Aula 6 – Comandos Básicos e Ciclo de Vida do Git

Você já se viu trabalhando em um projeto, fazendo várias alterações e, de repente, percebe que uma delas quebrou tudo? Ou talvez você precise voltar para uma versão anterior do seu código, mas não sabe como, ou pior, não tem um registro claro do que foi feito? Essa é uma dor comum para desenvolvedores, designers e qualquer pessoa que lida com arquivos digitais em constante evolução. A boa notícia é que existe uma ferramenta poderosa que resolve esses problemas, e ela é a base para a colaboração e a eficiência em equipes de desenvolvimento modernas.

Nesta aula, vamos mergulhar no universo do Git, a ferramenta de controle de versão distribuído mais utilizada no mundo. Entenderemos não apenas como ele funciona, mas por que ele se tornou indispensável no dia a dia de qualquer profissional de tecnologia. Ao final, você não só conhecerá os comandos essenciais, mas também compreenderá a lógica por trás deles, permitindo que você gerencie seus projetos com muito mais segurança e controle.

Nosso objetivo é que você seja capaz de iniciar um repositório Git, acompanhar as mudanças nos seus arquivos, registrar essas mudanças de forma organizada e navegar pelo histórico do seu projeto. Vamos desvendar o ciclo de vida dos arquivos no Git e praticar os comandos que farão de você um usuário confiante. Prepare-se para dar um salto na sua capacidade de gerenciar projetos e colaborar de forma eficaz.

A Necessidade de um Controle de Versão: Uma Jornada no Tempo

Imagine que você está escrevendo um trabalho importante para a faculdade ou desenvolvendo um software complexo. Você cria o arquivo, faz algumas edições, salva. Depois, faz mais edições, salva novamente. Em algum momento, você pode querer experimentar uma ideia nova, mas tem medo de estragar o que já está funcionando. O que você faz? Provavelmente, cria cópias do arquivo com nomes como "trabalho_final_v1.docx", "trabalho_final_v2_com_ideia_nova.docx", "trabalho_final_final_mesmo.docx". Isso soa familiar, não é?

Essa abordagem, embora comum, é um verdadeiro pesadelo para a organização e a colaboração. Fica difícil saber qual é a versão mais recente, quais mudanças foram feitas em cada uma e, principalmente, como voltar a um ponto específico sem perder o trabalho subsequente. Para equipes, o problema se agrava exponencialmente, com cada membro criando suas próprias cópias e a fusão de trabalhos se tornando uma tarefa hercúlea e propensa a erros.

> 💡 **O Git como Máquina do Tempo**
>
> É aqui que entra o Git, como um verdadeiro "máquina do tempo" para seus projetos. Ele não apenas salva versões do seu trabalho, mas registra quem fez o quê, quando e por quê. Pense nele como um sistema de backup inteligente e colaborativo, que permite que você experimente, desfaça, refaça e colabore sem medo de perder o progresso ou criar confusão.

Ele se tornou a espinha dorsal do desenvolvimento moderno, permitindo que equipes distribuídas trabalhem em sincronia e com total rastreabilidade.

Desvendando o Ciclo de Vida dos Arquivos no Git

Para entender como o Git gerencia as mudanças, precisamos primeiro compreender o "ciclo de vida" de um arquivo dentro de um repositório Git. Não é apenas sobre "salvar" um arquivo, mas sobre como o Git o vê em diferentes estágios de preparação para ser registrado permanentemente. Essa compreensão é fundamental para usar o Git de forma eficaz e evitar surpresas.

> **Analogia da Cozinha**
>
> Imagine que seu projeto é uma cozinha. Quando você começa a cozinhar, os ingredientes estão na despensa (fora do controle do Git). Quando você os tira para usar, eles estão na bancada (o Git os vê, mas ainda não os "preparou"). Ao temperar e cortar, você os está modificando. Quando você os coloca em uma tigela para serem misturados e cozidos, eles estão "preparados" para a próxima etapa. Finalmente, quando o prato está pronto e servido, ele é uma "versão" final.

No Git, os arquivos passam por quatro estados principais: **Untracked**, **Unmodified**, **Modified** e **Staged**. Cada um desses estados representa um momento diferente na jornada de um arquivo, desde sua criação até seu registro no histórico do projeto. Entender essa transição é a chave para dominar o fluxo de trabalho básico do Git.

Os Quatro Estados de um Arquivo Git

Vamos detalhar cada um desses estados, que são como as etapas de um processo de produção, garantindo que cada mudança seja intencional e rastreável.

Untracked (Não Monitorado)

Este é o estado de um arquivo que o Git ainda não conhece. Você acabou de criar um novo arquivo no seu projeto, mas ainda não disse ao Git para começar a monitorá-lo. É como um novo ingrediente que você trouxe para a cozinha e ainda está na sacola de compras, fora da sua bancada de trabalho. O Git sabe que ele existe, mas não o inclui em seu sistema de controle de versão.

Unmodified (Não Modificado)

Um arquivo neste estado está sendo monitorado pelo Git e não foi alterado desde o último *commit* (registro permanente). Ele é idêntico à versão que está armazenada no histórico do seu repositório. Pense nisso como um ingrediente que você já usou em um prato anterior e que está guardado na geladeira, exatamente como foi armazenado.

Modified (Modificado)

Este estado indica que um arquivo monitorado pelo Git foi alterado desde o último *commit*. Você abriu um arquivo existente e fez edições nele. É como pegar um ingrediente da geladeira e começar a cortá-lo ou temperá-lo; ele não é mais o mesmo que estava guardado. O Git detecta essa mudança, mas ainda não a preparou para ser salva.

Staged (Preparado)

Um arquivo *staged* é um arquivo modificado que foi marcado para ser incluído no próximo *commit*. É a "área de preparação" do Git. Você decidiu que as alterações feitas neste arquivo estão prontas para serem salvas como parte de uma nova versão. Voltando à analogia da cozinha, é o ingrediente que, depois de cortado e temperado, foi colocado na tigela junto com os outros, pronto para ir ao forno.

Esses estados formam um ciclo contínuo, onde você cria arquivos (Untracked), os adiciona ao controle (tornando-os Unmodified após o primeiro commit), os modifica (Modified), os prepara (Staged) e, finalmente, os salva no histórico (voltando a Unmodified para a próxima rodada de mudanças).

Iniciando um Repositório: O Primeiro Passo com git init

Agora que entendemos os estados dos arquivos, vamos colocar a mão na massa com os comandos essenciais. O primeiro passo para usar o Git em qualquer projeto é inicializar um repositório. Isso transforma uma pasta comum em um local onde o Git pode começar a rastrear e gerenciar suas mudanças.

Imagine que você está começando um novo diário. Antes de escrever qualquer coisa, você precisa comprar o diário em si, certo? É um objeto físico que você vai usar para registrar suas memórias. O comando `git init` faz exatamente isso: ele "compra o diário" para o seu projeto, criando a estrutura interna necessária para o Git funcionar.

Ao executar `git init` dentro de uma pasta, o Git cria um subdiretório oculto chamado `.git`. É dentro dessa pasta `.git` que toda a mágica acontece: o Git armazena o histórico do seu projeto, as configurações, os objetos de versão e tudo o mais que ele precisa para funcionar. Você não precisa interagir diretamente com essa pasta, mas é importante saber que ela existe e que é o coração do seu repositório. Sem ela, sua pasta é apenas uma pasta comum, sem controle de versão.

> **📁 Pasta .git**
>
> O coração do seu repositório. Toda a mágica do Git acontece aqui!

Exemplo Prático

Vamos criar um novo projeto. Abra seu terminal ou prompt de comando e siga os passos:

01

Crie uma nova pasta para o seu projeto

```
mkdir meu_primeiro_projeto_git
```

02

Entre na pasta

```
cd meu_primeiro_projeto_git
```

03

Inicialize o repositório Git

```
git init
```

Você verá uma mensagem como `Initialized empty Git repository in /caminho/para/meu_primeiro_projeto_git/.git/`. Isso significa que seu diário foi comprado e está pronto para ser usado!

Verificando o Status: O Olhar Atento do `git status`

Com o repositório inicializado, como sabemos o que o Git está vendo? Como identificamos quais arquivos estão Untracked, Modified ou Staged? Para isso, usamos o comando `git status`. Ele é seu melhor amigo no dia a dia com o Git, fornecendo um resumo claro do estado atual do seu repositório.

> 🔍 **Seu Check-list do Git**
>
> Pense no `git status` como um "check-list" ou um "relatório de progresso" da sua cozinha. Ele te diz quais ingredientes estão na despensa (Untracked), quais estão na bancada e foram cortados (Modified), e quais estão na tigela prontos para o forno (Staged). Ele é essencial para você saber exatamente onde está no fluxo de trabalho do Git antes de tomar a próxima ação.

Este comando é crucial para manter a organização e evitar que você comite arquivos indesejados ou esqueça de incluir alterações importantes. Ele te dá a visibilidade necessária para decidir o que fazer em seguida: adicionar arquivos novos, preparar mudanças existentes ou finalizar um conjunto de alterações.

Exemplo Prático

Continuando com nosso projeto, vamos criar um arquivo e ver o que o `git status` nos diz:

Crie um arquivo README.md

```
echo "# Meu Primeiro Projeto Git" > README.md
```

Verifique o status do repositório

```
git status
```

Você verá algo como:

```
On branch master (ou main)
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    README.md
nothing added to commit but untracked files present (use "git add" to track)
```

> ✅ **Interpretação**
>
> O Git nos informa que `README.md` é um arquivo **Untracked**. Ele sabe que o arquivo existe, mas ainda não o está monitorando ativamente.

Preparando as Mudanças: O Poder do `git add`

Depois de criar ou modificar um arquivo, o próximo passo é dizer ao Git que você quer incluir essas mudanças no próximo *commit*. É aqui que entra o comando `git add`. Ele move os arquivos do estado **Modified** (ou **Untracked**) para o estado **Staged** (preparado).

Voltando à nossa analogia da cozinha, o `git add` é como pegar os ingredientes que você cortou e temperou na bancada (Modified) e colocá-los na tigela de mistura (Staged). Você está preparando-os para a próxima etapa, que é o cozimento (o *commit*). Você pode adicionar arquivos individualmente ou todos de uma vez, dando a você controle granular sobre o que será incluído na sua próxima "versão".

> ⚠️ **Importante**
>
> `git add` não salva as mudanças permanentemente no histórico do Git. Ele apenas as prepara.

É importante notar que `git add` não salva as mudanças permanentemente no histórico do Git. Ele apenas as prepara. Você pode usar `git add` várias vezes para adicionar diferentes conjuntos de mudanças à área de *staging* antes de fazer um único *commit*. Isso permite que você organize suas mudanças em blocos lógicos, tornando seu histórico mais limpo e fácil de entender.

Exemplo Prático

Vamos adicionar o arquivo `README.md` à área de *staging*:

Adicione o README.md

```
git add README.md
```

Verifique o status novamente

```
git status
```

Agora, a saída será diferente:

```
On branch master (ou main)
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   README.md
```

O Git agora mostra `README.md` em "Changes to be committed", indicando que ele está no estado **Staged**. Ele está pronto para ser salvo no histórico.

Registrando as Mudanças: O Compromisso do git commit

Com as mudanças preparadas na área de *staging*, o momento de registrar essas alterações permanentemente no histórico do seu projeto chegou. Este é o papel do comando `git commit`. Um *commit* é como uma "fotografia" do seu projeto em um determinado momento, acompanhada de uma mensagem que descreve o que foi feito.

> 📸 **Servindo o Prato**
>
> Pense no `git commit` como o ato de "servir o prato" na nossa analogia da cozinha. Depois de preparar todos os ingredientes na tigela (Staged), você os cozinha e o prato está pronto. Você o serve e, talvez, tira uma foto e anota a receita no seu livro. Essa anotação é o *commit* – um registro imutável do estado do seu projeto naquele ponto, com uma descrição do que foi criado.

Cada *commit* possui um identificador único (um hash SHA-1), o nome do autor, a data e hora, e a mensagem do *commit*. Essas informações são cruciais para a rastreabilidade e para entender a evolução do projeto. Boas mensagens de *commit* são concisas, descritivas e explicam o "porquê" da mudança, não apenas o "o quê".

Exemplo Prático

Vamos fazer nosso primeiro *commit*:

1 Faça o *commit* das mudanças preparadas

```
git commit -m "feat: Adiciona README inicial ao projeto"
```

A flag `-m` permite que você forneça a mensagem do *commit* diretamente na linha de comando. Você verá uma saída detalhando o *commit*, como:

```
[master (root-commit) 1234567] feat: Adiciona README inicial ao projeto
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

2 Verifique o status novamente

```
git status
```

A saída agora será:

```
On branch master (ou main)
nothing to commit, working tree clean
```

Isso significa que não há arquivos modificados ou preparados; tudo está salvo e o repositório está "limpo". O arquivo `README.md` agora está no estado **Unmodified** em relação ao último *commit*.

Visualizando o Histórico: A Linha do Tempo com git log

Com vários *commits* feitos, como podemos ver o histórico do nosso projeto? Como revisamos as "fotos" que tiramos ao longo do tempo? O comando `git log` é a ferramenta para isso. Ele exibe uma lista cronológica de todos os *commits* feitos no repositório, do mais recente ao mais antigo.

Pense no `git log` como folhear as páginas do seu diário ou livro de receitas. Cada página é um *commit*, e você pode ver o que foi registrado em cada uma, quem escreveu e quando. É a sua janela para a história do projeto, permitindo que você entenda como ele evoluiu, quais problemas foram resolvidos e quais funcionalidades foram adicionadas.

O `git log` oferece diversas opções para personalizar a saída, permitindo que você veja o histórico de diferentes maneiras. Por exemplo, você pode ver um resumo mais conciso, um gráfico de ramificações, ou filtrar *commits* por autor ou data. Essa flexibilidade é crucial para navegar em projetos grandes e complexos.

> 📚 **Seu Diário de Projeto**
>
> Cada *commit* é uma página na história do seu código.

Exemplo Prático

Vamos fazer mais algumas alterações e *commits* para ter um histórico mais rico:

01

Modifique o README.md

```
echo "Este é um projeto de exemplo para a aula de Git." >> README.md
```

02

Crie um novo arquivo index.html

```
echo "<h1>Olá, Git!</h1>" > index.html
```

03

Verifique o status

```
git status
```

Você verá `README.md` como **Modified** e `index.html` como **Untracked**.

04

Adicione e comite ambos os arquivos

```
git add .
git commit -m "feat: Adiciona index.html e atualiza README"
```

O `git add .` adiciona todos os arquivos modificados e não monitorados no diretório atual à área de *staging*.

05

Agora, visualize o histórico

```
git log
```

Você verá algo como:

```
commit 789abcdef (HEAD -> master)
Author: Seu Nome <seu.email@exemplo.com>
Date:   Mon Jan 1 10:00:00 2025 -0300

    feat: Adiciona index.html e atualiza README

commit 1234567 (master)
Author: Seu Nome <seu.email@exemplo.com>
Date:   Mon Jan 1 09:00:00 2025 -0300

    feat: Adiciona README inicial ao projeto
```

Cada entrada representa um *commit*, com seu hash, autor, data e mensagem.

Entendendo o HEAD: Onde Você Está Agora

Ao olhar a saída do `git log`, você notou a palavra **HEAD** ao lado do *commit* mais recente? O HEAD é um ponteiro muito importante no Git. Ele indica o *commit* atual em que você está trabalhando. Em termos simples, ele aponta para a "ponta" da sua ramificação atual, ou seja, o último *commit* que você fez ou para o qual você navegou.

📍 O Marcador de Página

Imagine que o HEAD é o seu marcador de página em um livro. Ele sempre aponta para a página que você está lendo no momento. Se você virar a página (fizer um novo *commit*), o marcador se move para a nova página. Se você decidir voltar para uma página anterior para reler algo, o marcador se move para lá. O HEAD é essa referência dinâmica que o Git usa para saber qual é a versão do código que está carregada no seu diretório de trabalho.

🎯 Posição Atual

Normalmente, o HEAD aponta para a ponta da sua ramificação atual (como `master` ou `main`). No entanto, é possível mover o HEAD para um *commit* anterior, o que é útil para inspecionar versões antigas do código ou para desfazer mudanças. Compreender o HEAD é fundamental para operações mais avançadas, como viajar no tempo no histórico do Git ou trabalhar com ramificações.

Exemplo Prático

Na saída do `git log` anterior:

```
commit 789abcdef (HEAD -> master)
```

Isso significa que o HEAD está apontando para o *commit* `789abcdef`, e este *commit* é a ponta da ramificação `master`. Se você fizer outro *commit*, o HEAD e a ramificação `master` se moverão para o novo *commit*.

Prática Guiada: Criando e Versionando um Projeto Local

Agora, vamos consolidar tudo o que aprendemos com uma prática guiada, simulando um pequeno projeto. O objetivo é que você se sinta confortável com o fluxo básico: inicializar, criar/modificar, adicionar, comitar e inspecionar.

> 🎯 **Cenário**
>
> Você está criando um site simples e precisa versionar seus arquivos.

Crie um novo diretório para o projeto e inicialize o Git

```
mkdir meu_site_simples
cd meu_site_simples
git init
```

Verifique o status: `git status` (deve estar limpo, sem commits).

Crie o arquivo index.html

```
echo "<!DOCTYPE html><html><head><title>Meu Site</title></head><body><h1>Bem-vindo!</h1>
</body></html>" > index.html
```

Verifique o status: `git status` (deve mostrar `index.html` como **Untracked**).

Adicione index.html à área de *staging* e faça o primeiro *commit*

```
git add index.html
git commit -m "feat: Cria página inicial index.html"
```

Verifique o status: `git status` (deve estar limpo).

Adicione um arquivo de estilo style.css

```
echo "body { font-family: sans-serif; background-color: #f0f0f0; }" > style.css
```

Verifique o status: `git status` (deve mostrar `style.css` como **Untracked**).

Modifique index.html para linkar o CSS

Abra `index.html` em um editor de texto e adicione a linha `<link rel="stylesheet" href="style.css">` dentro da tag `<head>`.

```
<!DOCTYPE html>
<html>
<head>
  <title>Meu Site</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <h1>Bem-vindo!</h1>
</body>
</html>
```

Verifique o status: `git status` (deve mostrar `style.css` como **Untracked** e `index.html` como **Modified**).

Prática Guiada (Continuação)

Continuando nossa prática, agora com arquivos em diferentes estados, vamos preparar e comitar as mudanças.

1

Adicione ambos os arquivos à área de *staging* e faça o segundo *commit*

```
git add .
git commit -m "feat: Adiciona estilo CSS e linka no index.html"
```

Verifique o status: `git status` (deve estar limpo).

2

Visualize o histórico do projeto

```
git log
```

Você deve ver dois *commits*, um para a criação do `index.html` e outro para a adição do `style.css` e a modificação do `index.html`. O HEAD estará apontando para o *commit* mais recente.

3

Crie um arquivo .gitignore

Imagine que você tem arquivos temporários ou de configuração que não devem ser versionados.

```
echo "temp/" > .gitignore
mkdir temp
echo "arquivo_temporario.txt" > temp/arquivo_temporario.txt
```

Verifique o status: `git status` (deve mostrar `.gitignore` como **Untracked**, mas `temp/arquivo_temporario.txt` não deve aparecer, pois o `.gitignore` o está ignorando).

4

Adicione e comite o .gitignore

```
git add .gitignore
git commit -m "chore: Adiciona .gitignore para ignorar pasta temp"
```

Verifique o status e o log novamente.

🎉 Parabéns!

Você acabou de completar um ciclo básico de trabalho com Git, desde a inicialização até a visualização do histórico, passando por todos os estados dos arquivos e os comandos essenciais. Essa base é o alicerce para operações mais complexas e para a colaboração em equipe.

Conectando com o Mundo Real: GitOps e DevSecOps

A maestria dos comandos básicos do Git não é apenas uma habilidade técnica; é a porta de entrada para metodologias avançadas que estão moldando o futuro do desenvolvimento e operações. As tendências de 2025, como GitOps e DevSecOps, dependem fundamentalmente de uma compreensão sólida do Git.

GitOps

GitOps é uma abordagem que usa o Git como a "única fonte da verdade" para a infraestrutura e as aplicações. Isso significa que todas as mudanças, sejam elas na configuração de um servidor ou no código de uma aplicação, são feitas através de *pull requests* no Git. O que você aprendeu sobre `git add` e `git commit` é a base para criar essas mudanças rastreáveis. A automação então entra em ação, aplicando essas mudanças ao ambiente real. A rastreabilidade do `git log` se torna vital para auditorias e para entender o estado de qualquer sistema a qualquer momento.

DevSecOps

Já o **DevSecOps** integra segurança em todas as etapas do ciclo de vida do desenvolvimento, "deslocando a segurança para a esquerda" (Shift-Left). O Git desempenha um papel crucial aqui, pois cada *commit* e cada *pull request* podem ser automaticamente verificados por ferramentas de segurança. A capacidade de inspecionar o histórico com `git log` e de reverter para versões anteriores com segurança é uma ferramenta poderosa para mitigar vulnerabilidades e garantir a conformidade. A adoção massiva dessas práticas mostra que o Git não é apenas uma ferramenta de controle de versão, mas um pilar estratégico para a agilidade, segurança e resiliência de sistemas modernos.

Quadro Comparativo: Estados do Arquivo no Git

Para solidificar a compreensão dos estados dos arquivos, veja um resumo conciso:

Conceito	Âmbito/Aplicação	Base/Origem	Exemplo
Untracked	Arquivos novos, ignorados ou não monitorados.	Arquivo existe no diretório, mas não no Git.	`novo_modulo.py` recém-criado.
Unmodified	Arquivos monitorados, idênticos ao último *commit*.	Versão atual do arquivo é a mesma do histórico.	`main.js` após um `git commit`.
Modified	Arquivos monitorados que foram alterados.	Conteúdo do arquivo difere do último *commit*.	`config.yaml` com novas configurações.
Staged	Arquivos modificados marcados para o próximo *commit*.	Alterações prontas para serem registradas.	`README.md` após `git add README.md`.

Quadro Comparativo: Comandos Essenciais do Git

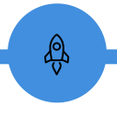
E para os comandos que você acabou de aprender:

Conceito	Âmbito/Aplicação	Base/Origem	Exemplo
git init	Inicializa um novo repositório Git.	Cria a pasta `.git` no diretório atual.	`git init` em uma nova pasta de projeto.
git status	Exibe o estado do diretório de trabalho e área de *staging*.	Lê o estado dos arquivos em relação ao histórico.	`git status` para ver arquivos modificados.
git add	Adiciona mudanças à área de *staging*.	Prepara arquivos para o próximo *commit*.	`git add meu_arquivo.txt` ou `git add .`
git commit	Registra as mudanças preparadas no histórico.	Cria um novo ponto no histórico do repositório.	`git commit -m "feat: Adiciona nova funcionalidade"`.
git log	Exibe o histórico de *commits*.	Lê os registros de *commits* no `.git`.	`git log --oneline` para um histórico conciso.
HEAD	Ponteiro para o *commit* atual da ramificação.	Referência simbólica para o *commit* ativo.	HEAD aponta para o último *commit* na `main`.

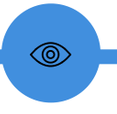
Consolidação e Próximos Passos

Nesta aula, desvendamos os mistérios do Git, começando pela necessidade de controle de versão e mergulhando no ciclo de vida dos arquivos. Você aprendeu a inicializar um repositório, a acompanhar o estado dos seus arquivos, a preparar e registrar suas mudanças, e a navegar pelo histórico do seu projeto. Esses comandos e conceitos são a base sólida para qualquer interação com o Git, seja em projetos pessoais ou em equipes complexas.

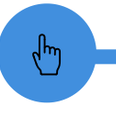
Em prática:



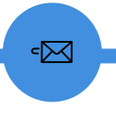
Sempre inicie seus projetos com git init



Use git status constantemente para saber onde você está



Seja seletivo com git add, preparando apenas as mudanças relacionadas



Escreva mensagens de git commit claras e descritivas

Consulte git log para entender a evolução do seu trabalho

A compreensão desses fundamentos é o que permite que você participe de projetos colaborativos, adote práticas como GitOps e DevSecOps, e gerencie seu código com confiança. O Git é mais do que uma ferramenta; é uma filosofia de trabalho que promove a organização, a rastreabilidade e a colaboração.

Autoavaliação

1

Qual dos seguintes comandos é utilizado para inicializar um novo repositório Git em um diretório existente?

- a) git start
- b) git new
- c) git init
- d) git create

2

Um arquivo que foi modificado, mas ainda não foi adicionado à área de *staging* (preparação), encontra-se em qual estado no ciclo de vida do Git?

- a) Untracked
- b) Unmodified
- c) Staged
- d) Modified

3

Qual comando permite visualizar o histórico de *commits* de um repositório Git?

- a) git history
- b) git show
- c) git log
- d) git timeline

4

O que o ponteiro HEAD representa no Git?

- a) O primeiro *commit* do repositório.
- b) O *commit* mais antigo da ramificação atual.
- c) O *commit* atual em que você está trabalhando.
- d) O *commit* que será enviado para o repositório remoto.

Gabarito:

1. c) **2.** d) **3.** c) **4.** c)

Questão Discursiva:

> ✍️ **Reflita**
>
> Explique a importância da área de *staging* (preparação) no fluxo de trabalho do Git, diferenciando-a do diretório de trabalho e do repositório local.

Próxima Aula e Recursos Adicionais

Próxima Aula

> 🚀 **Aula 7**
>
> Na Aula 7, vamos expandir nosso conhecimento sobre Git, explorando um de seus recursos mais poderosos e flexíveis: as **Ramificações (Branching) e Fusões (Merging)**. Você aprenderá como trabalhar em diferentes linhas de desenvolvimento simultaneamente e como integrar essas mudanças de volta ao projeto principal, essencial para a colaboração em equipe.

Recursos Adicionais

- **Documentação Oficial do Git**

  Para aprofundar nos comandos e conceitos.

- **Pro Git Book (online)**

  Um guia completo e gratuito sobre Git.

- **Git Immersion**

  Um tour guiado interativo para praticar os comandos.

NOTA IMPORTANTE: As informações técnicas desta aula estão atualizadas até 2025. Consulte sempre a documentação oficial do Git para verificar as versões mais recentes e possíveis alterações.