GITHUB

GIT E GITHUB

Git é um sistema de versionamento distribuído (controla as modificações de um projeto por meio de versões chamadas “**commits**”), ou seja, uma ferramenta de gerenciamento de código fonte que permite que várias pessoas trabalhem em um projeto simultaneamente, com um histórico completo de todas as alterações feitas no código. O Git permite que desenvolvedores controlem e gerenciem diferentes versões de um mesmo projeto, facilitando a colaboração e o trabalho em equipe.

GitHub, por sua vez, é uma plataforma web que fornece hospedagem para projetos Git. É um serviço de hospedagem de repositórios Git na nuvem, que oferece diversas funcionalidades, tais como controle de versão, gerenciamento de problemas, colaboração em equipe, entre outras. O GitHub é amplamente utilizado pela comunidade de desenvolvimento de software como um espaço para compartilhar e colaborar em projetos de código aberto.

Em resumo, o Git é uma ferramenta de controle de versão, enquanto o GitHub é uma plataforma de hospedagem de repositórios Git. O Git pode ser utilizado sem o GitHub, mas o GitHub não funciona sem o Git.

COMMIT

Em Git, um commit é uma operação que registra uma alteração no código-fonte em um repositório Git. Ele é usado para salvar as alterações feitas em um arquivo ou em um conjunto de arquivos no repositório.

Cada commit no Git tem um identificador único chamado de hash SHA-1, que é gerado automaticamente pelo Git. Esse hash identifica exclusivamente o commit e permite que ele seja referenciado em outras operações do Git, como a criação de branches e merges.

Ao fazer um commit no Git, é necessário adicionar uma mensagem que descreva brevemente as alterações realizadas. Essa mensagem é importante para que outras pessoas que estejam colaborando no projeto possam entender o que foi feito em cada commit. Além disso, as mensagens de commit são úteis para fazer buscas e identificar rapidamente quais alterações foram feitas em determinado ponto da história do projeto.

Os commits são uma parte fundamental do processo de controle de versão em Git, pois permitem que os desenvolvedores trabalhem em equipe de forma eficiente, rastreando as alterações realizadas e revertendo alterações indesejadas, se necessário.

REPOSITÓRIO REMOTO E REPOSITÓRIO LOCAL

Em um projeto controlado pelo Git, o "repositório remoto" é uma cópia do repositório Git em um servidor remoto, que pode ser acessado por várias pessoas que estão trabalhando no projeto. O repositório remoto é geralmente usado para compartilhar o código com outros desenvolvedores e para fazer backup do código em um servidor seguro. Exemplos de plataformas de hospedagem de repositórios remotos Git incluem o GitHub, GitLab e Bitbucket.

O "repositório local", por outro lado, é uma cópia do repositório Git em um computador local. É a versão do projeto que um desenvolvedor tem em seu próprio computador e na qual ele pode trabalhar e fazer alterações sem interferir no trabalho de outros desenvolvedores. Todas as alterações feitas no repositório local são armazenadas em um histórico local e podem ser sincronizadas com o repositório remoto, permitindo que outras pessoas do projeto vejam e trabalhem nessas alterações.

Em resumo, o repositório remoto é o repositório Git compartilhado com outros desenvolvedores, enquanto o repositório local é a cópia do repositório Git em um computador local de um desenvolvedor.

COMANDOS “CLONE”, “PUSH” E “PULL

Em Git, o "clone" é uma operação que permite que um desenvolvedor faça uma cópia completa do repositório Git remoto em seu computador local. O comando "git clone" cria uma cópia idêntica do repositório remoto em um diretório local, incluindo todo o histórico de commits e branches do projeto.

O "pull", por sua vez, é uma operação que atualiza o repositório local com as alterações feitas no repositório remoto. O comando "git pull" baixa as atualizações mais recentes do repositório remoto e mescla essas alterações com o repositório local.

A principal diferença entre "clone" e "pull" é que o clone cria uma cópia completa do repositório remoto, enquanto o pull atualiza apenas as alterações mais recentes. Portanto, o clone é geralmente usado para fazer uma cópia inicial do repositório remoto, enquanto o pull é usado para atualizar o repositório local com as alterações mais recentes.

O "push" é uma operação que envia as alterações feitas no repositório local para o repositório remoto. O comando "git push" envia os commits que foram feitos no repositório local para o repositório remoto correspondente. O push é usado para compartilhar as alterações feitas no código com outros desenvolvedores que estão trabalhando no projeto.

Em resumo, o "clone" é usado para criar uma cópia completa do repositório remoto no computador local, o "pull" é usado para atualizar o repositório local com as alterações mais recentes do repositório remoto, e o "push" é usado para enviar as alterações feitas no repositório local para o repositório remoto.

CONFIGURANDO SUA IDENTIFICAÇÃO NO GIT

Configurar sua identificação no Git é importante para que você possa ser reconhecido como autor de alterações em um repositório e rastrear suas contribuições. Aqui estão os passos para configurar sua identificação no Git:

1. Abra um terminal ou prompt de comando e digite o seguinte comando para verificar se o Git está instalado no seu sistema:

**git –-version**

1. Depois de confirmar que o Git está instalado, digite os seguintes comandos para configurar seu nome de usuário e endereço de e-mail no Git:

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

**git config --global user.email "seuemail@exemplo.com"**

1. Você também pode configurar um editor de texto padrão para ser usado pelo Git. Por exemplo, se você quiser usar o editor Vim, digite o seguinte comando:

**git config --global core.editor "vim"**

1. Para visualizar as configurações que você acabou de definir, digite o seguinte comando:

**git config --list**

Isso mostrará uma lista de todas as configurações do Git em seu sistema, incluindo seu nome de usuário e endereço de e-mail.

CONFIGURAR CHAVE SSH PARA O GITHUB

SSH é um protocolo para comunicação de dados com segurança.

O Github aboliu a autenticação somente com usuário e senha.

A ideia básica é cadastrar previamente quais computadores podem acessar o Github em seu nome. Outros computadores não conseguem acessar.

Para isto você deve:

1. Gerar uma chave SSH no seu computador;
2. Cadastrar essa chave no seu Github.

A configuração de uma chave SSH para o GitHub permite que você se conecte ao GitHub de forma mais segura e fácil, sem a necessidade de inserir suas credenciais toda vez que fizer push ou pull de um repositório. Aqui estão os passos para configurar uma chave SSH para o GitHub:

1. Abra o Git Bash: O Git Bash é um terminal que inclui o Git para Windows. Ele permite que você use comandos do Git e do Unix no Windows. E digite o seguinte comando para verificar se você já tem uma chave SSH configurada:

**$ ls -al ~/.ssh**

Se você já tiver uma chave SSH configurada, verá uma lista de arquivos, incluindo "id\_rsa" e "id\_rsa.pub".

**CASO VOCÊ JÁ TENHA UMA CHAVE SSH CONFIGURADA:**

Para adicionar sua chave SSH ao agente SSH no Git Bash, siga os seguintes passos:

1. Abra o Git Bash e certifique-se de estar na pasta do usuário. Quando você abre o Git Bash, ele geralmente é iniciado na pasta do usuário atual. Mas para confirmar que você está na pasta do usuário, você pode executar o comando **pwd** (print working directory) para exibir o diretório atual em que você está. Se você não estiver na pasta do usuário, você pode usar o comando **cd** (change directory) para navegar até a pasta do usuário. O comando **cd ~** irá levá-lo diretamente para a pasta do usuário.
2. Execute o seguinte comando para verificar se o agente SSH está sendo executado:

**$ eval "$(ssh-agent -s)"**

1. Se o agente SSH não estiver em execução, você poderá iniciá-lo com o seguinte comando:

**$ ssh-agent bash**

1. Adicione sua chave privada SSH ao agente SSH executando o seguinte comando:

**$ ssh-add ~/.ssh/nome\_da\_chave\_privada**

Certifique-se de substituir "nome\_da\_chave\_privada" pelo nome do arquivo que contém sua chave privada SSH.

O nome da chave privada pode variar dependendo de como você a nomeou durante a criação da chave. Por padrão, a chave privada SSH é geralmente nomeada como **id\_rsa**. Quanto ao formato do arquivo, o formato padrão de uma chave privada SSH é OpenSSH, que tem a extensão **.pem** ou **.ssh**. Por exemplo, se você criou uma chave privada com o nome **id\_rsa**, a chave privada SSH será armazenada em um arquivo chamado **id\_rsa** ou **id\_rsa.pem** ou **id\_rsa.ssh**.

Insira a senha da chave privada SSH quando solicitado.

Com isso, sua chave SSH deve ser adicionada ao agente SSH e você deve ser capaz de se autenticar em serviços remotos que exigem autenticação por SSH usando essa chave.

**CASO VOCÊ NÃO TENHA UMA CHAVE SSH CONFIGURADA:**

1. Digite o seguinte comando para gerar uma nova chave SSH:

**$ ssh-keygen -t rsa -b 4096 -C "seuemail@exemplo.com"**

O comando acima irá gerar duas chaves, uma pública e outra privada.

A chave privada será armazenada em **"/.ssh/id\_rsa"** e a chave pública em **"/.ssh/id\_rsa.pub"**.

Digite o seguinte comando para visualizar sua chave pública:

**$ cat ~/.ssh/id\_rsa.pub**

Copie a saída completa.

1. Acesse o seu perfil no GitHub e clique em "Settings" (Configurações). Em seguida, clique em "SSH and GPG keys" (Chaves SSH e GPG).
2. Clique em "New SSH key" (Nova chave SSH), dê um nome descritivo para sua chave e cole a chave pública na caixa de texto.
3. Clique em "Add SSH key" (Adicionar chave SSH). O GitHub agora está configurado para permitir que você se conecte usando sua chave SSH.

Pronto! Agora você configurou uma chave SSH para o GitHub e pode se conectar de forma mais segura e fácil. Certifique-se de nunca compartilhar sua chave privada com ninguém e de protegê-la com uma senha forte.

SALVAR PRIMEIRA VERSÃO DE UM PROJETO NO GITHUB

Considerando que agora seu ambiente já está todo configurado (usuário e email, chave SSH), sempre que você criar um novo projeto, os passos básicos serão estes:

1. Abra um terminal do GitBash na pasta do projeto clicando com o botão direito do mouse em “**GitBash Here**”;
2. O comando git init é usado para inicializar um repositório Git vazio no diretório atual. Quando você executa este comando, o Git cria uma nova subpasta oculta .git que contém todos os arquivos necessários para controlar as versões do seu projeto. Esta pasta oculta armazena o histórico de alterações, o índice de mudanças, configurações do projeto, entre outros arquivos necessários para o funcionamento do Git. Depois de executar **git init**, você pode começar a adicionar arquivos ao repositório com o comando git add e criar novos commits com o comando git commit.

**$ git init**

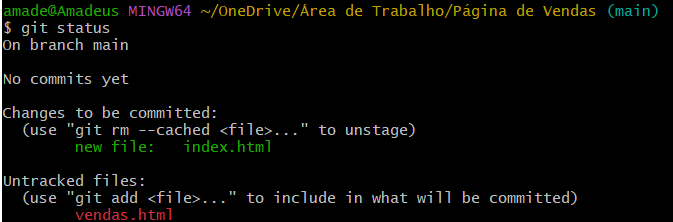
1. O comando "git add" é usado para adicionar arquivos e/ou alterações específicas em um repositório do Git. Ele prepara as alterações para serem incluídas no próximo commit. Quando você cria ou modifica um arquivo em seu repositório Git, essas alterações não são automaticamente incluídas no próximo commit. É necessário informar ao Git quais alterações devem ser incluídas no commit. O comando "**git add**" faz exatamente isso, ele adiciona as alterações especificadas ao index do Git, também conhecido como área de preparação ou staging area. Para adicionar todas as alterações de uma vez, você pode usar o comando "**git add .**" ou "**git add -A**". Para adicionar um arquivo específico, basta digitar o comando "**git add**" seguido pelo caminho do arquivo. Por exemplo, "git add arquivo.txt". Após usar o comando "git add" para adicionar as alterações, você pode então usar o comando "git commit" para criar um novo commit com essas alterações incluídas.

**$ git add .**

**$ git add -A**

**$ git add (caminho do arquivo)**

1. O comando "**git status**" é um comando que exibe o estado atual do seu repositório Git. Quando você executa este comando em um repositório Git, ele irá listar os arquivos que foram modificados, adicionados ou removidos desde o último commit. O resultado do comando "git status" também mostrará informações sobre o branch atual em que você está trabalhando e se há mudanças que ainda não foram adicionadas ao index. Além disso, ele também pode mostrar informações sobre quais arquivos foram criados ou excluídos que ainda não foram adicionados ao repositório. O "git status" é geralmente um dos primeiros comandos que você executará ao trabalhar com o Git, pois ele ajuda a entender o status atual do seu repositório e quais ações precisam ser tomadas antes de confirmar as mudanças.



1. O comando "**git commit -m**" é usado no Git para salvar as alterações que foram feitas em um arquivo ou conjunto de arquivos em um repositório Git. A parte "**-m**" significa "mensagem" e é usada para adicionar uma mensagem curta e descritiva que descreve as alterações feitas no commit. A mensagem deve ser informativa o suficiente para que outras pessoas que olharem para o histórico de commits do projeto possam entender rapidamente o que foi alterado. Por exemplo, se você tivesse feito algumas mudanças em um arquivo chamado "exemplo.txt" e quisesse fazer um commit dessas alterações, você poderia usar o seguinte comando:

**$ git commit -m "Adicionei uma nova linha ao arquivo"**

Isso criaria um novo commit no histórico do repositório com a mensagem "*Adicionei uma nova linha ao arquivo*" para que outros colaboradores possam entender o que foi alterado no projeto.

1. "Branch" em português significa "ramificação" ou "galho". No contexto do Git, a tradução de "branch" seria "ramo" ou "galho", referindo-se à criação de uma nova linha de desenvolvimento a partir da branch atual. No Git, uma "branch" é uma linha de desenvolvimento que permite que você trabalhe em diferentes versões do seu código fonte simultaneamente. Você pode pensar em uma branch como um caminho separado do seu projeto principal, que permite que você experimente novos recursos, faça correções de bugs ou trabalhe em recursos sem afetar o código na branch principal. Quando você cria um novo branch no Git, ele é baseado na branch atual em que você está. Isso significa que todo o histórico de commits (as alterações feitas no código) da branch atual é compartilhado com o novo branch, mas a partir desse ponto, as alterações feitas em cada branch são independentes uma da outra. O Git permite criar quantas branches forem necessárias para o desenvolvimento do seu projeto e, em seguida, mesclá-las quando o trabalho estiver concluído ou quando as alterações em uma branch estiverem prontas para serem incorporadas à branch principal. Com o comando "git branch -M main", você pode renomear a branch atual para "main". O parâmetro "-M" significa "move/rename", ou seja, ele move a branch atual para um novo nome especificado.

**$ git branch -M main**

1. Crie um novo repositório no site do github e depois execute o comando abaixo. O comando "git remote add" é usado para adicionar um novo repositório remoto ao seu repositório Git local. No exemplo dado, o comando está adicionando um novo repositório remoto chamado "origin" ao repositório Git local. O endereço "[git@github.com](mailto:git@github.com):Wagner-Amadeus/aula\_github.git" é o URL do repositório remoto no GitHub que está sendo adicionado como origin. Quando você executa este comando, ele associa esse URL ao nome "origin". Depois de executar esse comando, você poderá enviar (push) suas alterações para o repositório remoto do GitHub usando o comando "git push origin". Além disso, você também pode puxar (pull) as alterações do repositório remoto para o repositório Git local usando o comando "git pull origin".

**$ git remote add origin git@github.com:Wagner-Amadeus/aula\_github.git**

(esse endereço é fornecido apenas quando é criado o repositório no site do github)

1. Para verificar o endereço que o nome "origin" está apontando no seu repositório Git, você pode utilizar o seguinte comando abaixo. Esse comando irá exibir a URL do repositório remoto configurado com o nome "origin" no seu repositório local. Se você tiver configurado vários repositórios remotos no seu repositório local, pode usar o mesmo comando com o nome de outro repositório remoto no lugar de "origin" para verificar sua URL correspondente.

**$ git remote get-url origin**

1. O comando abaixo é usado para enviar (push) as alterações do ramo (branch) "main" do seu repositório local para o repositório remoto configurado com o nome "origin" e definir o upstream (ponto de referência) desse ramo. Mais especificamente, o parâmetro "-u" ou "--set-upstream" instrui o Git a configurar o ramo "main" no repositório remoto "origin" como o upstream (ponto de referência) do ramo "main" no repositório local. Isso significa que, depois de executar esse comando, sempre que você usar o comando "git push" no ramo "main", o Git saberá para onde enviar as alterações e qual ramo remoto deve ser atualizado. Por padrão, o Git espera que você envie as alterações para um ramo remoto com o mesmo nome do ramo local. Nesse caso, o comando está enviando as alterações do ramo "main" do repositório local para o ramo "main" do repositório remoto "origin". Esse comando é geralmente executado uma vez no início de um novo projeto para definir o upstream do ramo principal. A partir daí, você pode usar o comando "git push" para enviar as alterações do ramo "main" sem precisar especificar o nome do ramo remoto toda vez.

**$ git push -u origin main**

1. O comando "git push" é usado para enviar (push) as alterações locais do seu repositório Git para um repositório remoto. Por padrão, quando você executa o comando "git push" sem especificar um ramo específico, ele envia as alterações do ramo atual para o ramo correspondente no repositório remoto configurado com o nome "origin". Isso significa que, se você está atualmente no ramo "main" do seu repositório local e deseja enviar as alterações desse ramo para o repositório remoto "origin", basta executar o comando "git push". Se você quiser enviar as alterações de um ramo específico, pode especificar o nome do ramo como um argumento. Por exemplo, para enviar as alterações do ramo "develop" para o repositório remoto "origin", você pode executar o comando "git push origin develop".

**$ git psuh**

CLONAR E MODIFICAR UM PROJETO DE UM REPOSITÓRIO REMOTO QUE VOCÊ TEM PERMISSÃO PARA ALTERAR

1. O comando "git clone" é usado para criar uma cópia local de um repositório Git remoto. Ele permite que você faça o download de todo o histórico de commits e arquivos do repositório em seu próprio computador, para que você possa trabalhar com ele localmente. A sintaxe básica do comando git clone é a seguinte:

**$ git clone <***URL do repositório remoto***>**

Ao executar esse comando, o Git irá baixar o repositório remoto para o diretório atual de trabalho. Você também pode especificar um diretório de destino diferente usando a opção "-o":

**$ git clone <***URL do repositório remoto***> <***diretório de destino***>**

Este comando irá clonar o repositório remoto no diretório especificado como destino. Uma vez que o repositório tenha sido clonado, você pode trabalhar com ele localmente e realizar operações como commit, push, pull, merge e outras operações Git normalmente.

1. O comando "code ." é usado para abrir o Visual Studio Code a partir do diretório atual de trabalho no seu terminal. Quando você executa esse comando, o VS Code será iniciado no diretório atual e uma nova janela será aberta com o conteúdo do diretório. Isso permite que você trabalhe com os arquivos do diretório no VS Code sem precisar navegar até ele manualmente através do VS Code. Uma vez que o VS Code está aberto, você pode criar, editar e salvar arquivos, bem como executar tarefas e extensões diretamente do VS Code. O comando "code ." é uma maneira rápida e fácil de começar a trabalhar com o VS Code em um projeto específico.

**$ code .**

1. O comando "git log" é usado para exibir o histórico de commits de um repositório Git. Ele lista os commits em ordem cronológica inversa, mostrando o commit mais recente primeiro. A sintaxe básica do comando "git log" é a seguinte:

**$ git log**

Ao executar esse comando, o Git irá exibir uma lista de todos os commits no repositório, incluindo o hash do commit, autor, data e hora do commit, e a mensagem de log do commit. Por padrão, a saída é exibida em ordem cronológica inversa, com o commit mais recente listado primeiro. Além disso, o comando "git log" suporta várias opções que permitem filtrar e personalizar a saída. Por exemplo, você pode usar a opção "--author" para filtrar os commits por autor:

**$ git log --author="Nome do autor"**

Você também pode usar a opção "--oneline" para exibir uma lista compacta de commits em uma única linha:

**$ git log --oneline**

GIT STATUS

O comando "git status" é usado para verificar o estado dos arquivos em um repositório Git local. Ele exibe informações sobre quais arquivos foram modificados, quais estão em espera para serem adicionados e quais foram excluídos. Os três estados principais que o "git status" pode mostrar são: "modified", "untracked" e "deleted".

1. "**Modified**": Quando um arquivo já existente em seu repositório Git é alterado, o Git o reconhece como um arquivo "modificado". Isso significa que o arquivo foi alterado, mas ainda não foi adicionado ao próximo commit. Quando você executa "git status", o Git exibirá uma lista de arquivos que foram modificados desde o último commit.
2. "**Untracked**": Quando você adiciona um novo arquivo ao seu diretório de trabalho, ele é considerado um arquivo "não rastreado". O Git não controla automaticamente esses arquivos e não os inclui em seus commits, a menos que você os adicione explicitamente usando o comando "git add". Quando você executa "git status", o Git exibe uma lista de arquivos não rastreados.
3. "**Deleted**": Quando você exclui um arquivo do seu diretório de trabalho, o Git o reconhece como "excluído". Isso significa que o arquivo ainda existe em seu histórico de commits, mas foi removido do diretório de trabalho. Quando você executa "git status", o Git exibe uma lista de arquivos que foram excluídos.

GIT DIFF

O comando "git diff" é usado para mostrar as diferenças entre o estado atual do repositório Git e um commit anterior ou entre dois ramos diferentes. Ele mostra as alterações em cada arquivo no repositório, destacando as linhas que foram adicionadas, removidas ou modificadas.

Este comando mostra as diferenças entre o estado atual do arquivo e a versão anterior:

**$ git diff arquivo**

Para comparar dois ramos diferentes, você pode usar o comando:

**$ git diff branch1..branch2**

Este comando mostra as diferenças entre o ramo "branch1" e o ramo "branch2". Ele exibe as alterações que foram feitas em um ramo, mas não no outro.

GIT CHECKOUT

O comando "git checkout" é usado no Git para alternar entre diferentes ramos (branches) ou commits no repositório. Sua sintaxe básica é a seguinte:

**$ git checkout <nome-do-ramo>**

ou

**$ git checkout <hash-do-commit>**

Ele permite que você mude o estado de trabalho do repositório Git para uma versão anterior ou para outro ramo, fazendo com que o HEAD (ponteiro para a última confirmação) aponte para um commit ou ramo diferente.

Quando usado para alternar para um ramo específico, o comando "git checkout" atualiza o estado de trabalho para refletir o estado do último commit no ramo selecionado. Isso significa que todos os arquivos no diretório de trabalho e na área de staging (index) são atualizados para refletir o estado do ramo selecionado.

Além disso, o comando "git checkout" também pode ser usado para criar e alternar para novos ramos a partir de um commit específico, usando a opção "-b". Por exemplo, o comando:

**$ git checkout -b new-branch 123456**

Cria um novo ramo chamado "new-branch" a partir do commit com o hash "123456" e muda o HEAD para esse novo ramo.

O "git checkout" também pode ser usado para alternar para um commit específico, usando seu hash ou referência simbólica, em vez de um nome de ramo. Isso permite que você explore versões anteriores do seu código e crie novos ramos a partir de commits específicos.

Em um sistema de controle de versão, cada commit é identificado por um código exclusivo, como um hash SHA-1. Essa identificação pode ser usada para referenciar um commit específico em vários contextos, como para revertê-lo ou mesclá-lo em outro branch.

Além disso, o commit mais recente em um branch pode ser referenciado pela palavra "HEAD". Isso é útil quando se deseja, por exemplo, criar um novo branch a partir do último commit ou verificar o status atual do branch.

Por fim, é possível referenciar commits anteriores em relação ao HEAD usando a sintaxe ~1 refere-se ao commit imediatamente anterior ao HEAD, enquanto HEAD~2 refere-se ao commit dois commits anteriores ao HEAD. Essa sintaxe é útil para navegar pelo histórico de commits e para fazer alterações em commits anteriores.

**IMPORTANTE**: antes de fazer o checkout para voltar para HEAD, certifique-se de que não haja mudanças nos arquivos. Se você acidentalmente mudou alguma coisa, desfaça as modificações usando:

**$ git reset**

**$ git clean -df**

**$ git checkout -- .**

.GITIGNORE

O arquivo .gitignore é usado para especificar quais arquivos ou pastas devem ser ignorados pelo Git durante o processo de versionamento. Isso significa que esses arquivos ou pastas não serão adicionados ao repositório Git e não serão rastreados por alterações.

Aqui está um exemplo de um arquivo .gitignore que especifica que arquivos com extensões ".log" e pastas chamadas "node\_modules" devem ser ignorados:

**# arquivos que devem ser ignorados pelo Git**

**\*.log**

**# pastas que devem ser ignoradas pelo Git**

**node\_modules/**

Nesse exemplo, o Git não irá rastrear arquivos com a extensão ".log" e a pasta "node\_modules". Isso é útil, por exemplo, quando se trabalha em projetos Node.js, onde a pasta "node\_modules" pode conter muitos arquivos e pastas que não são necessários para o versionamento do código-fonte. Ao usar o arquivo .gitignore, é possível evitar que esses arquivos e pastas desnecessários sejam adicionados ao repositório e, assim, reduzir o tamanho do repositório e melhorar a eficiência do processo de versionamento.

REMOVENDO ARQUIVOS DA ÁREA DE STAGE

O comando "git reset" é usado para desfazer as alterações em um commit específico ou para remover arquivos da área de staging (index) do Git. Existem três formas diferentes de usar o comando "git reset":

1. **Desfazer um commit:** o comando "git reset <commit>" pode ser usado para desfazer um commit específico. Isso significa que o commit será removido do histórico do Git e as alterações feitas nesse commit serão desfeitas. No entanto, as alterações nos arquivos permanecerão no diretório de trabalho. Para desfazer um commit e manter as alterações no diretório de trabalho, use o comando "git reset --soft <commit>".
2. **Remover arquivos da área de staging (index):** o comando "git reset <file>" pode ser usado para remover um arquivo específico da área de staging (index) do Git. Isso significa que o arquivo não será incluído no próximo commit.
3. **Desfazer todas as alterações:** o comando "git reset --hard" pode ser usado para desfazer todas as alterações no diretório de trabalho e na área de staging (index) do Git. Isso significa que todas as alterações feitas desde o último commit serão perdidas e o diretório de trabalho será restaurado para o estado do último commit.

É importante lembrar que o comando "git reset" pode ser um comando perigoso, pois pode causar a perda permanente de dados. Portanto, é recomendável fazer backup dos dados antes de usar esse comando.