🧩 Erros Comuns na Instalação do Git

🔹 1. Git não está instalado corretamente

💻 Windows

- Erro: Após a instalação, o comando `git` não funciona no terminal (CMD ou PowerShell).
- Causa: O Git não foi adicionado ao PATH do sistema.
- Solução:
    1. Reinstale o Git.
    2. Na tela de instalação, selecione: “Git from the command line and also from 3rd-party software”.
    3. Verifique com: `git --version`.

🐧 Linux (Ubuntu/Debian)

- Erro: `git: command not found`
- Causa: Git não está instalado.
- Solução:

🔹 2. Versão desatualizada do Git

💻 Windows

- Erro: Recursos como `git switch` não funcionam.
- Causa: Versão antiga do Git.
- Solução:
    - Baixe a versão mais recente em: https://git-scm.com

🐧 Linux

- Solução:

🔹 3. Problemas com autenticação (HTTPS vs SSH)

- Erro: `fatal: Authentication failed`
- Causa: Credenciais incorretas ou método de autenticação incompatível.
- Solução:
    - Use SSH para evitar digitar senha toda vez:
    - Adicione a chave pública ao GitHub/GitLab.

🔹 4. Configuração de usuário não feita

- Erro: Commits sem nome ou e-mail.
- Solução:

🔹 5. Permissões negadas ao clonar repositórios

- Erro: `Permission denied (publickey)`
- Causa: Chave SSH não configurada ou não adicionada ao serviço remoto.
- Solução:
    - Verifique se a chave está no agente SSH.
    - Adicione ao GitHub/GitLab via interface web.

🔹 6. Git não reconhecido em IDEs (VS Code, etc.)

- Erro: VS Code não detecta Git.
- Causa: Git não está no PATH ou não instalado.
- Solução:
    - Verifique se o Git está instalado com `git --version`.
    - Reinicie o VS Code após a instalação.

Git:

Git é um sistema de controle de versão distribuído, criado por Linus Torvalds. Diferente de sistemas centralizados como o SVN, onde o histórico do projeto fica em um servidor único, o Git permite que cada desenvolvedor tenha uma cópia completa do repositório, incluindo todo o histórico de alterações. Isso não só garante a redundância dos dados, mas também torna as operações como **commits** e navegação no histórico extremamente rápidas, já que a maioria das ações não precisa de conexão com a internet.

Os Comandos em Detalhes

- **git init**: Ao executar este comando, o Git cria uma pasta oculta chamada `.git` na raiz do seu projeto. É nessa pasta que ele armazena todas as informações do seu repositório, como o histórico de commits, a referência de **branches** e o estado do seu **staging area**.
- **git status**: Este comando analisa a relação entre sua **working directory** (a pasta do seu projeto), o **staging area** e o seu último **commit** no repositório.

Ele detalha quais arquivos foram modificados, quais estão na **staging area** aguardando o **commit** e quais arquivos não estão sendo rastreados pelo Git.
- **git add nome-do-arquivo**: Adicionar um arquivo não salva a alteração permanentemente. O `git add` move o arquivo para uma área intermediária, o **staging area**. Pense nela como uma espécie de "cesta de compras" onde você agrupa as alterações que quer incluir no próximo **commit**.
- **git commit -m "mensagem"**: O **commit** é o ponto crucial do controle de versão. Ele pega todas as alterações no **staging area** e as registra no histórico do projeto. Cada **commit** é um instantâneo único do seu código, com um identificador de **hash** único (SHA-1) que garante a integridade dos dados. A mensagem (**-m**) é essencial para descrever o objetivo daquele **commit**, facilitando a navegação futura.
- **git log**: Mais do que uma simples lista, o `git log` é uma janela para a história do seu projeto. Ele mostra a sequência de **commits** de forma cronológica, com informações como o autor, a data e a **hash** de cada **commit**, além da mensagem. Você pode usar parâmetros como `--oneline` para uma visualização mais compacta ou `--graph` para ver o histórico de **branches**.
- **git diff**: O `git diff` compara diferentes estados do seu repositório. Sem nenhum parâmetro, ele mostra as alterações entre o seu **working directory** e o **staging area**. Usando `git diff --staged`, você pode ver a diferença entre o **staging area** e o último **commit**.

Por Que Usar Git, e Não Prompt ou PowerShell?

O **Prompt de Comando** (ou **PowerShell** no Windows, e o **Terminal** no Linux/macOS) é uma interface de linha de comando que permite interagir diretamente com o sistema operacional. Eles são ferramentas poderosas, mas são feitos para executar comandos do sistema, gerenciar arquivos e processos. O Git, por outro lado, é uma **aplicação** especializada em controle de versão.

Aqui estão os principais motivos para utilizar o Git:

- **Gerenciamento de Histórico**: O Prompt de Comando não rastreia alterações em arquivos. Se você apagar uma linha de código, ela se foi para sempre, a menos que você tenha uma cópia de segurança manual. O Git, com seus **commits**, mantém um histórico imutável de cada mudança, permitindo que você volte no tempo para qualquer versão anterior, se necessário.
- **Colaboração e Sincronização**: O Prompt de Comando não possui funcionalidades para gerenciar a colaboração em projetos. O Git foi construído para isso. Com comandos como `git clone`, `git push` e `git`

`pull`, ele permite que equipes trabalhem simultaneamente no mesmo projeto, unam alterações de forma organizada e resolvam conflitos.

- **Branching (Ramificação)**: A principal força do Git é o conceito de **branches**. Você pode criar uma ramificação do seu projeto para trabalhar em uma nova funcionalidade, sem afetar o código principal. Se a funcionalidade der certo, você pode mesclar a **branch** de volta (com `git merge`). Se não, pode simplesmente abandoná-la. Isso é impossível de fazer de forma segura e eficiente usando apenas ferramentas de sistema operacional.
- **Integridade de Dados**: O Git usa o algoritmo de **hash** SHA-1 para garantir a integridade dos seus dados. Cada **commit** tem uma **hash** única que depende de todo o histórico anterior, o que significa que qualquer alteração maliciosa ou corrupção de dados seria imediatamente detectada. O Prompt de Comando não tem essa capacidade.