GUIA COMPLETO DO INÍCIO AO AVANÇADO DE GIT E GITHUB

Módulo 1: Introdução ao Git

O que é Git?

- Git é um sistema de controle de versão distribuído criado por Linus Torvalds. Ele permite que desenvolvedores salvem, revertam e colaborem em versões de código-fonte de forma eficiente.

Conceitos-chave

- **Repositório**: local onde o histórico do projeto é armazenado.
- **Commit**: uma "foto" do estado atual dos arquivos.
- **Branch**: ramificação independente do histórico.
- **Merge**: junção de branches.
- **HEAD**: referência ao commit atual.

CONCEITOS BÁSICOS:

Para se conectar ao seu Git em máquinas diferentes use:

git config --global user.name "Seu Nome" - Aqui você vai colocar o nome que tem no seu perfil do GitHub

git config --global user.email "seu@email.com" - Aqui você vai colocar o email que você cadastrou no GitHub

git init - O comando `git init` cria um novo repositório Git em um diretório especificado. Ele inicializa um diretório oculto chamado `.git`, que contém todas as configurações e metadados necessários para o controle de versão.

Como usar `git init`?

Para iniciar um repositório Git, basta navegar até o diretório onde deseja criar o repositório e executar:

git init

Isso criará um repositório Git vazio nesse diretório.

Se quiser iniciar um repositório em um diretório específico sem precisar navegar até ele, pode usar:

git init nome_do_repositorio

Isso criará o diretório `nome_do_repositorio` e inicializará um repositório Git dentro dele.

O que acontece internamente?

Após executar `git init`, o Git cria uma estrutura de diretórios dentro de `.git` com os seguintes componentes:

- **HEAD:** Aponta para o branch atual.
- **config:** Contém configurações do repositório.
- **description:** Usado em repositórios `git daemon` para fornecer uma descrição.
- **hooks/:** Scripts que podem ser executados automaticamente em eventos do Git.
- **info/:** Contém arquivos de controle, como `exclude`, que define padrões de arquivos a serem ignorados.
- **objects/:** Armazena objetos de dados do Git.
- **refs/:** Contém referências para commits, branches e tags.

Exemplo prático:

mkdir meu_projeto -> cria uma pasta para o seu projeto
cd meu_projeto -> navegar até essa pasta
git init -> cria um novo repositório
ls -a -> verifica se foi configurado corretamente, exibe o diretório oculto

git status - exibe o estado atual do seu repositório, mostrando quais arquivos foram modificados, adicionados ou removidos e informando se há algo pendente para commit.

Como usar `git status`?

Basta rodar este comando dentro do seu repositório Git:

git status

Ele fornecerá informações como:

- Branch atual: Mostra em qual branch você está trabalhando.
- Arquivos não rastreados: Arquivos que ainda não foram adicionados ao controle de versão.
- Modificações pendentes: Arquivos alterados mas ainda não preparados para commit.
- Staged changes: Arquivos que já foram adicionados com `git add`, prontos para commit.

git add . - Adiciona todos os arquivos modificados ou novos no diretório atual ao staging area. Isso inclui alterações em arquivos existentes e arquivos recém-criados.

git add <arquivo> - Adiciona apenas o arquivo especificado ao staging area, preparando-o para o commit sem afetar outros arquivos.

Como usar `usar o git add ou git add <arquivo>.`

```
git add . # usando git add .

git status  # Verifica quais arquivos foram adicionados

git add exemplo.txt  # Adiciona um arquivo específico novamente, se necessário

git commit -m "Adicionando todas as mudanças e ajustando um arquivo específico"


git add index.html # usando git add <arquivo>

git add style.css # usando git add <arquivo>

git commit -m "Adicionando arquivos específicos e depois tudo"
```

git commit -m “msg” - Esse comando cria um **commit**, que é um registro das mudanças feitas nos arquivos adicionados ao **staging area**. A flag `-m "msg"` permite adicionar uma mensagem descritiva ao commit.

Como usar?

Após adicionar arquivos com `git add`, execute:

```
git commit -m "Minha mensagem de commit"
```

git log - exibe uma lista dos commits realizados no repositório, mostrando detalhes como:

- **Hash do commit** (identificador único).
- **Autor do commit**.
- **Data e hora**.
- **Mensagem do commit**.

Como usar?

git log

git log --oneline # Isso mostrará cada commit em uma única linha

Opções úteis

- **`git log --graph`** - **Mostra a estrutura do histórico visualmente.**
- **`git log --author="Nome"`** - **Filtra commits por autor.**
- **`git log -n 5`** - **Exibe apenas os 5 commits mais recentes.**
- **`git log --since="2 days ago"`**- **Mostra commits dos últimos 2 dias**

Trabalhando com Branches

Branches (ramificações) permitem criar diferentes versões do projeto sem afetar a branch principal. Isso é útil para:

- **Desenvolvimento de novas funcionalidades** sem interferir no código principal.
- **Correções de bugs** em paralelo ao desenvolvimento.
- **Trabalho colaborativo**, onde cada desenvolvedor pode ter sua própria branch.
- Desenvolver **novas funcionalidades** sem alterar o código estável.
- **Corrigir bugs** isoladamente.
- **Trabalhar colaborativamente**, onde cada desenvolvedor pode ter sua própria branch.

Por padrão, o Git inicia com uma branch chamada `main` ou `master`.

Como usar?

git branch nome-da-branch # criar uma nova branch

git checkout nome-da-branch # Para alternar para uma branch existente

git switch nome-da-branch # se estiver usando uma versão mais recente do Git

git checkout -b nome-da-branch # pode criar e alternar para a branch ao mesmo tempo

git switch -c nome-da-branch # na versão mais recente do Git

git branch # Para listar todas as branches **locais, A branch ativa será marcada com** *

git branch -r # Ver **branches remotas**

git branch -a # Para listar **todas** (locais + remotas):

Depois de trabalhar em uma branch, você pode incorporá-la à branch principal:

git checkout main # Alternar para a branch principal

git merge nome-da-branch # Mesclar as alterações

Se houver conflitos, o Git pedirá que você os resolva antes de finalizar a mesclagem.

Se uma branch não for mais necessária, pode removê-la:

git branch -d nome-da-branch # Excluir uma branch local

git branch -D nome-da-branch # Forçar a exclusão

git push origin --delete nome-da-branch # Excluir uma branch remota

O que é GitHub?

- O **GitHub** é uma plataforma online que permite hospedar e gerenciar **repositórios Git**. Ele facilita o compartilhamento de código, colaboração entre desenvolvedores e controle de versão

Configuração do GitHub

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

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

git remote add origin <url> # Conecta ao GitHub na url específica, ou na sua própria url

git clone <url> # clona um repositório GitHub no seu repositório local do Git

git pull # Baixa alterações feitas na brach ou naquele repositório específico, **é um padrão de projeto (boa prática)sempre fazer um git pull para adquirir as atualizações feitas por outros membros que estão trabalhando na mesma branch**

git fetch # Busca **atualizações do repositório remoto**, mas **não** aplica essas mudanças automaticamente ao seu repositório local. Ele é útil para ver o que mudou no remoto antes de atualizar sua cópia local.

`git fetch` - Baixa as mudanças **sem aplicá-las**. Você pode analisá-las antes de atualizar sua cópia local.

`git pull` - Baixa **e aplicar imediatamente** as mudanças na sua branch atual.

git push -u origin master # Envia Branch para GitHub, **é uma boa prática sempre fazer assim que terminou de trabalhar no projeto.**

Comandos Avançados

git stash # Salva alterações permite guardar alterações temporariamente sem fazer um commit. Isso é útil quando você precisa alternar de branch ou trabalhar em outra tarefa sem perder seu progresso atual.

git stash list # Isso exibe uma lista de alterações salvas.

git stash drop stash@{n} # Onde **n** é o índice do stash na lista

git stash pop # Restaure as alterações salvas no stash e remova esse stash da lista. Isso recupera a última alteração salva no stash.

git stash apply stash@{n} # Se quiser aplicar um stash específico sem removê-lo da lista, use

git rebase # permite reorganizar commits ao aplicar mudanças de um branch sobre outro. É útil para manter um histórico limpo.

git cherry-pick <hash> # Esse comando permite copiar um commit de um branch para outro sem trazer todos os commits da sequência

git revert <hash> # Diferente do `git reset`, `git revert` cria um novo commit que **inverte** as alterações de um commit anterior.

git reset --hard <hash> # Reverte seu repositório completamente para um estado anterior, **apagando todas as mudanças posteriores**.

exemplo: git reset --hard a1b2c3d

Isso descarta todas as alterações e redefine o histórico para o commit `a1b2c3`.
Cuidado: As alterações serão **perdidas** se não forem salvas antes!

EXEMPLO COMPLETO E COMENTADO:

🎯 1. Criando um novo repositório Git

```
mkdir meu_projeto
cd meu_projeto
git init
```

- Isso inicializa um repositório Git local.

📜 2. Criando um arquivo e adicionando-o ao Git

```
echo "Primeira versão do código" > app.js
git add app.js
git commit -m "Adicionando arquivo inicial"
```

- Isso adiciona `app.js` ao **staging area** e o registra no histórico com um commit.

🌍 3. Criando um repositório remoto e conectando ao GitHub

```
git remote add origin https://github.com/seu-usuario/meu_projeto.git
```

```
git push origin main
```

🔹 Agora o repositório está vinculado ao GitHub e sincronizado.

🔄 4. Criando e alternando para uma nova branch

```
git branch nova-feature
git checkout nova-feature
```

🔹 Criamos e alternamos para a branch `nova-feature`, onde trabalharemos em novas funcionalidades.

🏗️ 5. Fazendo alterações e commitando na nova branch

```
echo "Nova funcionalidade adicionada" >> app.js
git add app.js
git commit -m "Implementando nova funcionalidade"
```

🔹 Isso adiciona e registra as novas alterações no histórico da branch.

🔁 6. Salvando alterações temporárias com `git stash`

Se precisarmos mudar de tarefa sem perder progresso:

```
git stash
```

🔹 Isso guarda as alterações e nos permite alternar de branch sem perder código.

Quando quisermos recuperá-las:

```
git stash pop
```

🔄 7. Sincronizando com repositório remoto e verificando histórico

```
git fetch
git log --oneline
```

🔹 `git fetch` traz atualizações do repositório remoto sem aplicá-las.

🔹 `git log --oneline` exibe o histórico de commits.

🔗 8. Mesclando a branch `nova-feature` na `main`

```
git checkout main
git merge nova-feature
```

🔹 Agora `nova-feature` foi integrada à `main`.

Se quisermos deletar a branch:

```
git branch -d nova-feature
```

🎯 9. Revertendo um commit

Caso um commit tenha causado problemas:

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

🔹 Isso cria um novo commit **revertendo** o commit problemático.

🚀 10. Copiando um commit específico para outra branch

```
git cherry-pick <hash-do-commit>
```

🔹 Isso copia um commit específico para nossa branch atual.

🔥 11. Reaplicando commits com `git rebase`

Se quisermos manter um histórico organizado:

```
git checkout nova-feature
git rebase main
```

🔹 Isso reorganiza os commits da `nova-feature`, garantindo que estejam alinhados com `main`.

❌ 12. Resetando para um estado anterior

Caso precise **descartar** mudanças e voltar para um commit antigo:

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

⚠️ **Cuidado!** Isso removerá todas as alterações feitas após esse commit.

🏆 Resumo do fluxo completo

1️⃣ **Criamos um repositório Git e o vinculamos ao GitHub**

2️⃣ **Criamos e gerenciamos branches**

3️⃣ **Commitamos alterações e usamos stash para mudanças temporárias**

4️⃣ **Sincronizamos o repositório local e remoto**

5️⃣ **Mesclamos branches, revertendo commits problemáticos**

6️⃣ **Utilizamos comandos avançados como `cherry-pick`, `rebase` e `reset`**