PET.COMP

GIT E GITHUB

NA PRÁTICA: CONTROLE DE VERSÃO PARA PROJETOS MODERNOS

1 Tutorial Git..... 1
1.1 Introdução..... 1
1.1.1 Conceitos-Chave do Git..... 1
1.1.2 Como o Git funciona no dia a dia..... 2
1.1.3 Git e GitHub..... 3
1.2 Instalação..... 3
1.2.1 Git Bash..... 4
1.2.2 Verificando Instalação..... 4
1.2.3 Editor de Texto Padrão..... 5
1.2.4 Variável de Ambiente PATH..... 5
1.2.5 Como adicionar o Git ao PATH após a instalação..... 5
1.2.6 Fim de Linha (Line Endings)..... 6
1.2.7 Atualizando ou Desinstalando o Git..... 6
1.2.8 Resolução de Problemas na Instalação..... 6
1.3 Configuração..... 7
1.3.1 Nome e Endereço de E-mail de Usuário..... 7
1.3.2 Configuração..... 7
1.3.3 Exemplos..... 8
1.4 Primeiros Passos com o Git..... 9
1.4.1 Etapas principais para começar..... 9
1.4.2 Criando a pasta do projeto..... 9
1.4.3 Inicializando o Git..... 9
1.4.4 Possíveis Problemas..... 11
1.5 Novos Arquivos no Git..... 11
1.5.1 Criando um Novo Arquivo..... 11
1.5.2 Listar os Arquivos da Pasta..... 12
1.5.4 Verificar o Status do Arquivo no Git..... 12
1.5.5 Diferença entre Arquivos Untracked e Tracked..... 13
1.6 Ambiente de Staging (Staging Area)..... 13
1.6.1 Principais Comandos de Staging..... 13
1.6.2 Adicionar Vários Arquivos ao Mesmo Tempo..... 14
1.6.3 Verificar Arquivos no Staging..... 14
1.6.4 Remover Arquivo do Staging (Unstage)..... 14
Problemas Comuns..... 15
1.7 Commit..... 15
1.7.1 Comandos principais para commits..... 15
1.7.2 Commit com Mensagem (-m)..... 15
1.7.3 Commit de Todas as Alterações (-a)..... 16
1.7.4 Tentando Commitar um Arquivo Novo com -a..... 16
1.7.5 Mensagens de Commit em Múltiplas Linhas..... 16
1.7.6 Outras opções úteis de commit..... 17
1.7.7 Erros comuns e como corrigir..... 17
1.7.8 Visualizar o Histórico de Commits..... 17

1.8 Git Tagging (Tagueamento no Git)..... 18
1.8.1 Criando Tags..... 18
1.8.2 Enviar Tags para o Repositório Remoto..... 19
1.8.3 Excluir Tags..... 19
1.8.4 Atualizar ou Substituir uma Tag..... 20
1.9 Git Stash..... 20
1.10 Git History..... 22
1.10.1 Ver Histórico..... 22
1.10.2 Comparar Mudanças..... 23
1.10.3 Mostrar Commits..... 24
1.11 Git Branch..... 25
1.11.1 Principais Comandos de Branch..... 25
1.11.2 Fluxo de Trabalho no Branch..... 26
1.11.3 Boas Práticas..... 26
1.12 Git Branch Merge..... 27
1.12.1 Principais Comandos de Merge..... 27
1.12.2 Boas Práticas para Merging..... 28
1.12.3 Tipos de Merge..... 28
1.12.4 Conflitos de Merge..... 28
1.12.5 Excluir Branch após Merge..... 29
2 Git e GitHub..... 30
2.1 Início..... 30
2.1.1 Criando sua conta no GitHub..... 30
2.1.2 Criar Um Repositório..... 30
2.2 Git Security com SSH..... 31
2.2.1 Configuração do SSH..... 32
2.2.2 Resolvendo Problemas (Troubleshooting)..... 33
2.3 Adicionando Chave SSH..... 34
2.3.1 Copiando sua Chave Pública..... 34
2.3.2 Adicionando a Chave ao GitHub..... 34
2.4 Configurando o Remote Origin do GitHub com SSH..... 36
2.4.1 Testando sua Conexão SSH..... 36
2.4.2 Obtendo o Endereço SSH do Repositório..... 36
2.4.3 Adicionando ou Atualizando o Remote Origin..... 37
2.5 Editando Código Diretamente no GitHub..... 37
2.5.1 Editando um Arquivo..... 38
2.5.2 Escrevendo a Mensagem do Commit..... 38
2.5.3 Salvando as Alterações (Commit)..... 39
2.5.4 Criando uma Nova Branch pelo Editor..... 39
2.6 Git Pull do GitHub..... 39
2.6.1 Git Fetch..... 39
2.6.2 Git Merge..... 41
2.6.3 Git Pull..... 41
2.7 Git Push para o GitHub..... 42

2.7.1 Basic Push..... 42
2.7.2 Force Push..... 42
2.7.3 Push Tags..... 43
Confirmação no GitHub..... 43
2.8 Branches no GitHub..... 44
2.8.1 Switch Branch (Trocar de branch)..... 45
2.8.2 Delete Branch (Deletar branch)..... 45
2.8.3 Rename Branch (Renomear branch)..... 45
2.8.4 Merge Branch (Mesclar branches)..... 45
2.8.5 View Branches (Visualizar branches)..... 46
2.8.6 Protected Branches (Branches protegidos)..... 46
2.9 Git Pull Branch from GitHub..... 46
2.9.1 Atualizar o repositório local..... 46
2.9.2 Verificar status..... 46
2.9.3 Listar branches locais e remotos..... 46
2.9.4 Trazendo o branch remoto para local..... 47
2.9.5 Confirmar atualização..... 47
2.9.6 Conferir branches locais..... 47
2.10 Git Push Branch to GitHub..... 48
2.10.1 Criar e mudar para um novo branch..... 48
2.10.2 Editar arquivos e verificar status..... 48
Problemas comuns..... 49
2.11 GitHub Flow..... 49
2.11.1 Etapas do GitHub Flow..... 49
2.11.2 Benefícios do GitHub Flow..... 50
3 Git Contribute..... 52
3.1 GitHub Fork..... 52
3.1.1 O que é um Fork?..... 52
3.1.2 Quando usar Fork?..... 52
3.1.3 Como fazer um Fork no GitHub..... 52
3.2 Clonando um Fork do GitHub..... 53
3.2.1 Como clonar o repositório..... 53
3.2.2 Entrando no repositório clonado..... 54
3.2.3 Conferindo o histórico..... 54
3.2.4 Configurando Remotes..... 54
4 Git Undo..... 57
4.1 Git Revert..... 57
4.1.1 Executando o git revert..... 57
4.1.2 Revisando alterações após o revert..... 57
4.2 Git Reset..... 58
4.2.1 Resumo das opções do git reset..... 58
4.2.2 Como encontrar o commit para resetar..... 58
4.2.3 git reset --soft..... 60
4.2.4 git reset --mixed (padrão)..... 60

4.2.5 git reset --hard..... 60
4.2.6 Revisando mudanças após o reset..... 60
4.3 Git Amend..... 61
4.3.1 Alterar Mensagem do Último Commit..... 61
4.3.2 Adicionar Arquivos ao Último Commit..... 61
4.3.3 Remover Arquivos do Último Commit..... 62
4.3.4 Exemplo de Log Antes e Depois..... 62
4.4 Git Rebase..... 62
4.4.1 Rebase Básico..... 63
4.4.2 Rebase Interativo..... 63
4.4.3 Continuar, Abortar ou Pular..... 63
4.5 Git Reflog..... 64
4.5.1 Mostrar o Reflog..... 65
4.5.2 Encontrar e Recuperar Commits Perdidos..... 65
4.5.3 Limpar o Reflog..... 66
4.5.4 Dicas & Boas Práticas..... 66
4.6 Git Recovery..... 67
4.6.1 Recuperar Commits Perdidos com git reflog..... 67
4.6.2 Restaurar um Branch Deletado..... 67
4.6.3 Recuperar Arquivo Deletado ou Alterado..... 67
4.5.4 Recuperar Após um Hard Reset..... 68

1 Tutorial Git

1.1 Introdução

O Git é um sistema de controle de versão distribuído criado por Linus Torvalds em 2005 — o mesmo criador do kernel do Linux. Desde então, é mantido principalmente por Junio Hamano e se tornou a ferramenta mais utilizada no mundo para rastrear alterações em códigos e permitir trabalho colaborativo entre desenvolvedores.

Atualmente, mais de 70% dos desenvolvedores utilizam o Git como ferramenta principal de controle de versão. Ele permite que pessoas trabalhem juntas de qualquer lugar do mundo, mantendo um histórico completo e seguro do projeto e possibilitando reverter alterações quando necessário.

Em termos simples, o Git funciona como uma "máquina do tempo" para o seu projeto: ele registra todas as versões dos arquivos, permitindo que você volte para qualquer ponto anterior, compare alterações e trabalhe em novas funcionalidades sem bagunçar o código principal.

1.1.1 Conceitos-Chave do Git

- **Repositório (Repository):** Pasta do seu projeto monitorada pelo Git. Contém todos os arquivos e o histórico de alterações.

💡 *Exemplo*: Uma pasta `meu-projeto/` onde o Git acompanha tudo que acontece.

- **Clone**: Faz uma cópia completa de um repositório remoto para o seu computador.

💡 *Exemplo*: git clone https://github.com/usuario/repositorio.git

- **Stage (Staging Area)**: Área onde você coloca as alterações que deseja salvar no próximo commit.

💡 *Exemplo*: git add arquivo.txt

- **Commit**: Registro permanente das alterações adicionadas à área de stage.

💡 *Exemplo*: git commit -m "Adiciona função de login"

- **Branch**: Linha de desenvolvimento independente dentro do projeto, usada para criar novas funcionalidades ou testar ideias.

💡 *Exemplo*: git branch nova-funcionalidade

- **Merge**: Combina as alterações de uma branch com outra.

💡 *Exemplo*: git merge nova-funcionalidade

- **Pull**: Baixa as alterações mais recentes do repositório remoto para o seu computador.

💡 *Exemplo*: git pull origin main

- **Push**: Envia suas alterações para o repositório remoto.

💡 *Exemplo*: git push origin main

1.1.2 Como o Git funciona no dia a dia

1. **Inicializar** um repositório Git em uma pasta:

```
git init
```

   Isso cria uma pasta oculta `.git` que armazena todo o histórico e configurações do repositório.

2. **Modificar arquivos**: Sempre que você altera, adiciona ou remove um arquivo, o Git detecta essas mudanças.
3. **Selecionar alterações (Stage)**: Você decide quais arquivos quer salvar no próximo commit:

```
git add arquivo.txt
```

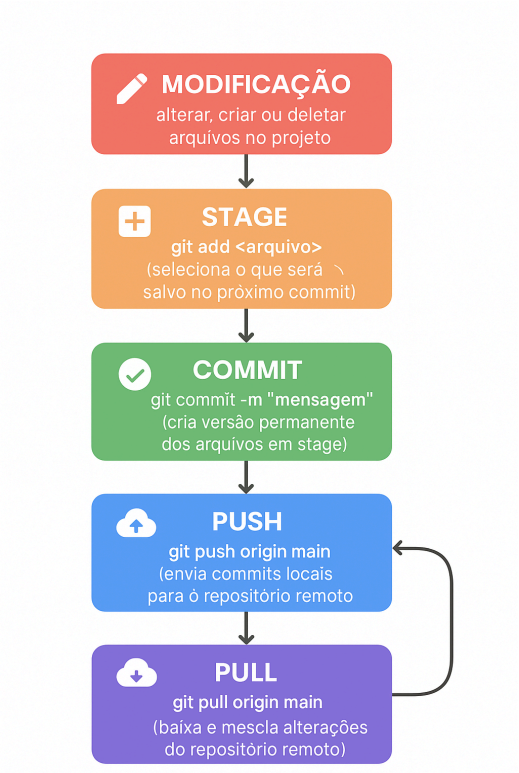
4. **Registrar alterações (Commit)**: Cria uma versão permanente do estado atual dos arquivos:

```
git commit -m "Mensagem descritiva"
```

5. **Histórico e reversões**: Você pode visualizar todas as alterações feitas:

```
git log
```

   E até voltar para versões anteriores quando necessário.



1.1.3 Git e GitHub

Embora sejam frequentemente mencionados juntos, **Git** e **GitHub** não são a mesma coisa:

- **Git** → Ferramenta que gerencia o histórico e as versões do seu projeto no seu computador (e também pode se conectar a servidores).

- **GitHub** → Plataforma online que hospeda repositórios Git e oferece recursos extras de colaboração, como issues, pull requests e integração com outras ferramentas.

O **GitHub** é o maior serviço de hospedagem de código do mundo e foi adquirido pela **Microsoft** em 2018.

1.2 Instalação

Você pode instalar o Git gratuitamente no site: https://git-scm.com .

Windows:

1. Baixe e execute o instalador.
2. Clique em “Next” para aceitar as configurações recomendadas.
3. Isso instalará o Git e o Git Bash.

macOS:

- Se você usa o Homebrew, abra o terminal e digite:

```
brew install git
```

- Ou baixe o arquivo *.dmg* e arraste o Git para a pasta Aplicativos.

macOS:

- Abra o terminal e use seu gerenciador de pacotes.
- por exemplo, no Ubuntu:

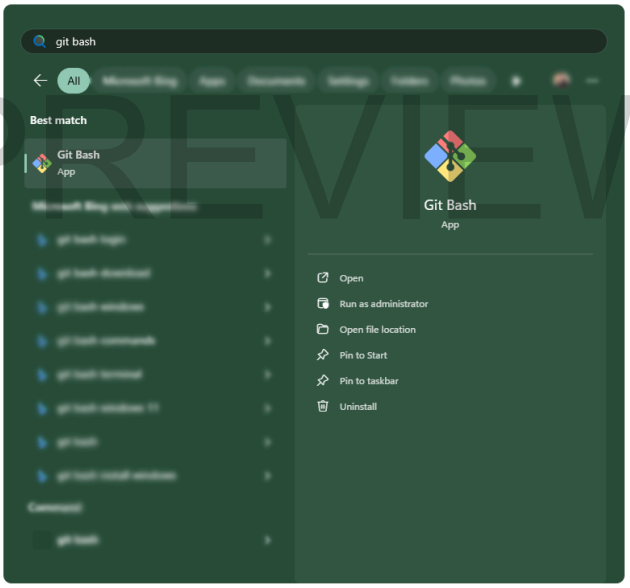
```
sudo apt-get install git
```

Após a instalação, você estará pronto para utilizar o Git no seu terminal ou prompt de comando.

1.2.1 Git Bash

O Git Bash é um terminal para Windows que permite utilizar os comandos do Git.
Após a instalação do Git, você poderá encontrar o Git Bash na Área de Trabalho ou no Menu Iniciar do seu sistema.

Ele funciona como um Prompt de Comando, mas com comandos extras no estilo Unix (como ls e pwd).d)



1.2.2 Verificando Instalação

Após instalar, verifique se o Git está funcionando:

1. Abra o terminal (ou o Git Bash no Windows).
2. Rode o comando:

```
git -- version
```

3. Se estiver instalado corretamente, você verá algo como:

```
git version X.Y.Z
```

1.2.3 Editor de Texto Padrão

Durante a instalação, é perguntado qual editor de texto será o padrão para o Git.
Esse editor será aberto quando for necessário escrever mensagens (como mensagens de commit).

Exemplo: definir o VS Code como editor padrão:

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

Se você não tiver certeza, pode escolher o Bloco de Notas. É possível mudar essa configuração depois.

Exemplo: definir o Bloco de Notas como editor padrão:

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

1.2.4 Variável de Ambiente PATH

Escolher adicionar o Git ao PATH significa que você poderá usar os comandos Git em qualquer terminal.Isso é altamente recomendado durante a instalação.

Se não adicionar, só será possível usar o Git no Git Bash (Windows) ou no Terminal (macOS/Linux).

Exemplo: verificar se o Git está no PATH

```
git --version
git version 2.43.0.windows.1
```

Se aparecer um erro, é necessário adicionar o Git ao PATH.

1.2.5 Como adicionar o Git ao PATH após a instalação

Windows:

1. Pesquise por **"Variáveis de Ambiente"** no Menu Iniciar e abra.
2. Clique em **"Variáveis de Ambiente..."**.
3. Na seção **"Variáveis do Sistema"**, encontre a variável **Path** e clique em **Editar**.
4. Clique em **Novo** e adicione os caminhos para as pastas `bin` e `cmd` do Git (por exemplo: `C:\Program Files\Git\bin` e `C:\Program Files\Git\cmd`).
5. Clique em **OK** e reinicie o terminal.

macOS:

- Se instalado pelo Homebrew, o PATH é configurado automaticamente.
- Caso contrário, abra o Terminal e adicione esta linha ao arquivo `~/.zshrc` ou `~/.bash_profile`:

```
export PATH="/usr/local/bin:$PATH"
```

Salve o arquivo e execute:

```
source ~/.zshrc
```

ou

```
source ~/.bash_profile
```

1.2.6 Fim de Linha (Line Endings)

O Git pode converter caracteres de fim de linha em arquivos de texto. No Windows, o mais comum é selecionar "Checkout Windows-style, commit Unix-style line endings" durante a instalação. Isso evita problemas ao compartilhar código com usuários de outros sistemas operacionais.

1.2.7 Atualizando ou Desinstalando o Git

- **Atualizar:** Baixe e execute o instalador mais recente, ou use seu gerenciador de pacotes.

  ```
  Exemplos:
  brew upgrade git
  sudo apt-get upgrade git
  ```

  É recomendável manter o Git atualizado para ter as últimas funcionalidades e correções de segurança.

- **Desinstalar:**
  - Windows → Use "Adicionar ou Remover Programas".
  - macOS/Linux → Use seu gerenciador de pacotes.

1.2.8 Resolução de Problemas na Instalação

Se algo não funcionar, siga estas dicas:
💡 Feche e reabra o terminal, ou reinicie o computador antes de tentar novamente.

Problemas comuns:

1. **"git não é reconhecido como comando interno ou externo"**
   - O Git não está no PATH. Adicione a pasta `bin` do Git ao PATH e reinicie o terminal.
2. **Erros de permissão ("Permission denied")**
   - No Windows: Abra o Git Bash ou terminal como Administrador.
   - No macOS/Linux: Use `sudo` se necessário.
3. **Erros SSL ou HTTPS ao clonar/push**
   - Verifique a conexão com a internet e atualize o Git.
4. **Versão incorreta do Git**
   - Verifique com `git --version` e atualize pelo site git-scm.com.

1.3 Configuração

Antes de começar a usar o Git, é importante configurar seu nome e e-mail. Essas informações serão registradas em cada commit que você fizer, identificando quem fez cada alteração no código.

💡 Dica para iniciantes: Essa configuração é segura. Você pode alterar esses valores a qualquer momento — isso só muda como seu nome e email aparecem nos commits futuros (os commits antigos mantêm as informações originais).

1.3.1 Nome e Endereço de E-mail de Usuário

O nome será anexado a todos os commits que você fizer.

Exemplo – Definir nome globalmente:

```
git config --global user.name "Seu Nome"
```

- Se você cometer um erro de digitação, basta executar o comando novamente com o valor correto.
- A opção --global aplica a configuração para todos os repositórios no seu computador.

O e-mail também será anexado aos seus commits.

Exemplo – Definir e-mail globalmente:

```
git config --global user.email "voce@exemplo.com"
```

- Troque pelo seu e-mail real.
- Esse e-mail provavelmente será o mesmo usado ao criar sua conta no GitHub (para manter a autoria dos commits associada ao seu perfil).
- Se você esquecer de configurar, o Git pedirá as informações na primeira vez que tentar fazer um commit.

O Git usa nome e e-mail para rotular cada commit. Sem essas informações, ele não saberá quem fez cada alteração.

Com essas configurações mínimas feitas, você já pode começar a trabalhar normalmente com o Git.

1.3.2 Configuração

Para ver todas as configurações do Git no seu sistema:

```
git config --list
```

Exemplo de saída:

```
user.name=Seu Nome
user.email=voce@exemplo.com
core.editor=code --wait
alias.st=status
init.defaultbranch=main
```

Para visualizar apenas um valor específico:

```
git config user.name
```

Saída esperada:

```
Seu Nome
```

Para alterar suas configurações basta executar novamente o `git config` com o novo valor.

No entanto, para remover use a opção `--unset`.

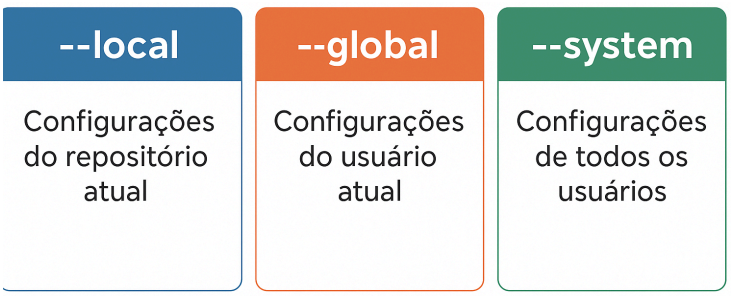
Exemplo – Remover configuração de editor padrão:

```
git config --global --unset core.editor
```


Você pode definir o nome padrão da branch principal para novos repositórios (por exemplo, `main` em vez de `master`):

```
git config --global init.defaultBranch main
```

O Git possui três níveis de configuração:



Ordem de precedência (o mais específico sobrepõe o mais genérico):

1. Local (repositório atual)
2. Global (usuário atual)
3. Sistema (todos os usuários)

1.3.3 Exemplos

- Configuração Local (apenas para o repositório atual):

```
git config user.name "Nome do Projeto"
```

- Configuração Global (para todos os repositórios do usuário atual):

```
git config --global user.name "Nome Global"
```

- Configuração de Sistema (para todos os usuários do computador):

```
git config --system user.name "Nome do Sistema"
```

1.4 Primeiros Passos com o Git

Agora que o Git já está instalado e configurado com seu nome e e-mail, podemos começar a utilizá-lo.
Nosso primeiro passo será **criar um repositório Git** e entender o que isso significa.

1.4.1 Etapas principais para começar

1. Criar uma pasta para o projeto
2. Acessar a pasta no terminal
3. Inicializar um repositório Git

1.4.2 Criando a pasta do projeto

Vamos começar criando uma nova pasta para armazenar nosso projeto:

```
mkdir meu_projeto
cd meu_projeto
```

- `mkdir` → cria um novo diretório (pasta).
- `cd` → muda o diretor de trabalho para a pasta escolhida.

Agora estamos dentro da pasta correta e já podemos inicializar o Git!

Dica (Windows):

Se você estiver no Windows, pode abrir o **Git Bash diretamente na pasta do projeto**:

- Clique com o botão direito na pasta no **Explorador de Arquivos**
- Escolha **Git Bash Here**
- Isso abrirá um terminal já na localização correta

1.4.3 Inicializando o Git

Com a pasta criada e acessada, basta rodar o seguinte comando para inicializar o Git:

```
git init
```

Saída esperada:

```
Initialized empty Git repository in
/Users/usuario/meu_projeto/.git/
```

Pronto! Você acabou de criar seu **primeiro repositório Git**!

Um **repositório Git** é uma pasta que o Git monitora para controlar alterações nos arquivos. Ele guarda todo o histórico de versões do seu projeto, permitindo voltar atrás, comparar versões e colaborar com outras pessoas.

O que acontece quando usamos `git init`?

Ao inicializar um repositório, no Git cria uma pasta oculta chamada `.git` dentro do projeto. É lá que ficam armazenadas todas as informações necessárias para controlar versões.

Para visualizar:

Linux/macOS

```
ls -a
```

Saída:

```
.   ..   .git
```

Windows

Habilite a opção **"Mostrar arquivos ocultos"** no Explorador de Arquivos para enxergar a pasta `.git`.

1.4.4 Possíveis Problemas

Erro: `git: command not found`
Solução: verifique se o Git está instalado e configurado no PATH do sistema. Reinicie o terminal se necessário.

Erro: `permission denied` **(Permissão negada)**
Solução:

- **Windows** → abra o terminal como Administrador
- **Linux/macOS** → execute o comando com `sudo`

Agora temos o ambiente pronto para começar a versionar nossos projetos!

1.5 Novos Arquivos no Git

Um **novo arquivo** é qualquer arquivo que você criou ou copiou para dentro da pasta do seu projeto, mas que ainda **não foi informado ao Git** para ser acompanhado.

Pontos principais:

- Criar um novo arquivo (com um editor de texto)
- `ls` → lista os arquivos da pasta
- `git status` → mostra quais arquivos estão sendo rastreados
- Diferença entre **arquivos rastreados (tracked)** e **não rastreados (untracked)**

1.5.1 Criando um Novo Arquivo

Ao inicializar um repositório, ele começa vazio. Vamos criar um primeiro arquivo com um editor de texto e salvar na pasta do projeto.

Exemplo de arquivo HTML simples (salve como `index.html`):

```
<!DOCTYPE html>
<html>
      <head>
            <title>Hello World!</title>
      </head>
      <body>

            <h1>Hello world!</h1>
            <p>This is the first file in my new Git Repo.</p>

      </body>
</html>
```

1.5.2 Listar os Arquivos da Pasta

Para verificar os arquivos existentes no diretório do projeto, usamos o comando:

```
ls
```

Saída esperada:


```
index.html
```

1.5.4 Verificar o Status do Arquivo no Git

Agora vamos checar se o Git já está rastreando o arquivo:

```
git status
```

Saída típica:

```
On branch master

No commits yet

Untracked files:
  (use "git add ..." to include in what will be committed)
    index.html
```

```
nothing added to commit but untracked files present (use "git add"
to track)
```

O Git identificou `index.html`, mas ele está como **não rastreado (untracked)**.

1.5.5 Diferença entre Arquivos Untracked e Tracked

- **Untracked (não rastreados):** arquivos criados/copiados para a pasta, mas que ainda não foram adicionados ao controle do Git.
- **Tracked (rastreados):** arquivos que o Git já acompanha e monitora mudanças.

Para transformar um arquivo **untracked** em **tracked**, será necessário adicioná-lo à área de staging (`git add`) – assunto da próxima seção.

Problemas Comuns

- **O arquivo não aparece no `ls`:** verifique se salvou na pasta correta.
  - Use `pwd` para confirmar o diretório atual.
- **Arquivo não listado no `git status`:** confira se está no repositório certo e se o arquivo foi salvo corretamente.

1.6 Ambiente de Staging (Staging Area)

O **ambiente de staging** (ou *staging area*) é como uma **sala de espera** para suas alterações.

Você o utiliza para dizer ao Git **quais arquivos exatamente** deseja incluir no próximo commit.

Isso te dá **controle total** sobre o que entrará no histórico do projeto.

1.6.1 Principais Comandos de Staging

- `git add <arquivo>` → adiciona um arquivo ao staging
- `git add --all` ou `git add -A` → adiciona **todas as mudanças**
- `git status` → mostra o que está no staging
- `git restore --staged <arquivo>` → remove arquivo do staging

Exemplo: Adicionar um Arquivo ao Staging

```
git add index.html
```

Agora o arquivo está no **staging**. Para verificar:

```
git status
```

Saída:

```
On branch master
No commits yet

Changes to be committed:
  (use "git restore --staged ..." to unstage)
    new file: index.html
```

1.6.2 Adicionar Vários Arquivos ao Mesmo Tempo

```
git add --all
```

ou

```
git add -A
```


1.6.3 Verificar Arquivos no Staging

```
git status
```

Saída:

```
On branch master
No commits yet

Changes to be committed:
  (use "git restore --staged ..." to unstage)
    new file:   README.md
    new file:   bluestyle.css
    new file:   index.html
```

1.6.4 Remover Arquivo do Staging (Unstage)

Se você adicionou um arquivo por engano:

```
git restore --staged index.html
```

ou

```
git reset HEAD index.html
```

Problemas Comuns

- **Adicionei o arquivo errado** → use `git restore --staged <arquivo>`
- **Esqueci de adicionar um arquivo** → rode `git add <arquivo>` antes do commit
- **Não sei o que está staged** → rode `git status` para conferir

1.7 Commit

Um **commit** é como um ponto de salvamento no seu projeto. Ele registra uma **versão (snapshot)** dos seus arquivos em um determinado momento, junto com uma **mensagem descrevendo o que mudou**. Você sempre pode voltar para um commit anterior se precisar.

1.7.1 Comandos principais para commits

- `git commit -m "mensagem"` → Cria um commit com as mudanças que já estão na **staging area**.
- `git commit -a -m "mensagem"` → Cria um commit com **todas as mudanças rastreadas** (pulando a etapa de staging).
- `git log` → Exibe o histórico de commits.

1.7.2 Commit com Mensagem (-m)

Para salvar as alterações que já foram adicionadas à staging area:

```
git commit -m "Primeira versão do Hello World!"
```

Exemplo de saída:

```
[master (root-commit) 221ec6e] Primeira versão do Hello World!
 3 files changed, 26 insertions(+)
 create mode 100644 README.md
 create mode 100644 bluestyle.css
 create mode 100644 index.html
```

Sempre escreva mensagens claras, para que você (e outros) entendam o que foi alterado.

1.7.3 Commit de Todas as Alterações (-a)

Você pode pular a etapa de staging para arquivos já **rastreados** com:

```
git commit -a -m "Atualização rápida no README"
```

Exemplo de saída:

```
[master 123abcd] Atualização rápida no README
 1 file changed, 2 insertions(+)
```

Atenção:

- Esse comando **não inclui novos arquivos (untracked)**.
- Para novos arquivos, é necessário antes usar `git add <arquivo>`.

1.7.4 Tentando Commitar um Arquivo Novo com -a

Se você tentar commitar um arquivo novo com `-a`, verá algo assim:

```
$ git commit -a -m "Testando novo arquivo"
On branch master

No commits yet

Untracked files:
  (use "git add ..." to include in what will be committed)
        index.html

nothing added to commit but untracked files present (use "git add"
to track)
```


1.7.5 Mensagens de Commit em Múltiplas Linhas

Se você rodar apenas `git commit` (sem `-m`), o editor padrão será aberto, permitindo escrever uma mensagem mais detalhada.

Boa prática:

- Primeira linha curta (≤ 50 caracteres).
- Modo imperativo (“Adicionar recurso”, e não “Adicionado recurso”).
- Linha em branco após o resumo.
- Explicar **por que** a mudança foi feita, não apenas **o que** mudou.

1.7.6 Outras opções úteis de commit

- Criar um commit vazio:

```
git commit --allow-empty -m "Início do projeto"
```

- Reusar a mensagem anterior (sem editor):

```
git commit --no-edit
```

- Adicionar mudanças ao último commit, sem alterar a mensagem:

```
git commit --amend --no-edit
```

1.7.7 Erros comuns e como corrigir

- **Esqueceu de adicionar um arquivo?**

  Use `git add <arquivo>` e depois `git commit --amend`.

- **Erro de digitação na mensagem?**
  Use:

```
git commit --amend -m "Mensagem corrigida"
```


- **Commit errado?**
  Para desfazer o último commit mas manter as alterações na staging area.

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

1.7.8 Visualizar o Histórico de Commits

Exibir todo o histórico:

```
git log
```

Exemplo de saída:

```
commit 09f4acd3f8836b7f6fc44ad9e012f82faf861803 (HEAD -> master)
Author: w3schools-test
Date:   Fri Mar 26 09:35:54 2021 +0100

    Atualizado index.html com uma nova linha

commit 221ec6e10aeedbfd02b85264087cd9adc18e4b26
Author: w3schools-test
```

```
Date:   Fri Mar 26 09:13:07 2021 +0100

    Primeira versão do Hello World!
```

Visualização curta (um commit por linha):

```
git log --oneline
```

```
09f4acd Atualizado index.html com uma nova linha
221ec6e primeira versão do Hello World!
```

Ver estatísticas de alterações em cada commit:

```
git log --stat
```

1.8 Git Tagging (Tagueamento no Git)

Principais Comandos para Tags

- `git tag <tagname>` → Cria uma tag *leve* (lightweight)
- `git tag -a <tagname> -m "mensagem"` → Cria uma tag *anotada* (annotated)
- `git tag <tagname> <commit-hash>` → Marca um commit específico
- `git tag` → Lista as tags
- `git show <tagname>` → Mostra detalhes da tag

Uma **tag** no Git é como um **marcador ou rótulo** associado a um commit específico. Geralmente, é usada para **marcar versões importantes do projeto**, como releases (`v1.0`, `v2.0`). As tags oferecem uma forma **simples e confiável** de acompanhar versões e compartilhá-las com sua equipe ou usuários.

Exemplos de uso:

- **Releases** → marcar quando o projeto está pronto para uma versão oficial.
- **Marcos (milestones)** → destacar funcionalidades concluídas ou correções críticas.
- **Deployments** → ferramentas de deploy usam tags para saber qual versão liberar.
- **Hotfixes** → corrigir versões antigas de forma organizada.

1.8.1 Criando Tags

Criar uma Tag Leve (Lightweight)

Uma **tag leve** é apenas um nome apontado para um commit (sem metadados extras).

```
git tag v1.0
```

Tag Anotada (Annotated)

Uma **tag anotada** armazena **autor, data e mensagem**. **Recomendada** para versões públicas e releases.

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

Tag em um Commit Específico

Você pode marcar um commit antigo informando seu **hash**:

```
git tag v1.1 1a2b3c4d
```

Listar e Ver Tags

- **Listar todas as tags**:

  ```
  git tag
  ```

- **Ver detalhes de uma tag**:

  ```
  git show v1.0
  ```


1.8.2 Enviar Tags para o Repositório Remoto

O comando `git push` **não envia tags automaticamente**! É preciso empurrá-las manualmente.

- **Enviar uma tag específica**:

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

- **Enviar todas as tags de uma vez**:

  ```
  git push --tags
  ```

1.8.3 Excluir Tags

- **Excluir localmente**:

```
git tag -d v1.0
```

- **Excluir no repositório remoto**:

```
git push origin --delete tag v1.0
```

1.8.4 Atualizar ou Substituir uma Tag

Se precisar mover uma tag para outro commit:

```
git tag -f v1.0 <novo-commit-hash>
git push --force origin v1.0
```

sobrescrever tags afeta todos os usuários do repositório.

Boas Práticas no Uso de Tags

- Use tags para releases, marcos importantes e versões estáveis.
- Prefira **tags anotadas** para compartilhar publicamente.
- Crie tags **após passar nos testes** ou antes de um **deploy**.

Troubleshooting (Resolvendo Problemas)

- **Tag já existe?**

  Apague com `git tag -d <tagname>` e recrie.

- **Pushed a tag errada?**

  Delete local e remoto, depois crie a correta.

- **Tag não aparece no remoto?**

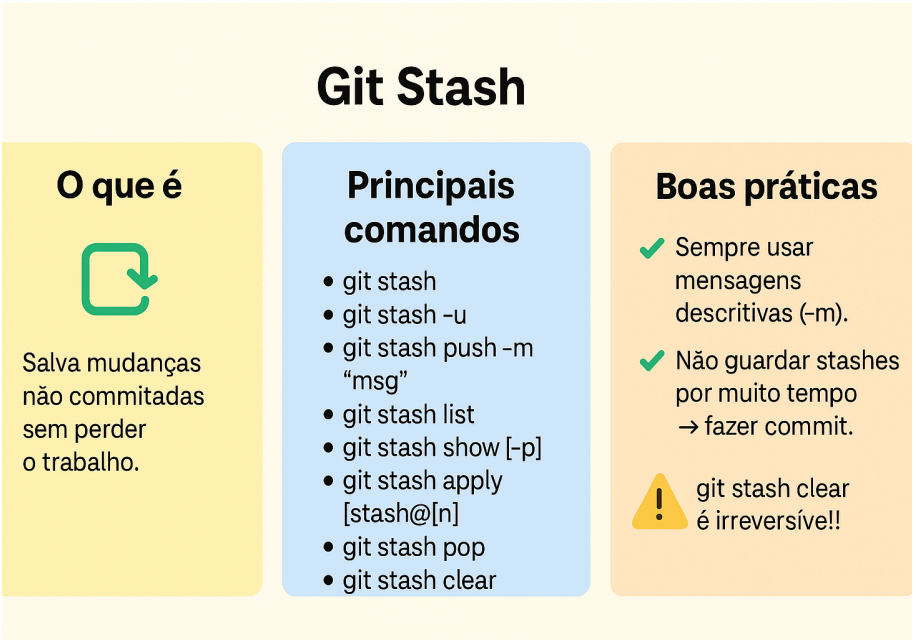
  Lembre-se de usar `git push origin <tag>` ou `git push --tags`.

- **Precisa sobrescrever uma tag no remoto?**

  Use `git push --force origin <tag>` (com cautela).

1.9 Git Stash

Git Stash, salva mudanças não commitadas sem perder o trabalho. E também permite mudar de branch, corrigir bugs urgentes ou guardar WIP.



Principais Comandos

Comando	Função
`git stash`	Salva mudanças atuais (tracked).
`git stash -u`	Salva mudanças incluindo arquivos não rastreados.
`git stash push -m "msg"`	Stash com mensagem.
`git stash list`	Lista todos os stashes.
`git stash show [-p]`	Mostrar resumo/diferenças.
`git stash apply [stash@{n}]`	Aplica stash (mantém no stack).
`git stash pop`	Aplica e remove o stack.
`git stash drop stash@{n}`	Remove stash específico.
`git stash clear`	Limpa todos os stashes.
`git stash branch <branch>`	Cria branch a partir de um stash.

- **Stack de stashes**: último em cima (LIFO).
- **Stash ≠ commit**: uso temporário.
- **Mensagens claras** ajudam na organização.

Boas Práticas

- Sempre usar mensagens descritivas (`-m`).
- Não guarda stashes por muito tempo → fazer commit.
- Revisar com `git stash list` e limpar os antigos.
- `git stash clear` é irreversível!t

1.10 Git History

O Git mantém um registro detalhado de todas as mudanças feitas no seu projeto. Com os comandos de histórico, você pode ver **o que mudou, quando mudou e quem fez a mudança**. Isso é útil para:

- Acompanhar o progresso do projeto.
- Localizar bugs.
- Entender a evolução do código.

Principais Comandos para Visualizar o Histórico

- **`git log`** → Mostra todo o histórico de commits.
- **`git log --oneline`** → Mostra um resumo dos commits.
- **`git show <commit>`** → Mostra detalhes de um commit específico
- **`git diff`** → Mostra mudanças não preparadas (unstaged).
- **`git diff --staged`** → Mostra mudanças que já foram preparadas (staged).

Boas Práticas para Visualizar o Histórico

1. Faça commits frequentes e significativos para manter o histórico claro.
2. Escreva mensagens de commit descritivas para você e sua equipe entenderem melhor as mudanças.
3. Use `git log --oneline` para uma visão rápida.
4. `git diff` antes de confirmar (commit) para revisar suas alterações.

1.10.1 Ver Histórico

Ver Histórico Completo de Commits (git log)

```
git log
```

Saída (exemplo):

```
commit 09f4acd3f8836b7f6fc44ad9e012f82faf861803 (HEAD -> master)
Author: w3schools-test
Date:   Fri Mar 26 09:35:54 2021 +0100

    Updated index.html with a new line
```

- Mostra autor, data e mensagem de cada commit.

- Use as **setas** para navegar e pressione “**q**” para sair.
- Pesquise dentro do log usando `/palavra` (ex.: `/fix`).

Ver Detalhes de um Commit (`git show <commit>`)

```
git show 09f4acd
```

Saída (exemplo):

```
commit 09f4acd3f8836b7f6fc44ad9e012f82faf861803
Author: w3schools-test
Date:   Fri Mar 26 09:35:54 2021 +0100

    Updated index.html with a new line

diff --git a/index.html b/index.html
--- a/index.html
+++ b/index.html
@@ ...
+New Title
```


- Mostra autor, data, mensagem e alterações feitas.

1.10.2 Comparar Mudanças

Comparar Mudanças Não Preparadas (`git diff`)

```
git diff
```

- Mostra as diferenças entre o diretório de trabalho e o último commit.

Comparar Mudanças Preparadas (`git diff --staged`)

```
git diff --staged
```

- Mostra diferenças entre os arquivos preparados (staged) e o último commit.

Comparar Dois Commits

```
git diff <commit1> <commit2>
```

Resumo dos Commits (`git log --oneline`)

```
git log --oneline
```

Saída (exemplo):

```
09f4acd Updated index.html with a new line
8e7b2c1 Add about page
1a2b3c4 Initial commit
```

1.10.3 Mostrar Commits

Mostrar Commits de um Autor

```
git log --author="Alice"
```

Mostrar Commits Recentes

```
git log --since="2 weeks ago"
```

Mostrar Arquivos Alterados por Commit

```
git log --stat
```

Mostrar Gráfico de Branches

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


Exemplo de saída:

```
* 09f4acd Updated index.html with a new line
* 8e7b2c1 Add about page
|\
| * aabbccd Merge branch 'feature-x'
|/
```

Dicas e Solução de Problemas

- **Não vejo minhas mudanças no histórico**

  Certifique-se de ter feito o commit. Mudanças não confirmadas não aparecem no log.

- **O log está muito longo**

  Use `git log --oneline` ou `git log --since`.

- **Como sair da visualização do log ou diff?**

  Pressione `q`.

1.11 Git Branch

No **Git**, um *branch* é como um espaço de trabalho separado onde você pode fazer alterações e testar ideias sem impactar o projeto principal. Pense nele como um **"universo paralelo"** para o seu código.

Os branches permitem trabalhar em diferentes partes do projeto sem atrapalhar a **branch principal (master/main)**.

Exemplos de uso:

- Desenvolvimento de uma nova **feature**
- Correção de um **bug**
- **Experimentação** de ideias

Exemplo:

Sem Git

- Fazer cópias dos arquivos para não impactar a versão principal
- Alterar os arquivos e, em caso de dependências, copiar mais arquivos
- Se surge um erro urgente, salvar tudo, corrigir e depois voltar ao trabalho anterior
- Risco de inconsistências (erros não aplicados em versões futuras)

Com Git

- Criar um branch **new-design** e trabalhar sem afetar a principa
- Se surgir um erro, criar um branch **small-error-fix** a partir da main
- Corrigir, mesclar e voltar ao branch anterior
- No final, mesclar **new-design** → main.

1.11.1 Principais Comandos de Branch

Criar um branch

```
git branch hello-world-images
```

Listar branches

```
git branch
```

O branch atual aparece com `*`.

Trocar de branch

```
git checkout hello-world-images
```

ou

```
git switch hello-world-images
```

Criar e trocar em um só comando

```
git checkout -b novo-branch
```

Verificar branch atual

```
git status
```

Deletar branch

```
git branch -d hello-world-images
```

Se o branch não foi mesclado, usar:

```
git branch -D hello-world-images
```

1.11.2 Fluxo de Trabalho no Branch

1. Criar um novo branch para sua tarefa
2. Fazer alterações e salvar os arquivos
3. Adicionar ao **staging**

   ```
   git add --all
   ```

4. Realizar o **commit**

   ```
   git commit -m "Mensagem clara"
   ```

5. Ao finalizar, **mesclar com a main**

Exemplo de Branch de Emergência

Criar e mudar para um branch de correção urgente:

```
git checkout -b emergency-fix
```

Corrigir, commitar e depois mesclar na **main**.

1.11.3 Boas Práticas

- Nomear branches de forma clara e descritiva:
  - `feature/login-page`
  - `bugfix/header-crash`

- Manter **um objetivo por branch**
- Sincronizar regularmente com a **main**
- Excluir branches antigos para manter o repositório limpo

Exemplos Úteis

- Renomear branch:

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

- Ver todos os branches:

```
git branch
```

- Trocar de branch:

```
git switch nome-branch
```

- Forçar exclusão:

```
git branch -D nome-branch
```

Problemas Comuns

- **Alterações não aparecem na main** → Lembre-se: mudanças ficam no branch até serem mescladas.
- **Não consegue excluir branch** → Ele ainda não foi mesclado; use `-D` se tiver certeza.

1.12 Git Branch Merge

Merging significa **combinar as mudanças de uma branch em outra**. É assim que integramos o trabalho feito separadamente em features ou correções de bugs.

1.12.1 Principais Comandos de Merge

- `git merge <branch>` → Mescla a branch no branch atual
- `git merge --no-ff` → Sempre cria um commit de merge
- `git merge --squash` → Une todas as alterações em **um único commit**
- `git merge --abort` → Cancela um merge em andamento

Exemplo Básico: Fazendo Merge

1. Primeiro, vá para o branch de destino (ex.: `master`):

```
git checkout master
```

2. Depois, mescle a outra branch (ex.: `emergency-fix`)

```
git merge emergency-fix
```

Se não houver alterações conflitantes, o Git pode fazer um **fast-forward merge**, apenas movendo o ponteiro do branch.

1.12.2 Boas Práticas para Merging

- Sempre **commit ou stash** suas mudanças antes de mesclar.
- Traga atualizações regularmente do branch principal (`main`/`master`) para minimizar conflitos.
- Resolva conflitos com atenção — **não aceite tudo cegamente**.
- Escreva mensagens de merge claras e descritivas.

Exemplos Práticos

- **Cancelar merge em andamento:**

```
git merge --abort
```

- **Checar status durante merge:**

```
git status
```


- **Resolver conflitos:**
  - Edite o(s) arquivo(s) marcados com `<<<<<<<`, `=======`, `>>>>>>>`

  Depois use:

```
git add arquivo
git commit
```

1.12.3 Tipos de Merge

- **Fast-forward merge** → Nenhum commit divergente, apenas move o ponteiro.
- **No fast-forward merge** → Garante um commit de merge (histórico mais claro).

```
git merge --no-ff feature-branch
```

- **Squash merge** → Une todos os commits em **um único commit**.

```
git merge --squash feature-branch
```

1.12.4 Conflitos de Merge

Ocorrem quando duas branches modificam a **mesma parte de um arquivo**. O Git insere marcadores como:

```
<<<<<<< HEAD
Versão do branch atual
=======
Versão do branch a ser mesclado
>>>>>>> outro-branch
```

O usuário deve editar e decidir o que manter. Depois:

```
git add arquivo
git commit -m "merge resolvido"
```

1.12.5 Excluir Branch após Merge

Depois de concluir a integração:

```
git branch -d nome-da-branch
```

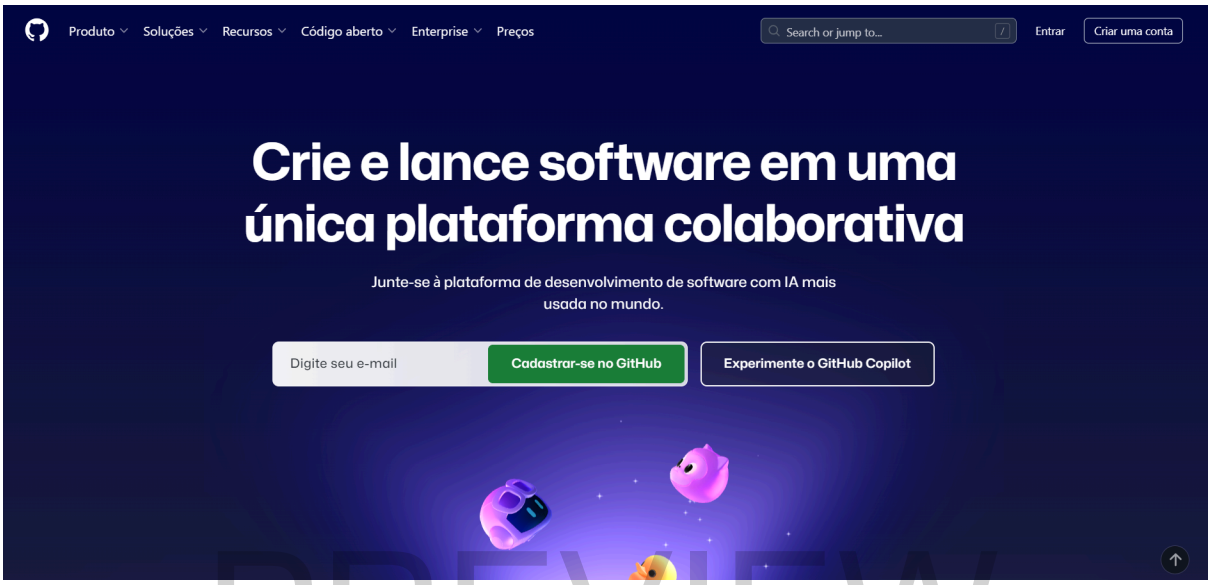

Dica: Use `git log --graph --oneline` para visualizar a árvore de merges.

2 Git e GitHub

2.1 Início

2.1.1 Criando sua conta no GitHub

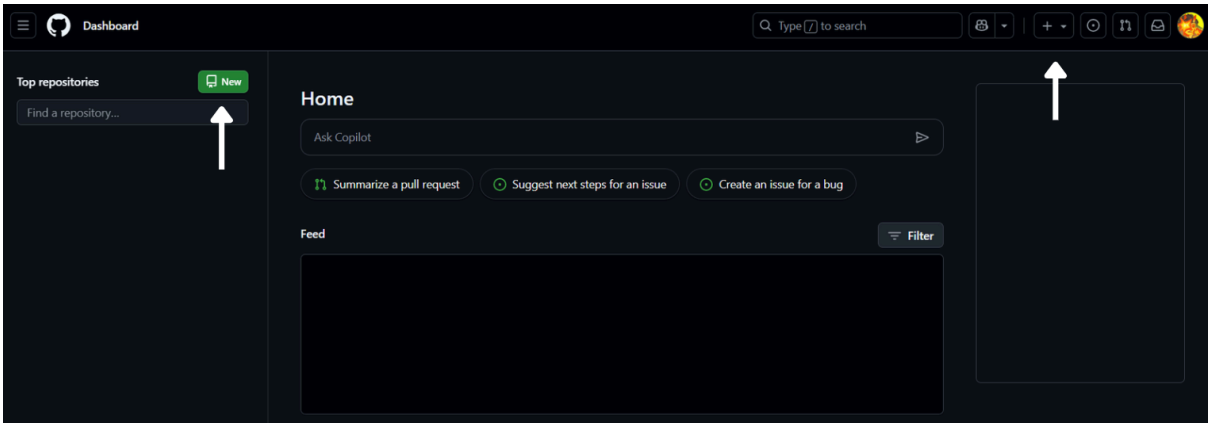
Vá para o pagina inicial do GitHub (https://github.com) e crie sua conta de forma gratuita:



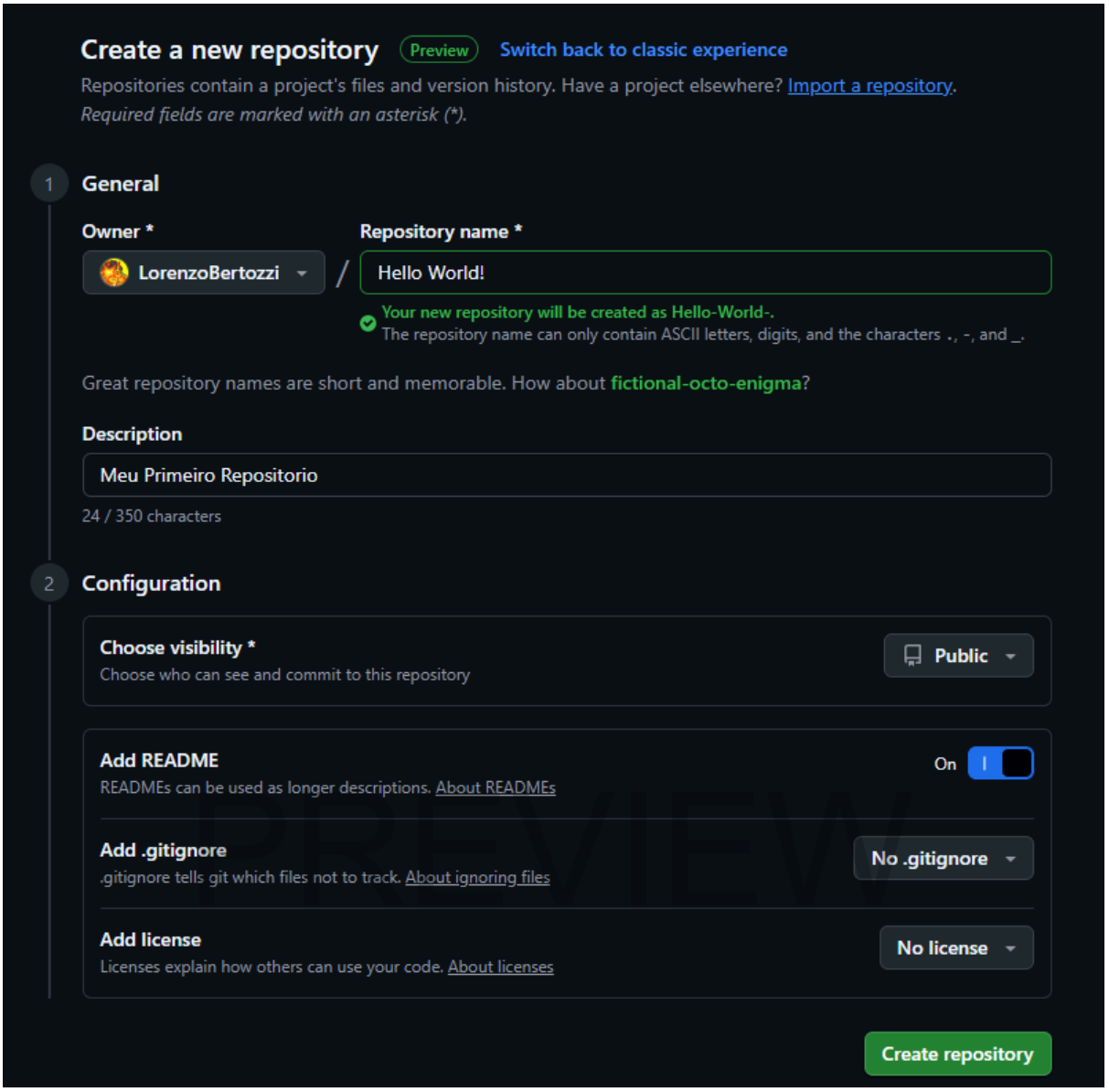
- Use o mesmo endereço de e-mail que você planeja usar para sua configuração do Git.

2.1.2 Criar Um Repositório

Após cadastrar-se, clique no botão “Create New” na parte superior direita da tela para criar um novo repositório.



Preencha os detalhes do repositório (nome, descrição, público/privado, etc.) e clique em Criar repositório:



Um repositório é uma versão do seu projeto hospedada na internet. O GitHub é uma plataforma popular para hospedar repositórios remotos, permitindo que você colabore com outras pessoas e faça backup do seu código.

2.2 Git Security com SSH

SSH (*Secure Shell*) é uma forma segura de se conectar a computadores e serviços remotos — como repositórios Git. Ele usa um **par de chaves (pública e privada)** para garantir que apenas você consiga acessar seu código.

Conceitos e Comandos de SSH

- `SSH key pair` → par de chaves (pública + privada) para acesso seguro
- `ssh-keygen` → gera um novo par de chaves
- `ssh-add` → adiciona a chave privada ao agente SSH

- `ssh -T git@github.com` → testa a conexão SSH com o GitHub
- `ssh-add -l` → lista chaves carregadas no agente
- `ssh-add -d` → remove uma chave do agente

Como funcionam as Chaves SSH?

- As chaves vêm em **pares**:
  - **Chave pública** → funciona como um cadeado
  - **Chave privada** → funciona como a chave

Você compartilha a chave **pública** com o servidor (GitHub, GitLab, Bitbucket), mas mantém a chave **privada** protegida no seu computador. Somente quem tiver a **chave privada** consegue acessar o que foi bloqueado pela pública.

2.2.1 Configuração do SSH

Primeira Configuração do Agente SSH

Ative o agente SSH no seu sistema:

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


Gerando um Par de Chaves SSH

Crie um novo par de chaves no terminal (Linux, macOS ou Git Bash no Windows):

```
ssh-keygen -t rsa -b 4096 -C "seu@email.com"
```

- Pressione **Enter** para salvar no local padrão
- Defina uma **passphrase** (opcional, mas recomendado para segurança extra)

Adicionando a Chave ao Agente

Após gerar a chave, adicione-a ao agente:

```
ssh-add ~/.ssh/id_rsa
```

Copiando sua Chave Pública

Você deve copiar a chave **pública** para configurá-la no GitHub/GitLab/Bitbucket.

- **macOS:**

  ```
  pbcopy < ~/.ssh/id_rsa.pub
  ```

- **Windows (Git Bash):**

```
clip < ~/.ssh/id_rsa.pub
```

- **Linux:**

```
cat ~/.ssh/id_rsa.pub
```

    - *(copie manualmente o conteúdo exibido)*

Listando e Removendo Chaves

- **Listar chaves carregadas:**

```
ssh-add -l
```

- **Remover uma chave do agente:**

```
ssh-add -d ~/.ssh/id_rsa
```

2.2.2 Resolvendo Problemas (Troubleshooting)

- **Erro "Permission denied"**
    1. Verifique se a chave pública foi adicionada ao GitHub/GitLab/Bitbucket.
    2. Confira se a chave privada está carregada no agente.
- **Permissões do arquivo**

    A chave primária deve ser acessível apenas por você:

```
chmod 600 ~/.ssh/id_rsa
```

- **Debug detalhado**

    Use saída verbosa:

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

- **URL correta**

    Verifique se o repositório remoto usa **SSH**:

```
git@github.com:usuario/repositorio.git
```

Boas Práticas de Segurança

- Nunca compartilhe sua **chave privada**.
- Use uma **passphrase** para proteger sua chave.

- Se sua chave privada for comprometida → **gere um novo par** e atualize no servidor.

2.3 Adicionando Chave SSH

Agora que você já gerou sua chave SSH, o próximo passo é **adicionar sua chave pública à sua conta do GitHub**.

2.3.1 Copiando sua Chave Pública

Dependendo do seu sistema operacional:

Windows (Git Bash):

```
clip < ~/.ssh/id_rsa.pub
```

macOS:

```
pbcopy < ~/.ssh/id_rsa.pub
```

Linux:

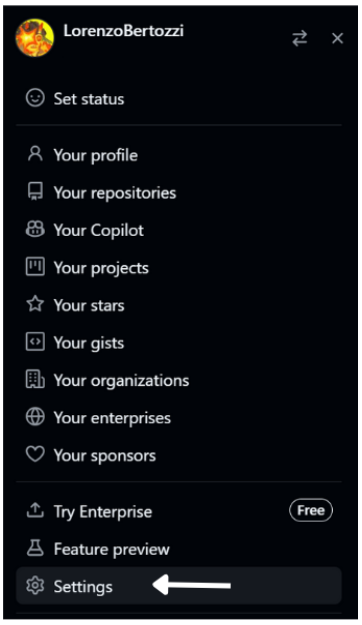
```
cat ~/.ssh/id_rsa.pub
```


- (copie manualmente o texto exibido)

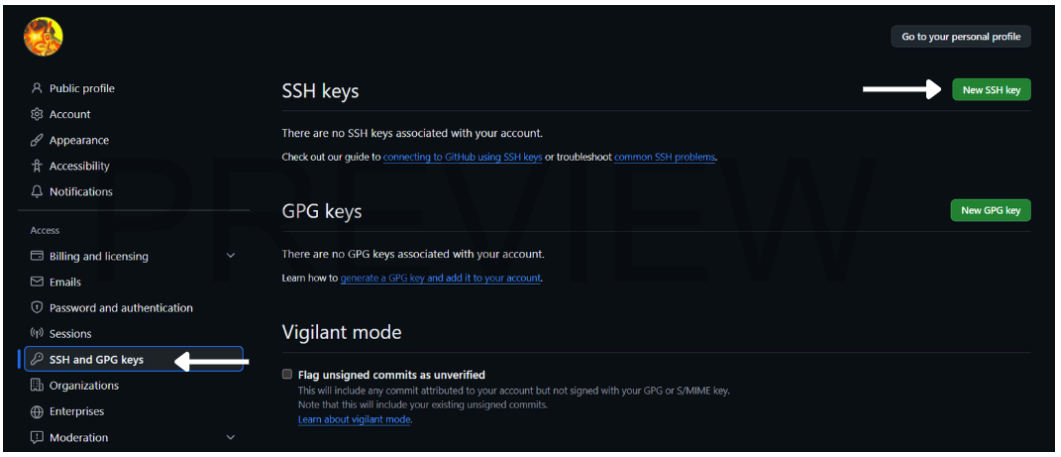
Importante: Esses passos devem ser feitos **antes** de tentar usar comandos como `git push`, `git pull` ou `git clone` com SSH. Este guia **pressupõe que você já tenha gerado sua chave SSH e a adicionado ao agente**. Se ainda não fez isso, volte à seção de **Configuração SSH**.

2.3.2 Adicionando a Chave ao GitHub

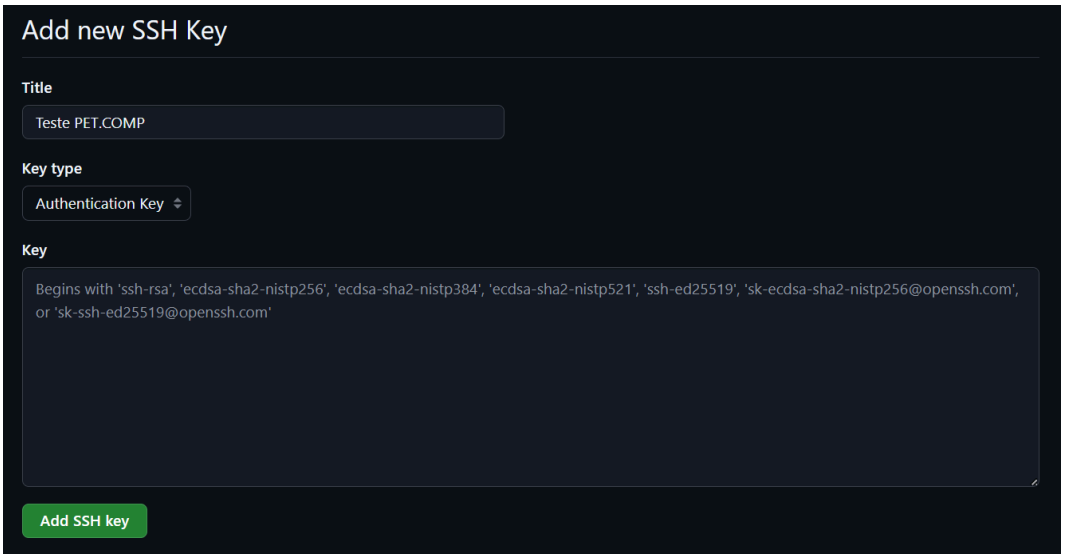
1. Acesse o GitHub, clique no seu perfil no canto superior direito e selecione Configurações:



2. Na barra lateral, selecione as chaves SSH e GPG e clique no botão Nova chave SSH.



3. Dê um título descritivo à sua chave, cole sua chave SSH pública no campo "Chave" e clique em Adicionar chave SSH.



Você poderá ser solicitado a fornecer sua senha do GitHub ou usar a autenticação de dois fatores para confirmar a adição. Em seguida, você testará sua conexão SSH e definirá sua origem remota do GitHub usando SSH.

2.4 Configurando o Remote Origin do GitHub com SSH

Agora que sua chave SSH já foi adicionada ao GitHub, você pode **conectar com segurança** seu repositório local ao GitHub usando **SSH**.

2.4.1 Testando sua Conexão SSH

Antes de configurar o repositório, teste se sua conexão SSH com o GitHub está funcionando:

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

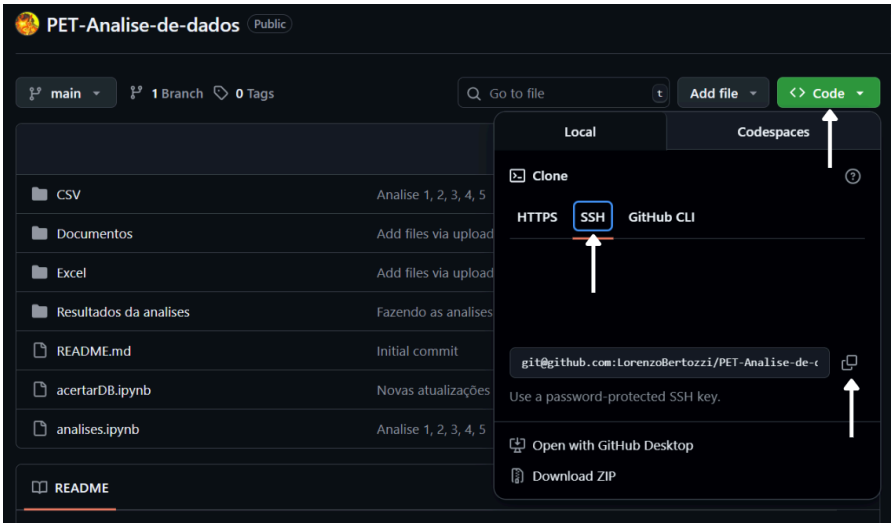
Exemplo de saída no terminal:

```
The authenticity of host 'github.com (140.82.121.3)' can't be established.
RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'github.com,140.82.121.3' (RSA) to the list of known hosts.
Hi seu-usuario! You've successfully authenticated, but GitHub does not provide shell access.
```


Se a última linha mostrar o seu **nome de usuário do GitHub**, a autenticação SSH foi concluída com sucesso!

2.4.2 Obtendo o Endereço SSH do Repositório

1. No GitHub, acesse o repositório desejado.
2. Clique no botão **Code** (verde).
3. Selecione a opção **SSH**.
4. Copie o link exibido (ele sempre começa com `git@github.com:`).



Exemplo:

```
git@github.com:seu-usuario/seu-repo.git
```

2.4.3 Adicionando ou Atualizando o Remote Origin

1. **Primeira configuração do remote origin:**

   ```
   git remote add origin
   git@github.com:seu-usuario/seu-repo.git
   ```

2. **Se já existir um remote configurado e você quiser atualizar para SSH:**

   ```
   git remote set-url origin
   git@github.com:seu-usuario/seu-repo.git
   ```

Agora seu repositório local está conectado ao GitHub usando **SSH**. Você já pode realizar operações como:

- `git push origin main` → enviar código para o GitHub
- `git pull origin main` → atualizar com alterações remotas
- `git clone` → clonar repositórios usando SSH

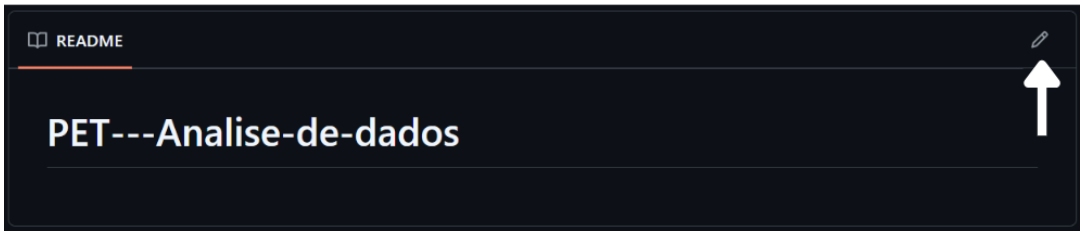
2.5 Editando Código Diretamente no GitHub

O GitHub permite **editar arquivos direto no navegador**, sem precisar usar o Git no computador.

Essa funcionalidade é útil para **pequenas alterações rápidas**, como ajustar o `README.md` ou corrigir erros de texto.

2.5.1 Editando um Arquivo

1. Acesse o repositório no GitHub.
2. Clique no nome do arquivo (ex.: `README.md`).
3. Clique no botão ✏️ **Edit (lápis)**.
4. O editor será aberto e você poderá alterar o conteúdo do arquivo.

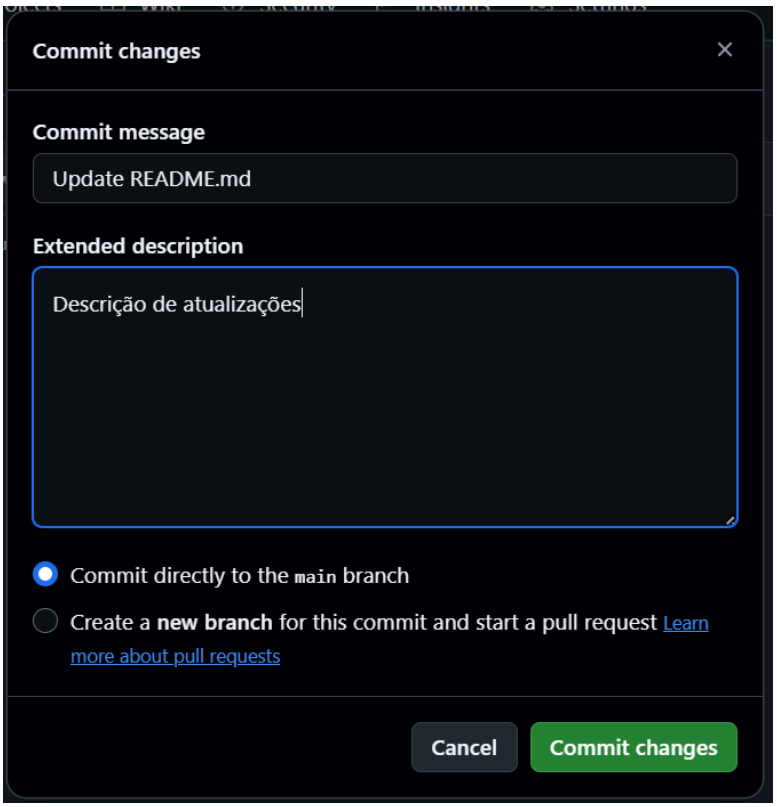


Você pode editar **qualquer arquivo**, não apenas o `README.md`.

2.5.2 Escrevendo a Mensagem do Commit

Após editar:

1. Role até a seção **Commit changes**.
2. Escreva uma **mensagem clara e curta** descrevendo a mudança.



Isso ajuda outros colaboradores a entender suas atualizações.

Visualizando Mudanças

Antes de salvar, clique em **Preview changes**. Assim, você pode comparar o arquivo **antes e depois** da edição.

2.5.3 Salvando as Alterações (Commit)

- Por padrão, o GitHub salva direto na branch principal (`main` ou `master`).
- Para mudanças maiores, selecione:

```
Create a new branch for this commit and start a pull request
```

Isso garante mais segurança e organização.

2.5.4 Criando uma Nova Branch pelo Editor

- Ao escolher criar uma branch, o GitHub sugere automaticamente um nome.
- Depois de confirmar o commit, você pode abrir um **Pull Request** para propor suas mudanças.

Agora você já sabe como **editar arquivos diretamente pelo GitHub** de forma rápida e segura.

2.6 Git Pull do GitHub

Nos capítulos anteriores, criamos uma conta no GitHub e configuramos o acesso via **SSH**. Também fizemos algumas alterações diretamente no GitHub.

Agora, queremos atualizar nosso repositório local com as mudanças feitas no GitHub.

Quando trabalhamos em equipe em um projeto, é importante que todos mantenham seus repositórios atualizados. Sempre que você começar a trabalhar em um projeto, deve trazer as alterações mais recentes para sua cópia local. No Git, isso pode ser feito com o comando **pull**.

O `pull` é uma combinação de **dois comandos diferentes**:

- `fetch`
- `merge`

Vamos analisar cada um deles.

2.6.1 Git Fetch

O comando **git fetch** baixa novos dados de um repositório remoto, mas **não altera seus arquivos locais nem seus branches**.

Ele apenas mostra o que foi atualizado no servidor antes de você decidir incorporar essas mudanças.

Exemplo:

```
git fetch origin
```

Saída (exemplo):

```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 733 bytes | 3.00 KiB/s, done.
From https://github.com/w3schools-test/hello-world
   e0b6038..d29d69f  master     -> origin/master
```

Agora podemos verificar o status:


```
git status
```

Saída:

```
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be
fast-forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
```

Nosso branch local está **1 commit atrás do remoto**.

Verificando os Commits e Diferenças

Podemos visualizar os commits mais recentes do repositório remoto:

```
git log origin/master
```

Ou verificar diretamente as diferenças entre o branch local e o remoto:

```
git diff origin/master
```

Isso garante que sabemos exatamente quais mudanças serão aplicadas.

2.6.2 Git Merge

O comando **merge** combina o branch atual com outro branch. No nosso caso, podemos mesclar o `master` local com o `origin/master` remoto:

```
git merge origin/master
```

Saída (exemplo):

```
Updating e0b6038..d29d69f
Fast-forward
 README.md | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)
```

Depois, podemos confirmar que estamos atualizados:

```
git status
```

Saída:


```
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

Agora nosso repositório local está sincronizado!

2.6.3 Git Pull

Se você quiser simplificar e atualizar seu repositório local **em apenas um passo**, use o comando **git pull**.

Esse comando é equivalente a:

```
git fetch + git merge
```

Ele baixa todas as alterações do repositório remoto e as mescla no branch atual.

Exemplo:

```
git pull origin
```

Saída:

```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 794 bytes | 1024 bytes/s, done.
From https://github.com/w3schools-test/hello-world
    a7cdd4b..ab6b4ed  master       -> origin/master
Updating a7cdd4b..ab6b4ed
Fast-forward
  README.md | 2 ++
  1 file changed, 2 insertions(+)
```

É assim que mantemos o **Git local sempre atualizado** com o repositório remoto.

No próximo capítulo, veremos como funciona o **push** no GitHub.

2.7 Git Push para o GitHub

Quando fazemos alterações localmente, precisamos atualizar o repositório remoto com essas mudanças. A transferência das alterações locais para o repositório remoto é feita com o comando **push**. Existem diversos comandos que podemos utilizar para enviar mudanças para o GitHub.

Comandos principais de Push

- **Basic Push** (Envio básico)
- **Force Push** (Envio forçado)
- **Push Tags** (Envio de tags)
- **Troubleshooting** (Resolução de problemas)

2.7.1 Basic Push

Esse comando envia o branch atual para o repositório remoto chamado **origin**:

Exemplo

```
git push origin
```

Isso irá enviar seus commits locais para o GitHub.

Importante: você precisa já ter realizado o commit das mudanças com `git commit`.

2.7.2 Force Push

Se o seu push for rejeitado devido a atualizações não fast-forward (por exemplo, após um **rebase**), é possível forçar o envio.

Atenção: Isso pode sobrescrever alterações no repositório remoto. Use com cuidado!

Exemplo

```
git push --force origin feature-branch
```

Uma alternativa mais segura é utilizar `--force-with-lease`:

Exemplo

```
git push --force-with-lease origin feature-branch
```

2.7.3 Push Tags

Para enviar **todas as tags locais** para o GitHub:

Exemplo

```
git push --tags
```

Para enviar uma tag específica:

Exemplo

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

Troubleshooting (Resolvendo Problemas)

- **Erro Non-fast-forward:** acontece se alguém fez push para o mesmo branch antes de você. Rode `git pull --rebase` antes de tentar novamente.
- **Erro de autenticação:** confirme se você tem acesso ao repositório e se suas credenciais estão corretas.

Confirmação no GitHub

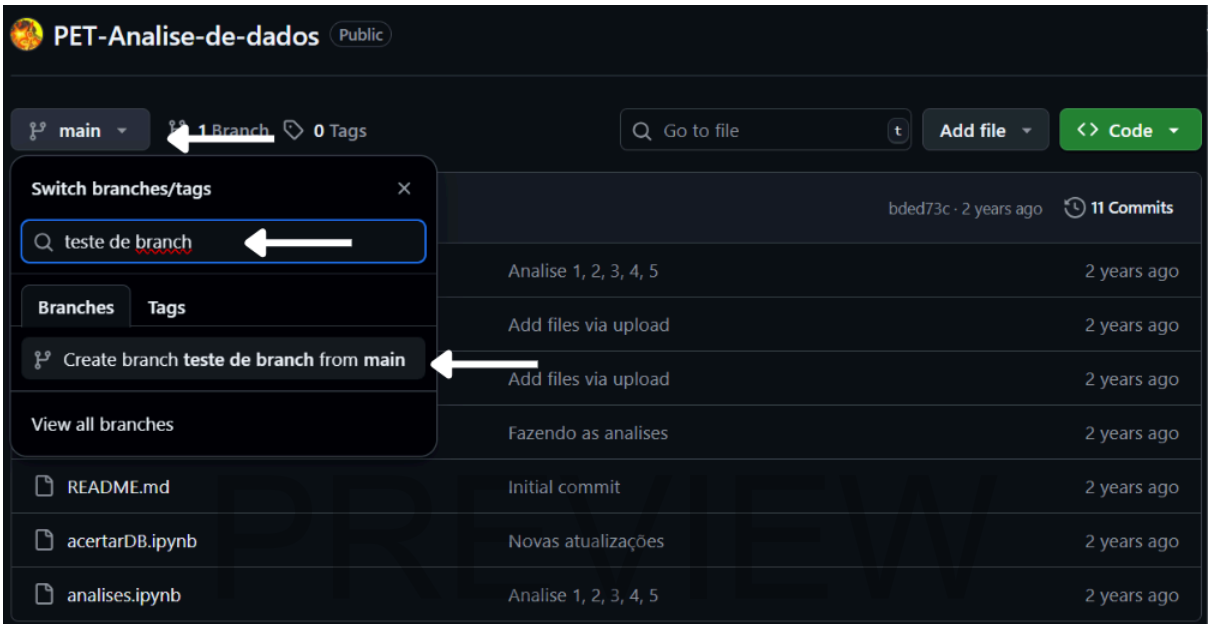
Após o push, acesse o GitHub e verifique se o repositório possui um novo commit.

No próximo capítulo, vamos começar a trabalhar com **branches no GitHub**.

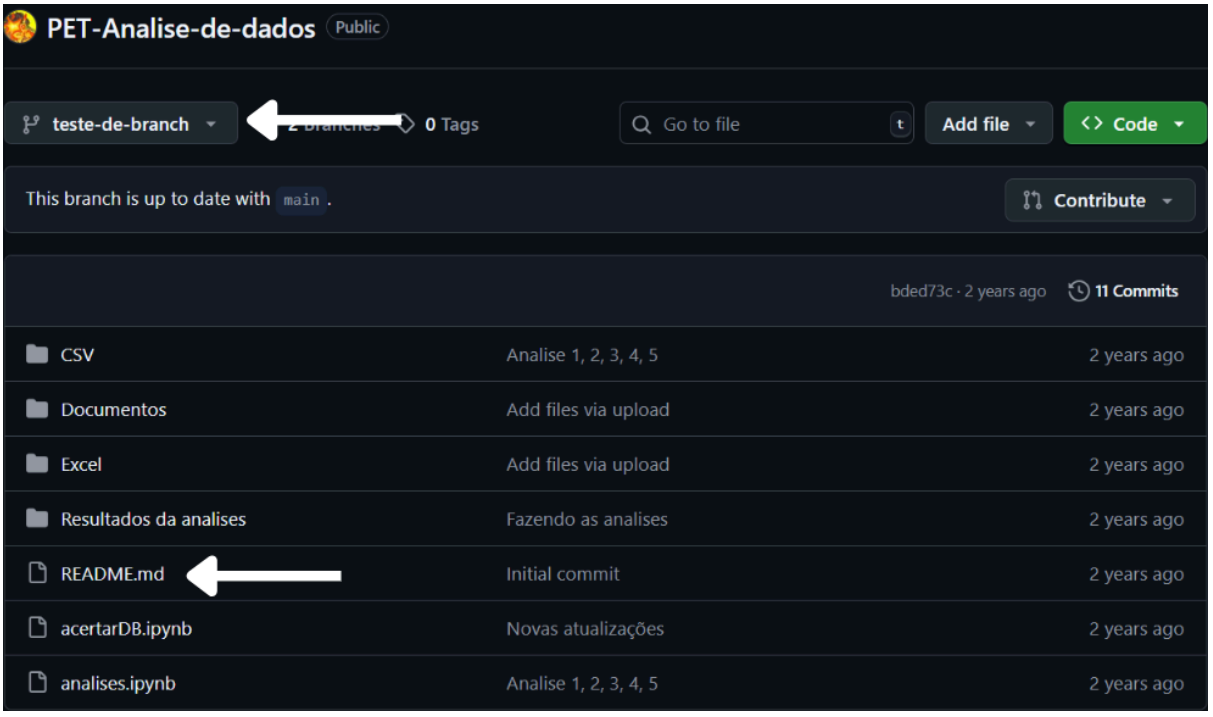
2.8 Branches no GitHub

Branches (ramos) permitem trabalhar em diferentes versões do código sem interferir diretamente no branch principal (geralmente o main). Com eles, podemos criar novas funcionalidades, corrigir erros ou testar ideias de forma isolada.

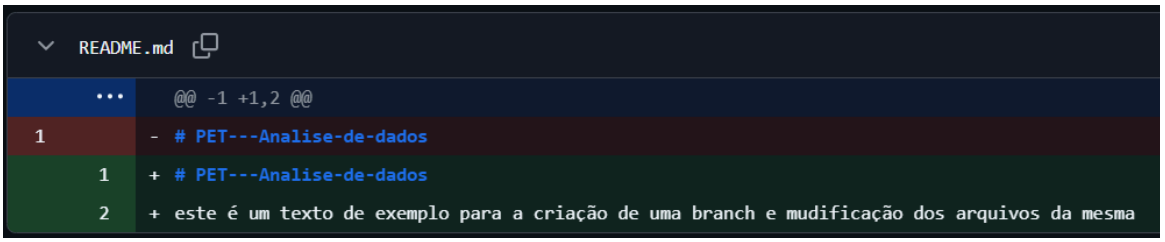
No GitHub, acesse seu repositório e clique no botão "Branch principal". Lá, você pode criar uma nova Branch. Digite um nome descritivo e clique em Create Branch:



A **Branch** agora deve estar criada e ativa. Você pode confirmar em qual Branch está trabalhando olhando o botão de Branch. Viu que agora diz "teste de branch" em vez de "main"?



Comece a trabalhar em um arquivo existente nesta branch. Clique no arquivo "README.md" e comece a editar. Depois de terminar de editar o arquivo, você pode clicar na aba "Visualizar alterações" para ver as alterações feitas destacadas:



Se estiver satisfeito com a alteração, adicione um comentário explicando o que você fez e clique em Confirmar alterações.

Agora você tem uma nova Branch no GitHub, atualizada com algumas alterações!

2.8.1 Switch Branch (Trocar de branch)

- **Na interface do GitHub:** Clique no menu suspenso de branches e selecione o branch que deseja usar.
- **Na linha de comando:**

```
git switch branch-name
```


2.8.2 Delete Branch (Deletar branch)

- **No GitHub:** Vá até a página de branches, encontre o branch e clique no ícone da lixeira.
- **Na linha de comando (local):**

```
git branch -d branch-name
```

- **Na linha de comando (remoto):**

```
git push origin --delete branch-name
```

2.8.3 Rename Branch (Renomear branch)

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

2.8.4 Merge Branch (Mesclar branches)

- **No GitHub:** Abra um **Pull Request (PR)** e siga as instruções para mesclar.
- **Na linha de comando:**

```
git merge branch-name
```

2.8.5 View Branches (Visualizar branches)

- **No GitHub:** Clique no menu suspenso de branches no topo da lista de arquivos.
- **Na linha de comando:**

```
git branch
```

2.8.6 Protected Branches (Branches protegidos)

Alguns branches (como `main`) podem ser **protegidos**, o que significa que:

- Não podem ser deletados.
- Não permitem force-push.
- Exigem revisão de código antes de merge.

Isso ajuda a evitar alterações acidentais em partes críticas do projeto.

2.9 Git Pull Branch from GitHub

Quando trabalhamos em equipe, outros desenvolvedores podem criar branches no repositório remoto (GitHub). Para manter o repositório local atualizado e acessar esses novos branches, usamos o **pull** e depois fazemos checkout do branch desejado.

2.9.1 Atualizar o repositório local

```
git pull
```

Isso baixa os commits e referências mais recentes do GitHub.

2.9.2 Verificar status

```
git status
```

Saída típica:

```
On branch master
Your branch is up to date with 'origin/master'.
nothing to commit, working tree clean
```

2.9.3 Listar branches locais e remotos

- **Locais apenas:**

```
git branch
```

- **Locais + Remotos:**

```
git branch -a
```

Saída típica:

```
* master
  remotes/origin/html-skeleton
  remotes/origin/master
```

O branch `html-skeleton` existe no remoto, mas ainda não localmente.

2.9.4 Trazendo o branch remoto para local

```
git checkout html-skeleton
```

Saída:

```
Switched to a new branch 'html-skeleton'
Branch 'html-skeleton' set up to track remote branch
'html-skeleton' from 'origin'.
```


Agora o branch remoto foi **checado** localmente e está rastreando (`tracking`) o GitHub.

2.9.5 Confirmar atualização

```
git pull
```

Se não houver mudanças novas:

```
Already up to date.
```

2.9.6 Conferir branches locais

```
git branch
```

Saída típica:

```
* html-skeleton
  master
```

Pronto! Agora você está **trabalhando no novo branch** (`html-skeleton`) que veio do GitHub. Você pode abrir seu editor e ver os arquivos atualizados.

2.10 Git Push Branch to GitHub

Depois de criar ou atualizar um branch no seu repositório local, você pode enviá-lo (**push**) para o GitHub. Isso é fundamental quando você quer compartilhar suas mudanças com a equipe ou abrir um Pull Request.

2.10.1 Criar e mudar para um novo branch

```
git checkout -b update-readme
```

Saída:

```
Switched to a new branch 'update-readme'
```

2.10.2 Editar arquivos e verificar status

```
git status
```


- **Adicionar e commitar mudanças**

  ```
  git add README.md
  git commit -m "Update readme for GitHub"
  ```

- **Enviar branch para o GitHub**

  ```
  git push origin update-readme
  ```

- **Push com rastreamento (upstream)**

  Use quando o branch ainda **não existe no remoto**, para criar e vincular automaticamente:

  ```
  git push --set-upstream origin update-readme
  ```

- **Forçar push**

  Atenção: sobrescreva a versão no GitHub com a sua localização. Só use quando tiver certeza do que está fazendo.

  ```
  git push --force origin update-readme
  ```

- **Deletar branch remoto**

```
git push origin --delete update-readme
```

- **Push de todos os branches**

```
git push --all origin
```

- **Push de tags**

```
git push --tags
```

Problemas comuns

- **Rejected (non-fast-forward):** Outro dev fez push antes de você.

```
git pull --rebase
git push
```

- **Falha de autenticação:** Verifique se está logado corretamente e se tem permissão de escrita no repositório.
- **Branch remoto não encontrado:** Confira se o nome do branch foi digitado corretamente.

2.11 GitHub Flow

O **GitHub Flow** é um fluxo de trabalho simples e eficiente para colaborar em projetos usando **Git** e **GitHub**. Ele permite que equipes trabalhem juntas de forma organizada, experimentem com segurança e entreguem novas funcionalidades ou correções rapidamente.

2.11.1 Etapas do GitHub Flow

1. **Criar um branch**
   - Sempre mantenha o branch principal (`main` ou `master`) estável e pronto para deploy.
   - Para qualquer nova funcionalidade ou correção, crie um branch separado.
   - Use nomes descritivos para facilitar a compreensão de todos.

```
git checkout -b feature-login
```

2. **Fazer mudanças e commits**
   - Trabalhe no branch criado.

- Sempre que atingir um pequeno marco, faça um commit.
- Mensagens de commit devem explicar **o que foi feito** e **por quê**.

```
git add login.js
git commit -m "Add login form with basic validation"
```

3. **Abrir um Pull Request (PR)**
   - Quando o branch estiver pronto, crie um **Pull Request** no GitHub.
   - Isso sinaliza para os outros que suas mudanças estão prontas para revisão.
   - Os PRs permitem revisão colaborativa antes do merge.

4. **Revisão**
   - Equipe revisa o código: discute, sugere melhorias e identifica problemas.
   - Novos commits podem ser adicionados ao mesmo branch e aparecerão no PR automaticamente.

     Isso garante **colaboração** e **qualidade do código**.

5. **Deploy para teste**
   - Antes do merge, é possível **deployar** o branch em um ambiente de teste ou staging.
   - Se houver problemas, basta retornar para o branch `main` em produção.

6. **Merge do branch principal**
   - Após aprovação e testes, o branch é fundido ao `main`.
   - O histórico de PRs serve como documentação de mudanças e decisões.

```
git checkout main
git pull origin main
git merge feature-login
git push origin main
```

2.11.2 Benefícios do GitHub Flow

- **Simplicidade** → Fácil de aprender e aplicar.
- **Segurança** → Branches isolam mudanças sem afetar o código principal.
- **Colaboração** → Pull Requests incentivam revisão e discussões construtivas.
- **Rastreabilidade** → Histórico de commits e PRs documenta o projeto.
- **Agilidade** → Permite entrega contínua de novas funcionalidades.

Esse é o fluxo mais comum para projetos hospedados no GitHub, especialmente em times que adotam **Integração Contínua (CI)** e **Entrega Contínua (CD)**.

3 Git Contribute

3.1 GitHub Fork

O Git é uma ferramenta de colaboração, mas existe uma limitação importante, onde você não pode enviar código diretamente para o repositório de outra pessoa sem ter permissão de acesso.
Para resolver isso, o GitHub oferece o recurso de Fork.

3.1.1 O que é um Fork?

- Um Fork é uma cópia de um repositório existente.
- Ele fica associado ao repositório original, mas é independente.
- Com o fork, você pode:
    - Fazer alterações livremente.
    - Testar novas ideias.
    - Contribuir para o projeto original, sugerindo suas mudanças.

Importante: Fork não é um comando do Git. Ele é uma funcionalidade das plataformas de hospedagem de código, como GitHub e GitLab.

3.1.2 Quando usar Fork?

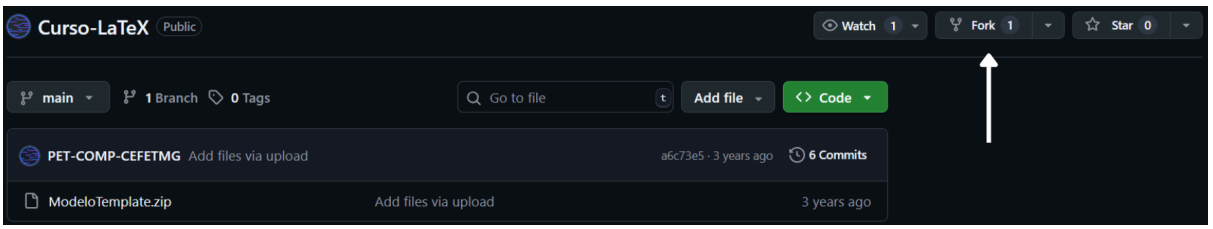
- Quando você quer contribuir em um projeto de código aberto.
- Quando você deseja personalizar um projeto existente para suas necessidades.
- Quando você quer experimentar em um repositório sem alterar o original.

3.1.3 Como fazer um Fork no GitHub

1. Acesse o repositório no GitHub.

    Exemplo: PET-COMP-CEFETMG/Curso-LaTeX

2. Clique no botão **Fork** (canto superior direito).



3. O GitHub criará uma **cópia do repositório** dentro da sua conta.

   Agora você pode clonar esse fork para sua máquina e trabalhar localmente:

```
git clone
https://github.com/PET-COMP-CEFETMG/Curso-LaTeX.github.io

cd Curso-LaTeX.github.io
```

3.2 Clonando um Fork do GitHub

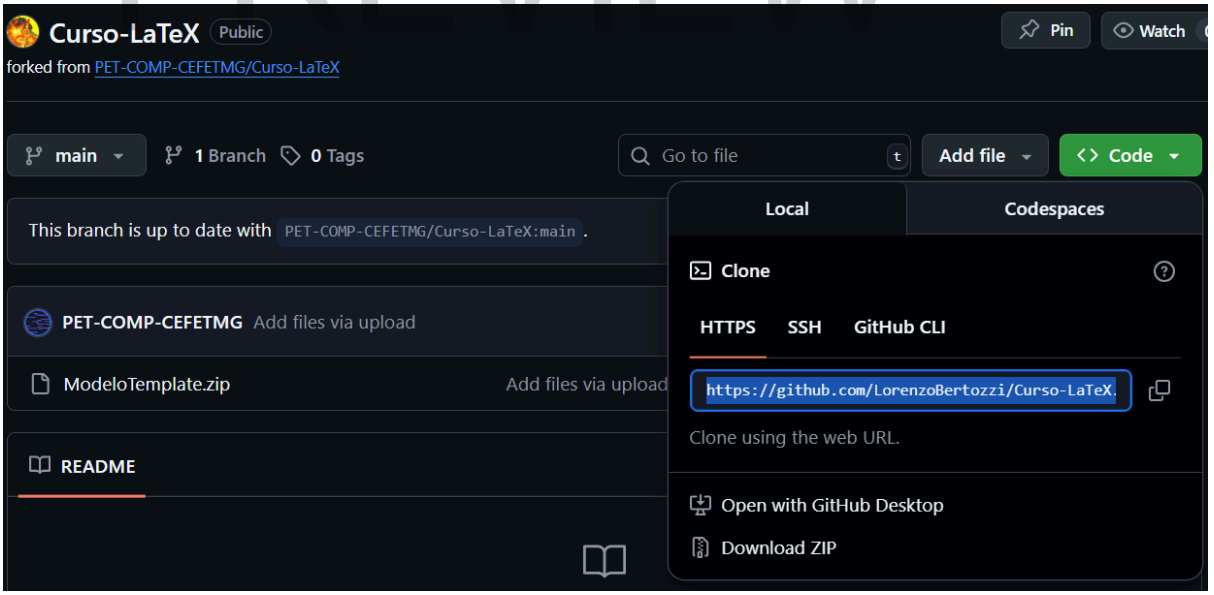
Depois de fazer o Fork, o repositório ainda está apenas na sua conta do GitHub. Para trabalhar localmente (no seu computador), precisamos clonar esse repositório.

Um Clone é uma cópia completa de um repositório Git, incluindo:

- Histórico de commits.
- Todas as versões dos arquivos.
- A estrutura do projeto.

3.2.1 Como clonar o repositório

1. Acesse o seu fork no GitHub.
2. Clique no botão verde **Code** e copie a URL (HTTPS, SSH ou GitHub CLI).



3. No terminal (Git Bash), digite:

```
git clone https://github.com/PET-COMP-CEFETMG/Curso-LaTeX.git
```

   Isso cria uma pasta com o nome do projeto.

   Exemplo:

```
ls
   -  Curso-LaTeX.git
```

Se quiser clonar para uma pasta com nome diferente:

```
git clone https://github.com/PET-COMP-CEFETMG/Curso-LaTeX.git
minha-pasta
```

3.2.2 Entrando no repositório clonado

```
cd w3schools-test.github.io
git status
```

Saída esperada:

```
On branch master
Your branch is up to date with 'origin/master'.
nothing to commit, working tree clean
```

3.2.3 Conferindo o histórico


```
git log
```

Isso confirma que você tem todo o histórico de commits do repositório.

3.2.4 Configurando Remotes

Agora precisamos organizar os remotes:

- origin: seu fork (onde você pode enviar alterações).
- upstream: repositório original (apenas leitura).

Passo 1: verificar o remote atual

```
git remote -v
```

Saída inicial:

```
origin
https://github.com/w3schools-test/w3schools-test.github.io.git
(fetch)
origin
https://github.com/w3schools-test/w3schools-test.github.io.git
(push)
```

Aqui o *origin* ainda aponta para o repositório original.

Passo 2: renomear o remote original para upstream

```
git remote rename origin upstream
```

Conferindo:

```
git remote -v

upstream
https://github.com/w3schools-test/w3schools-test.github.io.git
(fetch)
upstream
https://github.com/w3schools-test/w3schools-test.github.io.git
(push)
```

Passo 3: adicionar o fork como origin

```
git remote add origin
https://github.com/seu-usuario/w3schools-test.github.io.git
```


Verificando:

```
git remote -v

origin
https://github.com/seu-usuario/w3schools-test.github.io.git
(fetch)
origin
https://github.com/seu-usuario/w3schools-test.github.io.git (push)
upstream
https://github.com/w3schools-test/w3schools-test.github.io.git
(fetch)
upstream
https://github.com/w3schools-test/w3schools-test.github.io.git
(push)
```

Agora temos dois remotes configurados:

- `origin` → seu fork (onde você tem acesso total).
- `upstream` → repositório original (somente leitura).

Assim você pode manter seu fork atualizado com o original e, ao mesmo tempo, enviar suas alterações para o seu repositório.

4 Git Undo

4.1 Git Revert

O comando git revert desfaz um commit anterior criando um novo commit que reverte as alterações realizadas.Isso mantém o histórico de commits intacto e é a forma mais segura de desfazer mudanças em repositórios compartilhados.

Principais comandos do git revert

- `git revert HEAD` - Reverte o último commit
- `git revert <commit>` - Reverte um commit específico
- `git revert HEAD~2` - Reverte um commit mais antigo na linha do tempo
- `git revert --no-edit` - Reverte sem abrir o editor de mensagem
- `git log --oneline` - Lista os commits do repositório

Como encontrar o commit para reverter

Use o log resumido para visualizar o histórico:

```
git log --oneline
```


4.1.1 Executando o git revert

Depois de identificar o commit a ser revertido, basta executar:

```
git revert HEAD --no-edit
```

Exemplo de saída:

```
[master e56ba1f] Revert "Just a regular update, definitely no
accidents here..."
 Date: Thu Apr 22 10:50:13 2021 +0200
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 img_hello_git.jpg
```

4.1.2 Revisando alterações após o revert

Confira o histórico de commits:

```
git log --oneline
```

Saída:

```
e56ba1f (HEAD -> master) Revert "Just a regular update, definitely
no accidents here..."
52418f7 Just a regular update, definitely no accidents here...
9a9add8 (origin/master) Added .gitignore
...
221ec6e First release of Hello World!
```

Boas práticas

Prefira `git revert` em vez de `git reset` quando precisar desfazer commits sem perder o histórico.Use `git log --oneline` para identificar o commit a ser revertido.Se não quiser editar a mensagem do commit, use `--no-edit`.

Solução de problemas

Erro: could not revert…

Use `git revert --abort` para cancelar o processo.

Erro: could not apply…

Use `git revert --continue` para continuar após resolver conflitos.

4.2 Git Reset

O comando **`git reset`** move a referência do branch atual (**HEAD**) para um commit anterior.Dependendo da opção utilizada, ele pode:

- apenas mover o ponteiro,
- desfazer a preparação de arquivos (*unstage*),
- ou até mesmo apagar as alterações da sua pasta de trabalho.

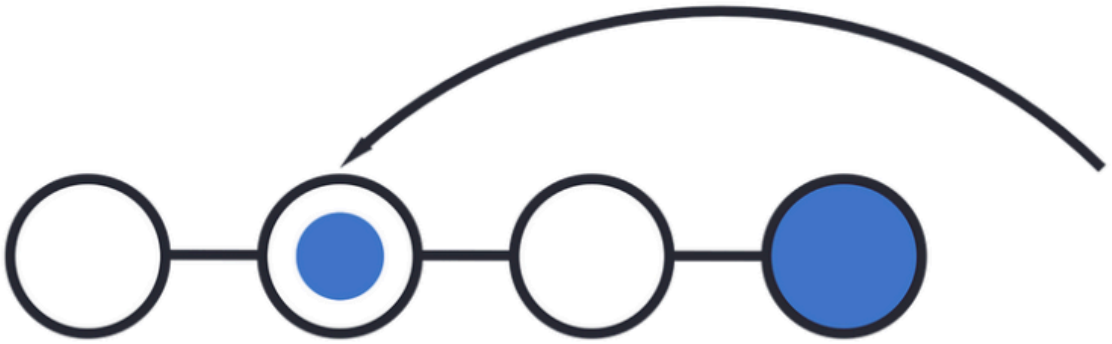
4.2.1 Resumo das opções do git reset

- `git reset --soft <commit>` - Move o HEAD, mantém as mudanças **no stage**
- `git reset --mixed <commit>` - Move o HEAD, mantém mudanças **não preparadas** (*default*)
- `git reset --hard <commit>` - Move o HEAD e **descarta todas as alterações**
- `git reset <file>` - Retira um arquivo do stage (sem alterar o conteúdo)
- `git log --oneline` - Mostra o histórico de commits resumido

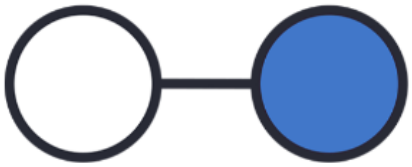
4.2.2 Como encontrar o commit para resetar

Antes de resetar, veja os commits disponíveis:

```
git log --oneline
```



Etapa 2: mova o repositório de volta para essa etapa:



Após o capítulo anterior, temos uma parte do nosso histórico de commits à qual podemos retornar.

Vamos tentar fazer isso com reset.

Primeiro, precisamos encontrar o ponto para o qual queremos retornar, para isso, precisamos percorrer o log. Para evitar uma lista de logs muito longa, usaremos a opção --oneline, que fornece apenas uma linha por commit mostrando:

- Os primeiros sete caracteres do hash do commit - é a isso que precisamos nos referir em nosso comando de reset.
- A mensagem do commit

Então, vamos encontrar o ponto para o qual queremos redefinir:

Exemplo de saída:

```
e56ba1f (HEAD -> master) Revert "Just a regular update, definitely
no accidents here..."
52418f7 Just a regular update, definitely no accidents here...
9a9add8 (origin/master) Added .gitignore
81912ba Corrected spelling error
3fdaa5b Merge pull request #1 from w3schools-test/update-readme
```

```
836e5bf (origin/update-readme, update-readme) Updated readme for
GitHub Branches
...
221ec6e First release of Hello World!
```

Suponha que queremos voltar para o commit **9a9add8 (Added .gitignore)**.

4.2.3 git reset --soft

Mantém as mudanças já **preparadas (staged)**. Útil para combinar commits ou reescrever o histórico mantendo o trabalho pronto.

```
git reset --soft 9a9add8
```

Agora todas as mudanças feitas após `9a9add8` continuam prontas no *stage*.

4.2.4 git reset --mixed (padrão)

Mantém as mudanças no diretório, mas **fora do stage**.


```
git reset --mixed 9a9add8
# ou simplesmente
git reset 9a9add8
```

As mudanças permanecem nos arquivos, mas precisam ser adicionadas novamente com `git add`.

4.2.5 git reset --hard

Cuidado! Remove completamente todas as mudanças **do stage e do diretório de trabalho**.

```
git reset --hard 9a9add8
```

Seu repositório ficará **exatamente** no estado do commit especificado.

4.2.6 Revisando mudanças após o reset

Depois de resetar, verifique o estado do repositório:

```
git status
```

E veja o histórico atualizado:

```
git log --oneline
```

Boas práticas

- Use `--soft` ou `--mixed` quando não quiser perder alterações.
- Evite `--hard` em projetos compartilhados, pois ele apaga tudo.
- Sempre confirme o commit alvo com `git log --oneline` antes de resetar.
- Combine `git reset` com `git status` para revisar seu repositório.

Avisos importantes

- `git reset` **reescreve o histórico**.
- Não utilize `git reset` em commits já enviados (*push*) para o repositório remoto sem antes conversar com a equipe.
- Para desfazer mudanças em repositórios colaborativos, prefira `git revert`.

4.3 Git Amend

O comando `git commit --amend` permite **alterar o commit mais recente**. Você pode:

- Corrigir erros de digitação na mensagem.
- Adicionar arquivos esquecidos.
- Remover arquivos adicionados por engano.

Atenção: Esse comando reescreve o último commit. Use com cuidado, principalmente se já tiver enviado (`push`) para um repositório remoto.

Quando Usar Git Amend ?

- Para **pequenas correções** no último commit.
- Para **atualizar a mensagem**.
- Para **incluir arquivos esquecidos**.
- Para **remover arquivos indesejados**.

4.3.1 Alterar Mensagem do Último Commit

```
git commit --amend -m "Nova mensagem do commit"
```

Exemplo:

```
git commit --amend -m "Corrigido erro no README"
```

4.3.2 Adicionar Arquivos ao Último Commit

1. Adicione o arquivo esquecido:

```
git add arquivo.txt
```

2. Faça o amend:

```
git commit --amend
```

Isso atualiza o commit mais recente com o novo arquivo incluído.

4.3.3 Remover Arquivos do Último Commit

1. Retire o arquivo do commit anterior:

```
git reset HEAD -- arquivo.txt
```

2. Refaça o commit sem o arquivo:

```
git commit --amend
```

4.3.4 Exemplo de Log Antes e Depois

Antes (mensagem errada):

```
git log --oneline
07c5bc5 Adding plines to reddme
```

Depois (corrigido):

```
git log --oneline
eaa69ce Added lines to README.md
```

O commit antigo foi substituído pelo novo.

Nota Importante

- Alterar commits **locais** geralmente é seguro.
- **Evite** usar `--amend` em commits já enviados para o repositório remoto, pois isso pode causar problemas para outros colaboradores.

4.4 Git Rebase

O **Git Rebase** move ou combina uma sequência de commits para um novo commit base. Ele é usado principalmente para manter um **histórico de commits limpo e linear**, facilitando a leitura e evitando *merges* desnecessárias.

Use o **Git Rebase** para:

- Manter um histórico limpo e linear
- Evitar commits de merge desnecessários
- Combinar múltiplos commits em um só
- Editar ou reordenar commits

4.4.1 Rebase Básico

Para mover sua branch atual sobre outra (por exemplo, atualizar sua branch de feature com a última versão da *main*):

Exemplo: Rebase na main

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

Isso re-aplica às mudanças da sua *feature branch* em cima da versão mais recente da branch *main*.

4.4.2 Rebase Interativo

O comando `git rebase -i <base>` permite editar, reordenar, combinar (*squash*) ou ajustar commits antes de um ponto específico. Isso é útil para **limpar o histórico de commits** antes de compartilhar seu código.

Exemplo: Iniciar Rebase Interativo

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

Esse comando abre um editor onde você pode escolher:

- **pick** → manter o commit como está
- **squash** → combinar commits
- **edit** → pausar e modificar um commit
- **reword** → alterar apenas a mensagem do commit

Passos:

1. Escolha a ação (pick, squash, edit, reword)
2. Salve e feche o editor
3. O Git aplicará as alterações e permitirá revisão

4.4.3 Continuar, Abortar ou Pular

- **Continuar após resolver conflitos:**

```
git add arquivo_corrigido.txt
git rebase --continue
```

- **Abortar o rebase:**

```
git rebase --abort
```

- **Pular um commit problemático:**

```
git rebase --skip
```

Revisão das Alterações

Após finalizar o rebase, revise as mudanças para garantir que tudo está correto.

Boas Práticas

O rebase **reescreve o histórico de commits**.

- Evite fazer rebase em commits já enviados para repositórios remotos compartilhados.
- Prefira usar `git rebase -i` para limpar o histórico antes de compartilhar.
- Use `git rebase --continue` após resolver conflitos.
- Use `git rebase --abort` se quiser cancelar o processo.

Resolução de Problemas

- Se houver conflitos: resolva-os e use `git rebase --continue`.
- Se não for possível corrigir um commit: use `git rebase --skip`.

Nota: nunca rebase commits já enviados para o remoto em repositórios compartilhados.

4.5 Git Reflog

O comando `git reflog` registra as atualizações feitas na ponta dos branches e no **HEAD**. Ele permite visualizar onde seu branch e HEAD já estiveram, inclusive mudanças feitas por engano. Isso é extremamente útil para **recuperar commits perdidos** ou **desfazer um reset**.

Use `git reflog` quando precisar de:

- Recuperar commits ou alterações perdidas
- Desfazer um reset ou merge

- Visualizar o histórico do seu branch e do HEAD

4.5.1 Mostrar o Reflog

Para ver o histórico de posições do HEAD e dos branches, use:

```
git reflog
```

Exemplo de saída:

```
e56ba1f (HEAD -> master) HEAD@{0}: commit: Revert "Just a regular
update, definitely no accidents here..."
52418f7 HEAD@{1}: commit: Just a regular update, definitely no
accidents here...
9a9add8 (origin/master) HEAD@{2}: commit: Added .gitignore
81912ba HEAD@{3}: commit: Corrected spelling error
3fdaa5b HEAD@{4}: merge: Merge pull request #1 from
w3schools-test/update-readme
836e5bf HEAD@{5}: commit: Updated readme for GitHub Branches
...
```


Essa lista mostra as posições recentes do HEAD, incluindo ações como commits, resets, merges e checkouts.

4.5.2 Encontrar e Recuperar Commits Perdidos

Se você resetou ou apagou commits por engano, o **reflog** pode ajudar a encontrá-los e restaurá-los.

Cada entrada no reflog tem uma referência, como `HEAD@{2}`.

Exemplo: desfazer um Hard Reset

```
git reflog

e56ba1f (HEAD -> master) HEAD@{0}: commit: Revert "Just a regular
update, definitely no accidents here..."
52418f7 HEAD@{1}: commit: Just a regular update, definitely no
accidents here...
9a9add8 (origin/master) HEAD@{2}: commit: Added .gitignore
81912ba HEAD@{3}: commit: Corrected spelling error
...
```

```
git reset --hard HEAD@{2}
```

Saída:

```
HEAD is now at 9a9add8 Added .gitignore
```

Isso coloca seu branch de volta ao estado em que estava naquele ponto.

4.5.3 Limpar o Reflog

O Git limpa o reflog automaticamente, mas você pode remover entradas antigas manualmente se necessário:

```
git reflog expire --expire=30.days refs/heads/main
git gc --prune=now
```

Saída:

```
Counting objects: 15, done.
Compressing objects: 100% (10/10), done.
Pruning objects
```


Esses comandos removem entradas do reflog mais antigas que 30 dias no branch `main` e executam a coleta de lixo.

4.5.4 Dicas & Boas Práticas

- Use `git reflog` para acompanhar mudanças e recuperar alterações perdidas
- Utilize `git reflog expire` para limpar entradas antigas
- Combine `git reflog` com `git reset` para restaurar estados anteriores

Solução de Problemas

- Se tiver conflitos ao recuperar commits, tente `git reset` com a referência correta (`HEAD@{n}`)
- Consulte a documentação oficial do Git para detalhes avançados
- Busque ajuda na comunidade se não conseguir resolver o problema

O **git reflog reescreve o histórico local**, tenha cuidado ao recuperar commits, pois você pode sobrescrever alterações atuais.

4.6 Git Recovery

Git Recovery significa recuperar **commits, branches ou arquivos perdidos**. O Git mantém um histórico interno de mudanças recentes, permitindo **desfazer erros** — mesmo após um `reset` ou `delete`.

Você pode usar o **Git Recovery** quando:

- Apagar acidentalmente um branch ou arquivo
- Faz um `git reset` e perde commits
- Precisa restaurar alterações importantes

4.6.1 Recuperar Commits Perdidos com `git reflog`

O comando `git reflog` registra mudanças no ponteiro das branches, permitindo encontrar **commits "perdidos"**.

Exemplo: Mostrar o Reflog

```
git reflog
e56ba1f (HEAD -> master) HEAD@{0}: commit: Revert "Just a regular
update, definitely no accidents here..."
52418f7 HEAD@{1}: commit: Just a regular update, definitely no
accidents here...
9a9add8 (origin/master) HEAD@{2}: commit: Added .gitignore
81912ba HEAD@{3}: commit: Corrected spelling error
3fdaa5b HEAD@{4}: merge: Merge pull request #1 from
w3schools-test/update-readme
836e5bf HEAD@{5}: commit: Updated readme for GitHub Branches
```


Basta copiar o **hash do commit** que deseja recuperar.

4.6.2 Restaurar um Branch Deletado

Se você apagou um branch, mas os commits ainda aparecem no `reflog`, recrie-o:

```
git checkout -b branch-name <commit-hash>
```

O branch será recriado exatamente a partir do commit escolhido.

4.6.3 Recuperar Arquivo Deletado ou Alterado

Se você apagou ou modificou um arquivo e deseja restaurá-lo:

```
git restore filename.txt
```

Isso traz o arquivo de volta do **último commit**.

w4.5.4 Recuperar Após um Hard Reset

Se você usou `git reset --hard` e perdeu commits, pode recuperá-los com o `reflog`.

```
git reflog
git reset --hard HEAD@{2}
```

Isso leva seu branch de volta ao estado em que estava no `HEAD@{2}`.

Dicas & Boas Práticas

- Faça **commits frequentes** para não perder trabalho
- Use `git reflog` para localizar commits desaparecidos
- Use `git restore` para recuperar arquivos deletados