo repositório GitHub, fazer fork e simular uma colaboração entre "dois utilizadores" (por exemplo, usando duas contas diferentes ou repositórios simulados).
* Configurar CI/CD simples (por exemplo, no GitHub Actions) para testar se o projeto é construído corretamente.

# 🎉 Resultado final

No fim desta missão, terias:

* Um projeto real para mostrar no teu portefólio (se quiseres).
* Um domínio sólido de **Git real, aplicado como um profissional**.
* E acima de tudo: uma compreensão prática que vai fazer Git parecer uma ferramenta natural para ti.