

O que é Git e Github: os primeiros passos nessas ferramentas



a

16/11/2020

COMPARTILHE



Esse artigo faz parte da
Formação DevOps

Confira neste artigo:

- [Antes de mais nada, um panorama geral](#)
- [História do Git](#)
- [O que é Git e para que serve?](#)
- [Qual é a linguagem do Git?](#)
- [Conceitos fundamentais do Git](#)
- [Quais são os principais comandos do Git?](#)
- [Quais os três objetos internos do git?](#)
- [Como o Git armazena as mudanças no repositório?](#)
- [Alterando commits \(reset, revert, --amend\)](#)
- [Modelos de Colaboração com Ramificações](#)
- [Qual a vantagem de utilizar o Git?](#)
- [O que é GitHub?](#)

- Conclusão

Pense na seguinte situação: você precisa gerenciar um projeto de desenvolvimento de software que envolve uma equipe em nível global de pessoas desenvolvedoras.

Imagina o desafio que é quando cada uma das pessoas contribui com uma parte do código?

Na Alura, por exemplo, o time de desenvolvimento é segmentado em diferentes funções: há uma equipe que desenvolve as interfaces visuais da plataforma e outra que processa toda a operação da plataforma.

Como podemos assegurar que todas as peças vão se encaixar perfeitamente? Como garantir um trabalho em conjunto harmonioso?

É aí que entram as ferramentas **Git** e **GitHub**, que, como você vai ver adiante, apresentam boas respostas para essas questões :)

Antes de mais nada, um panorama geral

Se você é dev ou está considerando entrar na área da tecnologia, é provável que já tenha se deparado com o termo "Git e Github".

À primeira vista, pode parecer que "Git" e "Github" são a mesma coisa, mas a resposta é **não**.

São ferramentas distintas, mas colaboram de maneira integrada para tornar o desenvolvimento de software mais eficiente.

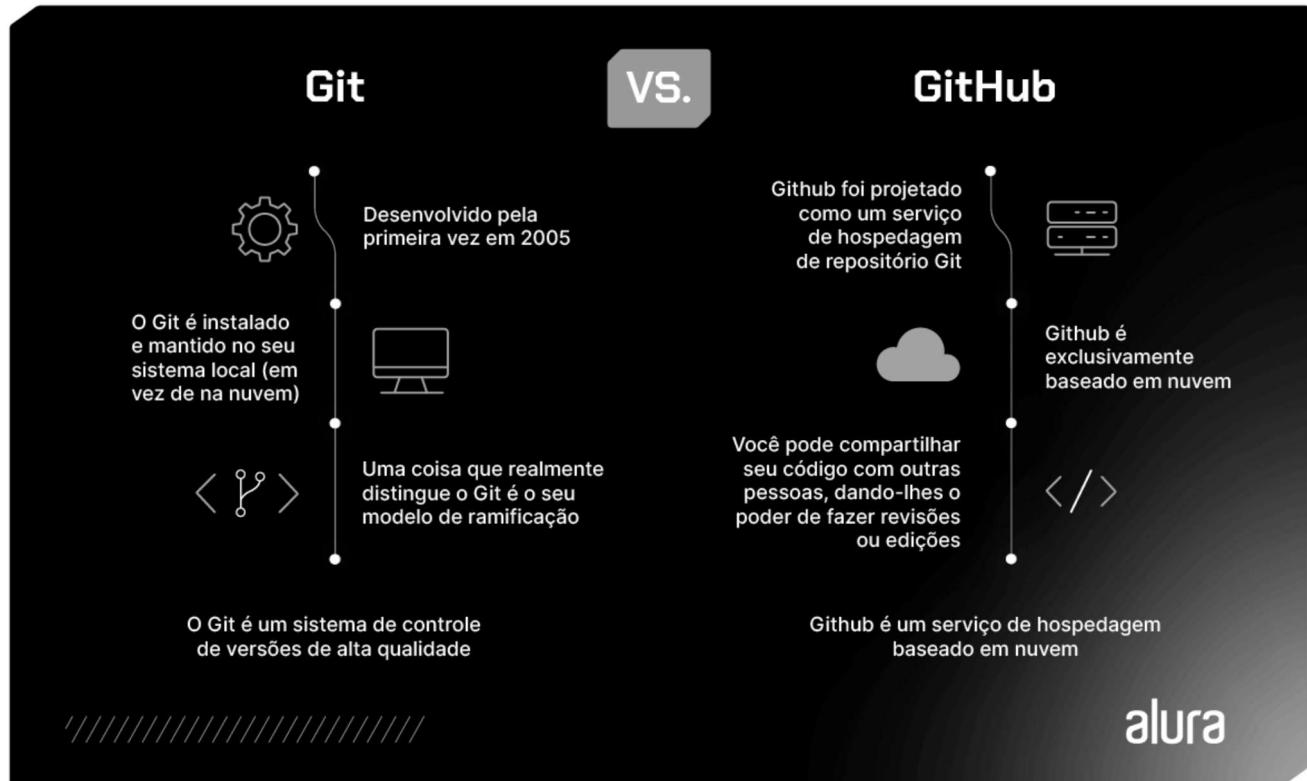
Aqui está um resumo rápido sobre cada uma delas — que também nos ajuda a diferenciá-las:

*O **Github** é uma “rede social dev” em que é possível armazenar e compartilhar projetos de desenvolvimento de software.*

*O **Git** é um sistema de controle de versão de arquivos; em outras palavras, é responsável por guardar o histórico de alterações sempre que alguém modificar algum arquivo que está sendo monitorado por ele.*

Desta maneira, o Git e o GitHub são pilares fundamentais que auxiliam as equipes de desenvolvimento a controlar o versionamento de código, rastrear mudanças, colaborar de forma eficiente e garantir que o trabalho em equipe flua sem problemas.

Na imagem a seguir, podemos exemplificar melhor essas diferenças:



Ao longo deste artigo, vamos nos aprofundar sobre os serviços dessas duas tecnologias.

Quer desvendar os segredos do Git e GitHub? Então vamos lá :)



História do Git

Durante o período de 1991 a 2002 aconteceu a maior parte da manutenção do projeto de código aberto do núcleo do Linux. Diferentes pessoas eram responsáveis por essas mudanças e compartilhavam tarefas e arquivos.

Assim, em 2002, perceberam a necessidade de usar uma ferramenta de controle de versão distribuído (DVCS).

Na época, a ferramenta era a “BitKeeper” — um software proprietário, lançado sobre a licença Apache 2.0.

Foi então que, em 2005, a relação entre a comunidade de dev do Linux e a empresa por trás do BitKeeper ficou tensa. Vix! O resultado disso foram diversas restrições e altos custos para usar a ferramenta.

Essa situação levou a comunidade, e especialmente Linus Torvalds, o criador do Linux, a criar sua própria ferramenta de controle de versão, o Git, que ganhou o coração das pessoas que trabalham com open source.

Desde então, o Git passou por melhorias e se tornou uma ferramenta relativamente fácil de usar, mantendo suas características originais, como velocidade, eficiência em projetos grandes e um sistema de ramificação para o desenvolvimento não linear.

Para conferir mais sobre, segue a documentação do Git, no capítulo 1.2

O que é Git e para que serve?

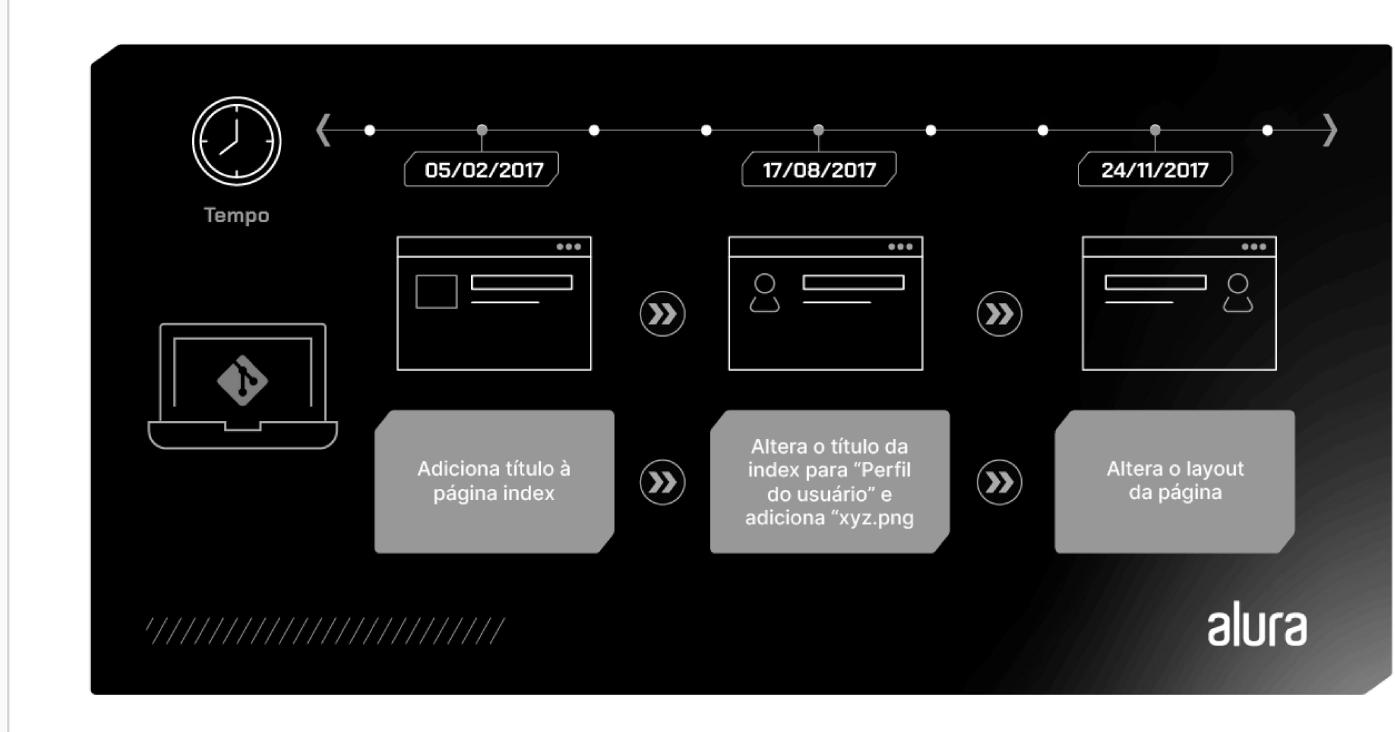
Em uma equipe, apenas poder acessar o código de outras pessoas colaboradoras não é suficiente.

Mais do que isso: precisamos manter o **histórico** dos nossos arquivos e das nossas modificações.

Afinal de contas, muitas vezes mudamos arquivos em grupo, num movimento único — que, no contexto do Git, é um **commit**. O que, em tradução literal para português, significa “compromisso” ou “comprometer-se” às alterações em um repositório.

Dessa forma, podemos voltar atrás e **recuperar o estado do sistema**: como ele era ontem, ou no ano passado, comparar as mudanças para encontrar bugs e estudar otimizações.

Vamos conferir na imagem abaixo um exemplo de histórico de um projeto de desenvolvimento.



Na imagem acima há destaque para 3 *commits*:

- 1 - “Adicionando título a página index”: Neste primeiro commit percebemos que o dev responsável criou a estrutura inicial de uma página de perfil de usuário.
- 2 - “Altera o título da index para Perfil do usuário e adiciona xyz.png”: Neste segundo commit notamos que alteraram o título da página e acrescentaram uma nova imagem.
- 3 - “Altera o layout da página”: Neste terceiro commit, observamos que alteraram o layout da página de usuário, cores e posições de elementos. Com base nestes commits, se porventura o cliente não gostar da mudança de layout implementada no commit 3, o time desenvolvimento pode voltar o layout como era no commit 2.

Além disso, podemos identificar que cada um desses commits contam alguma história, a partir do seu motivo de criação.

Isso auxilia bastante quem está lendo a identificar o que está procurando.

Como funciona o Git?

Mas como será que o Git guarda todas essas informações?

Todos os nossos arquivos, assim como seus históricos, ficam em um **repositório** e existiam vários sistemas que gerenciam repositórios assim, como CVS (Sistema de Versões Concorrentes) e SVN (Subversion do Apache).

O Git é uma alternativa com um funcionamento mais interessante ainda: ele é **distribuído** e todo mundo tem uma cópia inteira do **repositório**, não apenas o "servidor principal".

E uma das vantagens disso é que cada pessoa pode desenvolver offline, realizando seus commits e outras operações sem depender de uma conexão constante com o servidor principal.

Mas...o que é a ferramenta Git exatamente?

O Git é um sistema de controle de versão distribuído e amplamente adotado. O Git nasceu e foi tomando espaço dos outros sistemas de controle.

[O que são Git e Github? #HipstersPontoTube](#)

https://www.youtube.com/watch?v=P4BNi_yPehc

Como baixar e instalar o Git?

Para utilizar o Git, você pode seguir esse passo a passo de como fazer o download e instalar, para cada sistema operacional:

Windows:

1. Acesse o site oficial do Git em "<https://git-scm.com/download/win>".
2. Clique no link para download do Git para Windows.
3. Após o download, execute o instalador.
4. Siga as instruções do instalador, aceitando as configurações padrão, se não for um usuário avançado.
5. Conclua a instalação.

Linux:

1. No Linux, você pode instalar o Git usando o gerenciador de pacotes da sua distribuição. Por exemplo, no Ubuntu, use o comando `sudo apt-get install git`.
2. Se estiver usando outra distribuição, substitua o comando de acordo.

macOS:

1. No macOS, o Git pode ser instalado de várias maneiras, incluindo o uso do Xcode Command Line Tools, que geralmente já está instalado no sistema.
2. Abra o Terminal e digite `git --version` para verificar se o Git está disponível. Se não estiver, o sistema solicitará a instalação.

3. Siga as instruções para instalar o Git. Com esses passos simples, você pode instalar o Git no seu sistema operacional e começar a usar essa poderosa ferramenta de controle de versão.

Como executar o Git?

Depois de instalar o Git, você pode começar a usar o Git em um projeto. É só seguir essas etapas:

- **1. Configure seu nome de usuário e e-mail:**

- O Git registra quem fez cada alteração no código. Portanto, é importante configurar seu nome de usuário e e-mail. Use os comandos, no terminal:

```
git config --global user.name "Seu Nome"  
git config --global user.email "seu@email.com"
```

- **2. Crie um Repositório Git:**

- Para começar a rastrear seu código, crie um repositório Git em seu projeto. Navegue até a pasta do seu projeto e execute:

```
git init
```

- **3. Adicione Arquivos ao Controle de Versão:**

- Use o comando `git add` para adicionar arquivos ao "staging area", que é onde você prepara os arquivos para serem "commitados" ou salvos.

```
git add nome-do-arquivo
```

- **4. Faça um Commit:**

- Um commit é um snapshot de suas alterações. Use o comando `git commit` para criar um commit com uma mensagem descritiva do que foi alterado no projeto.

```
git commit -m "Sua mensagem de commit aqui"
```

- **5. Visualize o Histórico de Commits:**

- Use `git log` para ver o histórico de commits no repositório.

```
git log
```

Essas são as etapas fundamentais para começar a usar o Git. Com esses comandos, você pode iniciar o controle de versão de seu código e colaborar em projetos com outras pessoas desenvolvedoras.

Qual é a linguagem do Git?

Desde seu lançamento, a comunidade de desenvolvimento gradualmente passou a adotá-lo, especialmente devido à sua robustez como sistema de gerenciamento de versões e outras características — como ser rápido e distribuído.

Por este motivo, muitas pessoas que desenvolvem se perguntam em algum momento: "**Do que é feito o Git?**"

Inicialmente, por ser desenvolvido com o Linux em mente como plataforma, o Git foi desenvolvido em Shell Script que, apesar de funcionar como o esperado, amarrava a ferramenta a sistemas Linux, que tinham os utilitários necessários para interpretar o Shell Script.

Com a popularidade da ferramenta, outros sistemas buscaram dar suporte a ela, através da emulação de um sistema Linux, que ficava responsável por executar o Git.

No entanto, o uso da emulação de um sistema Linux tinha impacto na performance da ferramenta nos sistemas operacionais que usavam essa estratégia.

Tendo isso em mente, muitos comandos do Git, inicialmente escritos em Shell Script, foram reescritos na [linguagem C](#), que resultou em ganho de performance em plataformas que não usam o Shell Script como linguagem de linha de comando oficial, como é o caso do Windows.

Conceitos fundamentais do Git

Para o funcionamento do Git, existem conceitos com nomenclaturas um tanto diferentes, que são:

Repositórios, commits e árvores (Trees)

Um **repositório** é como uma pasta ou diretório que contém todos os arquivos e o histórico de um projeto.

Já o termo **commit** pode ter como tradução literal “compromisso”, que seria uma ação em que você faz uma alteração no projeto, se compromete e salva suas alterações no histórico do projeto.

Ou seja, cada commit é uma entrada no histórico que contém informações sobre as alterações feitas.

Árvores, por último, representam a estrutura do diretório e arquivos em um commit específico, que tem como função registrar a organização do projeto ao longo do histórico de desenvolvimento.

Ramificações (Branches) e fusões (Merges)

As “**ramificações**” ou **branches** permitem que você crie linhas separadas de desenvolvimento para trabalhar em recursos ou correções sem afetar a linha principal do projeto.

Cada branch é uma ramificação independente do código-fonte, possibilitando que você isole e desenvolva novas funcionalidades, refatore o código ou faça correções e testes em paralelo, sem interferir no código existente na branch principal, que geralmente é nomeada como “main”.

Em um projeto com branches diferentes, a fusão, ou merge, permite combinar as alterações dessas branches de volta à linha principal, quando as alterações estão prontas.

Controle de versão distribuído

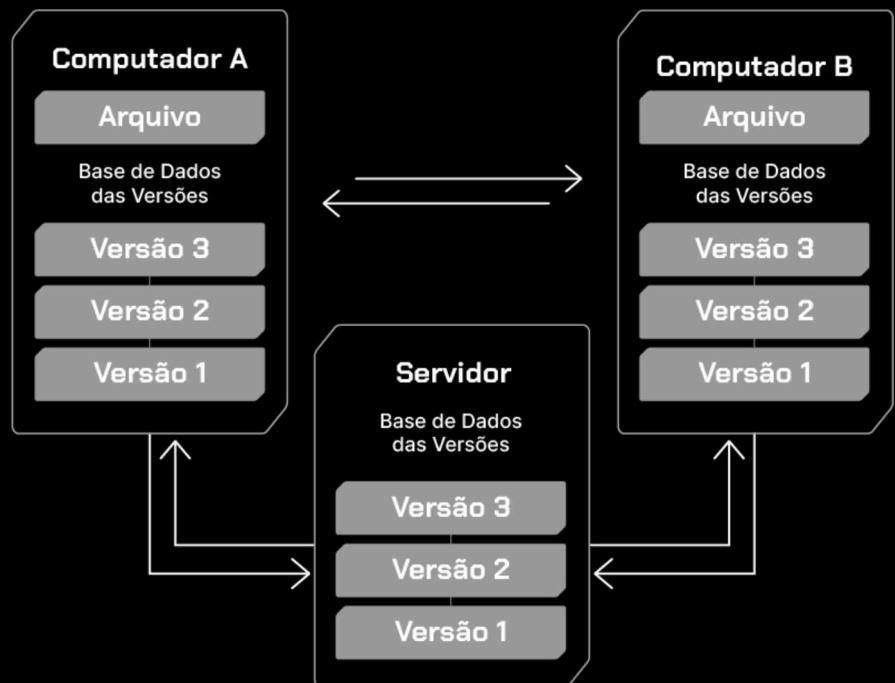
Existem dois tipos de sistemas de controle de versão. Em um deles, pode haver um único servidor central que armazena o projeto com seu histórico, com o qual as pessoas desenvolvedoras precisam interagir. Isso é característico de um sistema de Controle de Versão Centralizado.

No outro tipo, cada pessoa desenvolvedora pode manter uma cópia do projeto em sua máquina local, o que é conhecido como Controle de Versão Distribuído, que é o caso do Git.

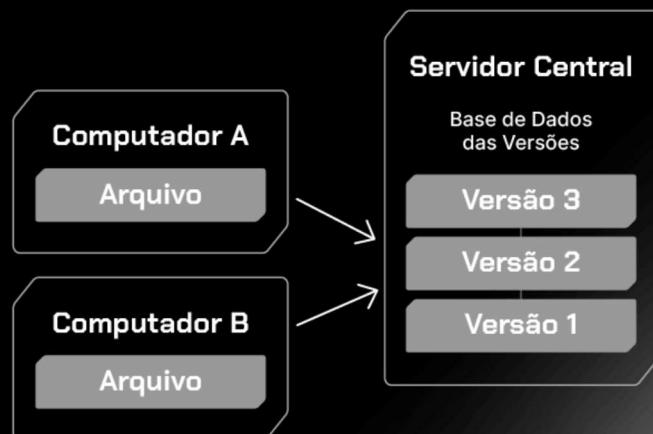
Com o Git, cada pessoa desenvolvedora tem uma versão completa do histórico do projeto. Isso proporciona independência e permite o desenvolvimento em paralelo.



Modelo Distribuído



Modelo Centralizado



alura

Quais são os principais comandos do Git?

O Git é uma ferramenta bastante robusta e oferece diversos utilitários para gerenciar as versões de um projeto em linha de comando. Confira os **principais comandos da**

ferramenta:

- Git init;
- Git clone;
- Git status;
- Git add;
- Git commit;
- Git log;
- Git branch;
- Git checkout;
- Git diff.

1) Git init

É utilizado para **inicializar um repositório Git dentro de um diretório do sistema**. Após sua utilização, a ferramenta passa a monitorar o estado dos arquivos no projeto.

2) Git clone

É utilizado para **criar uma cópia de um repositório remoto em um diretório da máquina**. Este repositório poder ser criado a partir de um repositório armazenado localmente, através do caminho absoluto ou relativo, ou pode ser remoto, através do URI na rede.

A partir de um repositório clonado, é possível acompanhar o estado de um projeto e suas modificações, além de contribuir com o projeto, a partir do envio das suas modificações ao repositório central.

3) Git status

É utilizado para **verificar o status de um repositório git**, bem como o estado do repositório central. O comando mostra informações sobre se o projeto local está sincronizado com o central, quais arquivos estão sendo monitorados pelo Git e em qual branch você está no projeto.

4) Git add

É utilizado para **adicionar arquivos ao pacote de alterações a serem feitas**. É possível adicionar um único arquivo, múltiplos arquivos de uma vez, como `git add <-arquivo1-> <-arquivo2->`

> ... , ou até mesmo um diretório, a partir de seu caminho. Uma vez que um arquivo é adicionado ao pacote de alterações com o comando `add` , ele está pronto para entrar no próximo **commit**.

5) Git commit

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

É utilizado para **criar uma nova versão do projeto a partir de um pacote de alterações**. O commit pega o pacote de modificações adicionado através do comando `git add` , fecha essas alterações num pacote e o identifica através de um Hashcode.

Além disso, para cada commit é necessário escrever uma mensagem para identificá-lo, com uma mensagem clara de quais alterações foram feitas neste commit.

6) Git log

```
git log
```

É utilizado para **ver o histórico de alterações do projeto**, onde aparecerão todos os commits feitos, com suas respectivas mensagens e códigos identificadores.

O comando é muito útil quando precisamos rastrear o andamento de um projeto e verificar em qual ponto cada funcionalidade foi implementada.

Além disso, o comando conta com várias opções para mostrar o histórico de forma resumida, gráfica e até mesmo mostrando a diferença entre os commits, que podem ser vistas na [documentação oficial do comando](#).

7) Git branch

É utilizado para **criar novos ramos de desenvolvimento**, bem como visualizar quais são os ramos existentes.

Para criar um novo ramo, basta utilizar o comando `git branch` seguido do nome do novo ramo, e para visualizar quais os ramos existentes a utilização do comando é bem similar: basta não informar um nome para a nova branch, e serão listadas todas as já criadas.

8) Git checkout

É utilizado para **navegar entre as versões do projeto**, bem como entre as diferentes ramificações criadas. Para navegar entre as versões, basta usar o comando:

```
git checkout <- Hashcode do commit ->
```

E todo o estado do projeto se modificará ao estado no qual o commit foi feito.

Similarmente, para navegar entre as ramificações podemos usar o comando:

```
git checkout <- nome da branch ->
```

E a branch será alterada. O comando também permite criar uma branch e imediatamente mudar para ela, através do comando:

```
git checkout -b <- nome da branch ->
```

Que vai criar a ramificação e navegar até ela.

9) Git diff

É utilizado para **visualizar modificações feitas entre commits**, sejam eles entre um commit arbitrário e o estado atual do projeto, dois commits arbitrários, ou até mesmo todas alterações entre dois commits distintos.

Para visualizar as alterações entre um commit distinto e o atual, basta usar o comando:

```
git diff <- Hashcode do commit anterior ->
```

E serão listadas todas as diferenças no projeto entre os dois commits.

Para conhecer mais sobre o comando `git diff` e seus casos de uso, além de outros comandos e utilitários do Git, confira a [documentação do Git](#) e o [curso de Git e Github](#) da Alura.

10. Git config

11.

O comando `git config` é usado para configurar e personalizar o ambiente Git no seu sistema. Ele permite que você defina informações como seu nome de usuário, endereço de e-mail, editor padrão e muitas outras configurações que definem como o Git interage com seus repositórios.

A estrutura básica do comando é:

```
git config <opções> chave valor
```

- `<opções>` : Pode ser global (`--global`) para definir configurações para todos os repositórios no seu sistema ou local (`--local`) para definir configurações específicas para um repositório em particular.
- `chave` : A chave de configuração que você deseja definir (por exemplo, `user.name` para o nome de usuário).
- `valor` : O valor que você deseja atribuir à chave (por exemplo, seu nome de usuário ou endereço de e-mail).

Por exemplo, para configurar seu nome de usuário globalmente, você pode usar o comando:

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

Isso é útil para garantir que todos os commits que você fizer em qualquer repositório Git no seu sistema tenham o seu nome associado a eles.

Além disso, o comando `git config` pode ser usado para personalizar muitos outros aspectos do seu ambiente Git, tornando-o mais adaptado às suas preferências e necessidades.

Quais os três objetos internos do git?

Internamente, o Git cria **conjuntos de dados e metadados** para armazenar o histórico de um projeto monitorado pela ferramenta.

A esses conjuntos de dados damos o nome de **objetos git**, e eles podem ser de **três tipos**:

1. **Blobs**;
2. **Trees**;

3. Commit.

1) Blobs

São objetos criados para **armazenar dados de arquivos**, porém não guardam seus metadados.

Na prática, a partir do conteúdo de um arquivo é gerado um Hashcode de identificação para ele, que será usado para guardar seu estado em um determinado ponto.

Caso duas pessoas diferentes criem arquivos com exatamente o mesmo conteúdo, o Git criará um blob idêntico para os dois arquivos, pois ele só se baseia no conteúdo do arquivo e não guarda metadados sobre quem criou o arquivo ou quando.

2) Trees

São objetos criados para **armazenar dados de pastas**, como **blobs** e até mesmo outras **trees**, podem ser entendidos como a representação de uma pasta dentro do git. Similar ao blob, a tree não guarda metadados e gera um Hashcode de identificação baseado em seu conteúdo.

3) Commit

São objetos que **guardam o snapshot de um momento do projeto**. Dentro de um commit são guardadas trees e blobs, que por sua vez identificam o estado dos arquivos e pastas no momento em que o commit é criado, assim como metadados como quando ele foi criado e por quem.

O Hashcode que identifica um commit é justamente o que aparece ao utilizar o comando `git log`, e é essencial para controlar as versões do projeto.

Como o Git armazena as mudanças no repositório?

Quando você inicia um repositório Git com o comando `git init`, uma pasta oculta chamada `.git` é criada na raiz do projeto.

Essa pasta é o cérebro por trás do Git, onde todas as informações sobre as versões, histórico de commits e configurações são armazenadas.

A pasta `.git` contém três componentes principais:

- 1. Diretório de objetos:** Esta pasta armazena os objetos Git, que representam os commits, árvores (arquivos e diretórios) e blobs (conteúdo dos arquivos). Os objetos são criados a partir das alterações que você salva por meio de commits.
- 2. Diretório de referências:** Aqui são armazenadas as informações sobre as branches e tags, que indicam as posições atuais no histórico do Git. Isso permite que o Git saiba qual commit é o mais recente e a que branch, ou ramificação, está vinculado.
- 3. Diretório de configuração:** Este diretório contém arquivos de configuração que definem as preferências do Git, que são definidas com o comando `git config`, como o nome de usuário, o email e opções de comportamento. Ao executar comandos Git, esses componentes são atualizados na pasta `.git`, garantindo que todas as mudanças e histórico sejam armazenados.

Dessa forma, o Git é capaz de recuperar qualquer versão anterior do seu projeto e rastrear cada alteração feita ao longo do tempo.

Alterando commits (`reset`, `revert`, `--amend`)

Em desenvolvimento de software, é bem comum lidarmos com bugs e ver a necessidade de voltar algumas alterações e, isso envolvendo o Git, não é diferente.

Então, podemos se deparar com a necessidade de fazer ajustes em commits anteriores ou desfazer mudanças já registradas.

Existem três comandos principais que permitem essas operações: `reset`, `revert` e `--amend`. Vamos dar uma olhada em como eles funcionam:

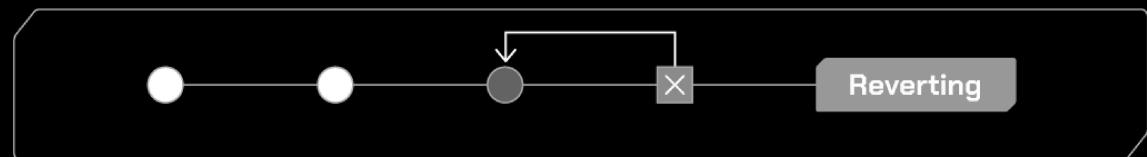
1. Reset: Desfazendo Commits

O comando `git reset` permite desfazer commits anteriores e mover a branch para um commit anterior. Para isso, você precisa especificar o commit para onde deseja reverter.

Exemplo:

```
git reset HEAD~1
```

Neste caso, `HEAD~1` significa "um commit antes do último commit". Isso moverá a branch para um commit atrás, desfazendo o último commit mais recente



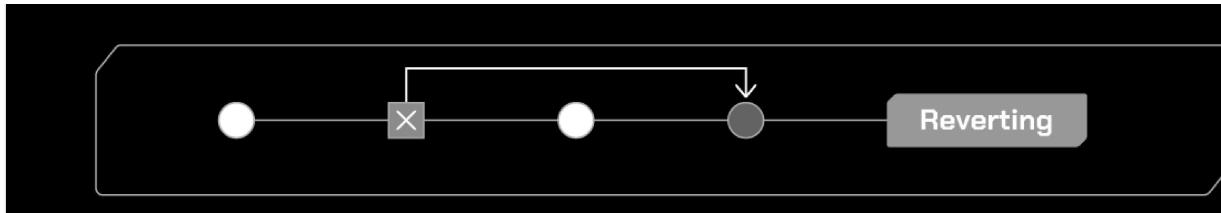
2. Revert: Criando Novos Commits de Reversão

O comando `git revert` é usado para criar um novo commit que desfaz as alterações de um commit específico. Em vez de remover o commit original, ele adiciona um novo commit de reversão ao histórico.

Exemplo:

```
git revert HEAD~1
```

Isso criará um novo commit que desfaz as alterações introduzidas pelo commit anterior.



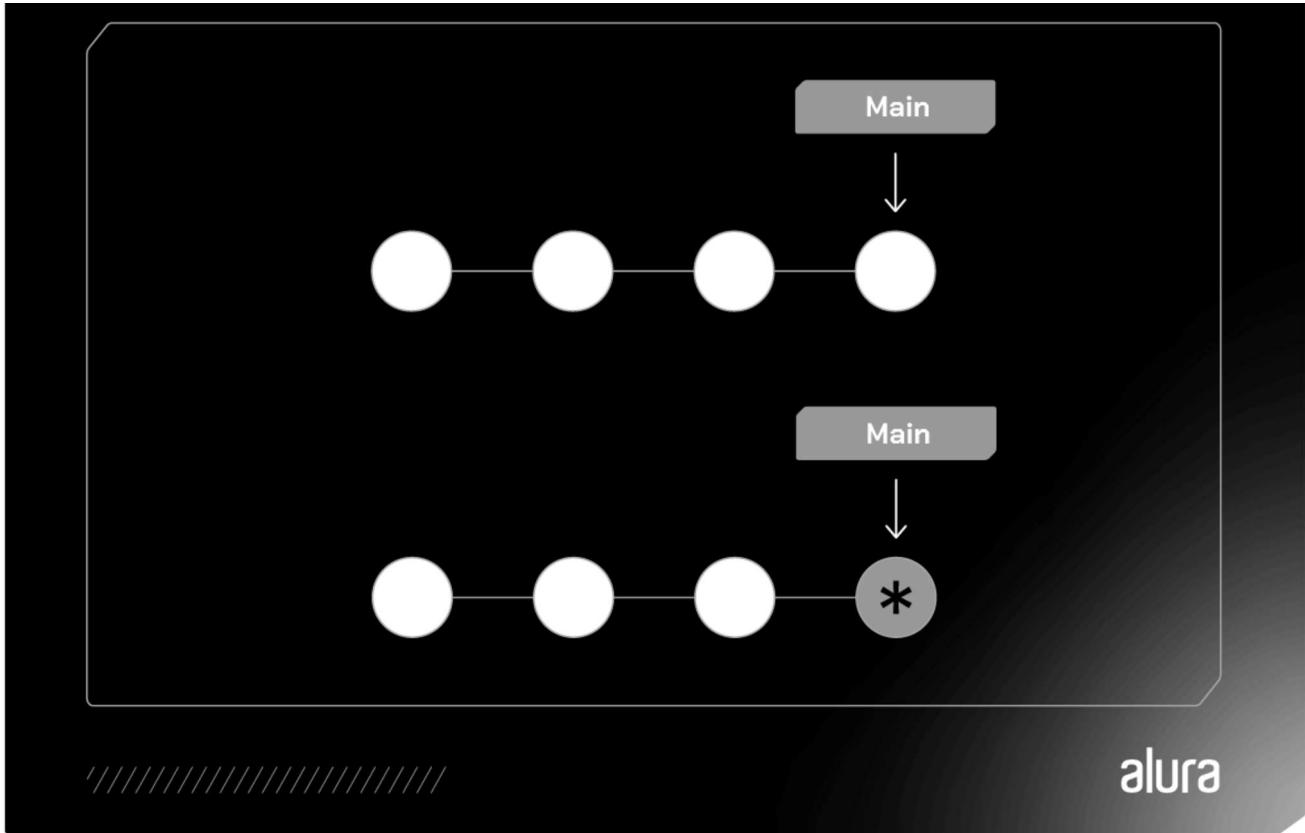
3. --amend: Modificando o Último Commit

O comando `git commit --amend` permite fazer modificações no último commit, como adicionar ou editar mensagens de commit ou incluir arquivos esquecidos.

Exemplo:

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

Isso altera a mensagem do último commit.



Esses comandos são úteis para ajustar o histórico do seu projeto, mas lembre-se de que devem ser usados com cuidado, especialmente quando você já compartilhou seu trabalho com outros desenvolvedores.

Compreender a posição do HEAD, que aponta para o commit atual, é fundamental para evitar conflitos em seu repositório Git.

Modelos de Colaboração com Ramificações

Todas essas funcionalidades que vimos sobre o Git se justificam porque diversas organizações utilizam essa ferramenta como sistema de controle de versões.

Por isso, conhecimentos sobre o Git são comumente encontrados em requisitos de vagas de emprego no ramo de desenvolvimento de software.

É comum que haja projetos que utilizam ramificações diferentes para criar novas features, fazer testes, correções de bugs e outras atividades.

Mas como será que essas ramificações são organizadas? Para isso, existem dois modelos de gestão de ramificações do Git que são o Git Flow e o Trunk-Based Development.

Gitflow: Um modelo estruturado para colaboração

O Gitflow é um modelo de colaboração que oferece uma estrutura bem definida para gerenciar as ramificações do Git em projetos de desenvolvimento de software.

Ele é projetado para facilitar a coordenação de diferentes tipos de tarefas com as seguintes branches:

- **Feature:** ramos específicos para novos recursos;
- **Develop:** é a branch onde fica o código do próximo deploy;
- **Hotfix:** onde ocorre as correções de bugs;
- **Release:** ramo para lançamentos de versões;
- **Main:** branch principal onde fica todo o código de produção

As ramificações são criadas e fundidas de acordo com as etapas específicas do ciclo de vida do software. Isso ajuda a evitar conflitos, mantendo as diferentes tarefas bem separadas e permitindo revisões de código apropriadas antes da mesclagem.

Para ler mais sobre e conferir como é feita sua implementação, confira o artigo [Git Flow: entenda o que é, como e quando utilizar] (<https://www.alura.com.br/artigos/git-flow-o-que-e-como-quando-utilizar>)

Trunk-Based Development: Simplicidade e Agilidade

O Trunk-Based Development é um modelo mais simples que se concentra na manutenção de uma única branch principal, que pode ser chamada de trunk, ou tronco, e é utilizada para todo o desenvolvimento contínuo do projeto.

Neste modelo, todas as alterações, sejam elas novos recursos, correções de bugs ou melhorias, são desenvolvidas diretamente na branch principal.

A justificativa para o modelo Trunk-Based Development é a garantia de agilidade e integração contínua.

No entanto, para garantir que as alterações não tragam muitos conflitos para o projeto, é fundamental fazer uso extensivo de testes automatizados e diversas práticas de revisão de código.

Ambos os modelos, Gitflow e Trunk-Based Development, têm suas vantagens e desvantagens, e a escolha depende das necessidades e da estrutura de cada projeto.

Para conferir mais sobre os dois modelos, veja o vídeo [Git Flow vs Desenvolvimento baseado em tronco](#).

https://www.youtube.com/watch?v=0jw8RpHuZ-Q&ab_channel=Alura

Qual a vantagem de utilizar o Git?

Com o acesso facilitado a computadores, trabalhar com arquivos tornou-se uma forte alternativa às pilhas de papéis, afinal, ao guardar as **informações** em memória, menos espaço físico é ocupado.

As grandes salas de arquivos com toneladas de papéis passaram a ser substituídas por computadores e servidores com grande capacidade de armazenamento.

Assim, múltiplas pessoas passaram a acessar um mesmo arquivo através de um servidor central, responsável por arquivar os documentos.

O acesso de muitas pessoas a um arquivo, no entanto, podia ser problemático. Imagine que três pessoas estão trabalhando no mesmo projeto e precisam editar o mesmo arquivo, cada uma em sua parte de responsabilidade.

Se as pessoas tentarem fazer alterações ao mesmo tempo, poderiam ocorrer duas coisas:

- o arquivo ser sobreescrito e perder informações do trabalho de uma pessoa; ou
- o arquivo ficar bloqueado para edição a partir do momento em que a primeira pessoa começar a editá-lo (apesar de não perder informações, prejudica a produtividade da equipe).

Além disso, não se tinha controle sobre quem era responsável por cada alteração, pois não era mantido um **histórico** de alterações do documento.

Com isso em mente, passaram a surgir **ferramentas para controlar as versões de um projeto**, que eram capazes de manter todo o histórico das alterações nos arquivos, bem como as pessoas responsáveis por elas, como o CVS e o Subversion.

Dessa maneira, além de ter o controle do histórico de um projeto, era possível reverter alterações que introduziram funcionalidades indesejadas ao projeto, ou voltar a um estado do projeto onde existia um arquivo que não existe mais na versão atual.

Apesar de já fornecer o controle do histórico de versões, estas ferramentas costumavam centralizar o controle de versão, ou seja, ferramentas que controlavam o versionamento do estado de um repositório central, em um servidor.

Sendo assim, existia uma vulnerabilidade no modelo: caso existisse alguma falha no servidor, não seria possível utilizar o controle de versão enquanto a falha existisse, e no caso de um

disco corrompido, isso poderia significar a perda de todo o histórico, caso não existisse nenhum backup.

Com isso, surgiram os sistemas de gerenciamento de versões distribuídos. Nestes, além de existir um **repositório central em servidor**, cada pessoa terá uma cópia do repositório em sua máquina, e através do **repositório local** poderá comunicar-se com os dados armazenados no servidor.

Com esse modelo, é mais fácil restaurar o estado de um servidor defeituoso, pois cada cliente tem uma cópia exata do repositório central, e basta copiá-la para o servidor para restaurá-lo.

Além disso, permite trabalhar com diversos repositórios remotos, o que aumenta as possibilidades de **fluxo de trabalho com diferentes grupos de pessoas**.

Em vista disso, podemos considerar que o Git é um sistema de gerenciamento de versões distribuído (SGVD).

Contudo, por que o Git se popularizou tanto? E outras alternativas como o Mercurial e o Helix Core não?

O Git se tornou bastante popular por, além de ser um SGVD, se tratar de uma ferramenta de **código aberto gratuita**, e ser fácil de começar a utilizar.

Além disso, trata-se de uma ferramenta muito veloz, devido à sua arquitetura, suporta desenvolvimento não-linear, com milhares de ramificações e funciona muito bem com projetos grandes, afinal, foi pensado para suportar o controle de versões do Linux, um dos sistemas operacionais mais adotados no mundo.

Ainda assim, talvez o principal motivo do Git ser tão adotado pela comunidade dev, desde devs iniciantes até experientes, seja a existência da **plataforma GitHub**.

A plataforma, que permite compartilhamento de código através da criação de repositórios, se tornou muito forte dentro da comunidade do open-source, devido à facilidade de compartilhar e contribuir em projetos abertos.

Além de ser um sistema completamente distribuído, o Git se destaca por ser muito performático, oferecer suporte a desenvolvimento em grandes projetos, com múltiplas ramificações. E é totalmente compatível com a maior rede de compartilhamento de código da atualidade, o GitHub.

O que é GitHub?

O Github tem sim muita relação com o Git.

GitHub é uma plataforma para gerenciar seu código e criar um ambiente de colaboração entre devs, utilizando o Git como sistema de controle.

Ela vai facilitar o uso do Git, escondendo alguns detalhes mais complicados de setup. É lá que você provavelmente vai ter seu repositório e usar no dia a dia.

O sistema web que o GitHub possui permite que você altere arquivos lá mesmo, apesar de não ser muito aconselhado, pois você não terá um editor, um ambiente de desenvolvimento e de testes.

Para se comunicar com o GitHub e mexer nos arquivos do seu repositório, você pode usar o **comando** do git e suas diretivas de commit, pull e push.

Parece assustador? Há uma alternativa: usar um aplicativo desktop mais intuitivo, o GitHub Desktop, conforme vamos acompanhar a seguir.

Como criar uma conta?

Vimos que o GitHub é uma plataforma amplamente utilizada para hospedar repositórios de código e colaborar em projetos de software. E para começar a explorar seus recursos, é necessário criar uma conta que pode ser da seguinte forma:

Passo 1: Acesse o Site

Abra o seu navegador da web e acesse o site do GitHub em "<https://github.com>".

Passo 2: Iniciar a Criação da Conta

Na página inicial do GitHub, você encontrará no canto superior direito um botão "Sign up" (Inscrever-se). Clique nele para iniciar o processo de criação da conta.

Passo 3: Preencha suas Informações

Você será direcionado para uma página em que deve preencher suas informações pessoais, incluindo seu nome de usuário desejado, endereço de email e senha.

Passo 4: Verificação de Captcha

Para garantir que você não é um robô, o GitHub pode solicitar que você complete uma verificação de Captcha. Siga as instruções para provar que você é um usuário legítimo.

Passo 5: Escolha um Plano (Opcional)

O GitHub oferece planos gratuitos e pagos. Selecione o plano que melhor atende às suas necessidades. Você pode começar com o plano gratuito e, se necessário, fazer upgrade posteriormente.

Passo 6: Confirme a Conta

Após preencher todas as informações e escolher um plano, clique no botão "Create account" (Criar conta) para confirmar o processo.

Passo 7: Verificação de Email (Opcional)

O GitHub pode enviar um email de verificação para o endereço fornecido. Verifique sua caixa de entrada e siga as instruções para confirmar seu email.

Pronto! Se você concluiu esses passos, já criou uma conta na plataforma com sucesso.

Agora, você pode explorar os recursos como criar repositórios, colaborar em projetos e compartilhar seu trabalho com outros desenvolvedores.

É importante destacar que as etapas para criar uma conta podem variar de acordo com as atualizações da plataforma, mas o processo geral de criação de uma conta online é semelhante.

Como criar um repositório no GitHub?

Criar um repositório no GitHub é um processo essencial para compartilhar seu código com outras pessoas desenvolvedoras. Aqui estão os passos básicos para criar um repositório:

- 1. Acesse sua Conta:** Certifique-se de estar logado na sua conta do GitHub. Se você não tiver uma conta, siga as etapas para criar uma, conforme explicado anteriormente.
- 2. Página Inicial:** Na página inicial do GitHub, clique no botão "New" (Novo) localizado no canto superior direito.
- 3. Nome e Descrição:** Preencha o nome do seu repositório e uma breve descrição. Escolha se deseja que o repositório seja público (visível para todos) ou privado (acessível apenas por convite).
- 4. Opções de Inicialização:** Você pode optar por inicializar o repositório com um arquivo README, que é uma boa prática para fornecer informações sobre o projeto. Além disso, você pode escolher uma licença para o seu código, se desejar.
- 5. .gitignore:** Você pode especificar tipos de arquivos que o Git deve ignorar ao rastrear alterações. Por exemplo, você pode selecionar uma linguagem de programação específica para gerar um arquivo .gitignore correspondente.
- 6. Escolha um Template (Opcional):** Se o seu projeto se encaixa em um dos modelos de projeto disponíveis, você pode escolher um para iniciar com estrutura pré-definida.
- 7. Create Repository:** Após preencher todas as informações necessárias, clique no botão "Create repository" (Criar repositório) para criar o seu repositório.

Seu repositório estará pronto e você poderá começar a adicionar arquivos, fazer commits e colaborar com outras pessoas.

Repositórios remotos e locais

No contexto do controle de versão com o Git, é importante entender a diferença entre repositórios locais e repositórios remotos.

- **Repositórios Locais:** Um repositório local é a cópia do seu projeto que reside no seu computador. É onde você faz as alterações, cria commits e mantém o histórico do projeto. Você pode trabalhar offline em um repositório local sem necessidade de conexão com a internet.
- **Repositórios Remotos:** Um repositório remoto é uma versão do seu projeto hospedada em um servidor na web, como o GitHub. Eles são usados para compartilhar seu código com outros desenvolvedores e colaborar em projetos. Você pode enviar (push) as alterações do seu repositório local para o repositório remoto e também obter (pull) as alterações feitas por outras pessoas colaboradoras.

Ao criar um repositório no GitHub, você está criando um repositório remoto onde seu código será hospedado e compartilhado com outras pessoas.

Lembre-se de sincronizar regularmente seu repositório local com o repositório remoto para manter todas pessoas colaboradoras atualizadas.

Como linkar os repositórios remoto e local?

Depois de criar um repositório no GitHub, é essencial conectar seu repositório local a ele para que você possa enviar suas alterações para o repositório remoto.

Aqui está um passo a passo, inspirado no tutorial fornecido pelo GitHub, assim que você clica em “Create Repository”:

- **Abra o Terminal:** Se estiver usando um sistema Unix (Linux ou macOS), abra o terminal de comandos, já caso estiver utilizando Windows, abra o Git Bash no Windows.
- **Navegue até o Diretório do Projeto:** Use o comando `cd <caminho/do/seu/repositorio>` para navegar até o diretório do seu projeto local.
- **Inicie um Re却itório Git Local:** Se o seu projeto ainda não é um repositório Git, use o comando `git init` para iniciá-lo.
- **Adicione o Remote:** Use o comando `git remote add origin <URL-do-Repositorio>` para adicionar o repositório remoto como um "remote" chamado "origin".

Exemplo:

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

Como enviar os commits para o repositório remoto?

Depois que seus repositórios, local e remoto, estão vinculados, você pode enviar seus commits para o repositório remoto:

- **Crie um Arquivo README (Opcional):** Se ainda não tiver um arquivo para capa do seu repositório, que explique o que é o projeto, suas funcionalidades, pré-requisitos etc., você pode criar um nesta etapa.

Para criar o arquivo você pode iniciar com o seguinte comando:

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

E para saber mais informações sobre como escrevê-lo, confira nosso artigo [Como escrever um README incrível no seu Github](#).

- **Adicione e Faça o Commit:** No terminal, use os comandos `git add` e `git commit` para adicionar e confirmar as alterações.

```
git add README.md  
git commit -m "Adicionando arquivo README"
```

- **Defina o Nome da Branch Principal:** Se você está usando a versão mais recente do Git, a branch principal é chamada "main". Use o comando `git branch -M main` para definir isso.

```
git branch -M main
```

- **Envie para o Repositório Remoto:** Use o comando `git push -u origin main` para enviar os commits para o repositório remoto.

```
git push -u origin main
```

Dica: Todos esses comandos com os campos preenchidos certinhos na tela do Github:

Quick setup — if you've done this kind of thing before

or [HTTPS](https://github.com/camilafernanda/test.git) [SSH](#) <https://github.com/camilafernanda/test.git>

Get started by creating a new file or uploading an existing file. We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# test" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/camilafernanda/test.git
git push -u origin main
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/camilafernanda/test.git
git branch -M main
git push -u origin main
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

Agora, seus commits locais estão refletidos no repositório remoto no GitHub. Este processo é crucial para a colaboração em equipe e para manter um histórico centralizado do seu projeto. Lembre-se de adaptar as URLs e nomes do repositório conforme necessário.

Como baixar um repositório do GitHub?

Além de conseguir subir um projeto no Github, também podemos fazer o download de repositórios.

Baixar um repositório do GitHub permite que você tenha uma cópia local do código em seu próprio computador. Este processo é conhecido como "clonar" um repositório.

Confira um guia passo a passo para baixar um repositório do GitHub para o seu ambiente local aqui no nosso artigo [Clonando um repositório com Git e GitHub](#).

Lembrando que esse processo é fundamental para contribuir com projetos de código aberto, colaborar em equipes e realizar desenvolvimento local.

Como baixar novos commits do repositório remoto?

Trabalhando com o GitHub, em uma equipe, diversas versões podem ter atualizações diferentes entre si a todo momento, nesse caso, manter-se atualizado com as alterações feitas por outras pessoas colaboradores é crucial.

Por isso, aqui está um passo a passo para baixar novos commits do repositório remoto para o seu repositório local, mantendo-o atualizado:

- 1. Abra o Terminal ou Prompt de Comando:** Semelhante aos outros passos, vamos fazer os comandos no terminal (Linux, macOS) ou GitBash (Windows).
- 2. Navegue até o Diretório do Repositório Local:** Use o comando `cd` para navegar até o diretório do seu repositório local.

```
cd <caminho/do/seu/repositorio>
```

- 3. Atualize o Repositório Local com os Novos Commits:** Utilize o comando `git pull` para buscar os novos commits do repositório remoto e atualizar sua branch local.

```
git pull origin nome-da-branch
```

Se você estiver na branch principal (por exemplo, "main"), pode simplesmente usar:

```
git pull origin main
```

Isso trará as últimas alterações feitas por outros colaboradores para o seu repositório local.

- 4. Resolva Conflitos (Se Aplicável):** Se ocorrerem conflitos durante o processo de atualização, o Git notificará você. Nesse caso, será necessário resolver os conflitos manualmente antes de continuar. As interfaces de [IDEs \(Ambientes de desenvolvimento integrado\)](#) ou o próprio GitHub, oferecem uma visualização otimizada de onde estão os conflitos para que você possa resolver.
- 5. Verifique as Alterações Locais:** Após o `git pull`, você pode verificar as alterações locais usando `git log` para visualizar os novos commits no histórico do seu

repositório.

```
git log
```

Assim, você terá seu repositório local atualizado com os últimos commits do repositório remoto.

Esse processo é fundamental para manter a sincronização entre o seu ambiente de desenvolvimento e as contribuições feitas por outras pessoas da equipe.

Como separar o desenvolvimento de diferentes funcionalidades?

Em uma equipe de desenvolvimento, você pode estar trabalhando na página Home, enquanto outra pessoa do trabalho estiver desenvolvendo o rodapé, o que é um modelo de trabalho bastante usual.

Então, ao trabalhar em projetos de software, é comum ter várias funcionalidades em desenvolvimento simultâneo.

Para isso, o Git oferece estratégias eficientes para isolar e gerenciar o desenvolvimento de diferentes funcionalidades, garantindo que as alterações em uma parte do código não interfiram nas outras.

Aqui estão algumas práticas e conceitos para alcançar isso:

1. Branches (Ramificações):

As branches podem ser utilizadas para isolar o desenvolvimento de diferentes funcionalidades. Cada branch representa uma linha independente de desenvolvimento. Por exemplo, você pode ter uma branch para desenvolver uma nova funcionalidade, outra para corrigir um bug e assim por diante.

- Comando para criar uma nova branch para uma nova funcionalidade:

```
git branch nova-funcionalidade
```

- Mudar para a nova branch:

```
git checkout nova-funcionalidade
```

2. Git Flow:

Considere adotar o modelo Git Flow, uma abordagem popular para organizar branches em um projeto. Ele define branches específicas para desenvolvimento, releases e features, o que facilita o gerenciamento de diferentes aspectos do ciclo de vida do software.

3. Feature Branches (Branches de Funcionalidades):

É comum a utilização de branches específicas para cada funcionalidade que está sendo desenvolvida. Isso mantém as alterações relacionadas a uma funcionalidade isoladas de outras partes do código. Exemplo:

```
git checkout -b feature/nova-funcionalidade
```

4. Git Merge:

Após completar o desenvolvimento em uma branch de funcionalidade, você pode mesclar as alterações de volta para a branch principal (por exemplo, "main" ou "master"). Isso integra a nova funcionalidade ao código principal.

- Mudar para a branch principal

```
git checkout main
```

- Em seguida, mescle as alterações da feature de volta para a branch principal

```
git merge feature/nova-funcionalidade
```

5. Pull Requests (Solicitações de Pull):

Em ambientes colaborativos, é utilizado pull requests para revisar e discutir as alterações antes de mesclá-las de volta à branch principal. Isso adiciona uma camada extra de controle de qualidade e colaboração ao processo.

Essas práticas ajudam a organizar e simplificar o desenvolvimento de várias funcionalidades, tornando o processo mais gerenciável e menos propenso a conflitos.

Além disso, permite uma integração contínua e um histórico de versão mais claro.

Branches e merges

As branches (ramificações) no Git são uma ferramenta poderosa para organizar o desenvolvimento, permitindo que diferentes linhas de código evoluem independentemente.

Mas quando terminamos de desenvolver as funcionalidades para uma branch?

Como unimos as alterações com a branch principal? Para isso, existe o **merge**, com ele, quando a funcionalidade estiver pronta, ela será incorporada à branch principal, exemplo:

- Primeiro é preciso, ir para a branch principal

```
git checkout main
```

- Mesclar as alterações da funcionalidade de volta para a branch principal, com:

```
git merge feature/nova-funcionalidade
```

Nessa etapa, podem ocorrer conflitos no código, mas não se preocupe pois o Git notificará, e você precisará resolver manualmente antes de continuar.

Branches e merges são essenciais para gerenciar o desenvolvimento paralelo de funcionalidades sem comprometer a estabilidade do código principal.

Fork/Pull Request

O processo de fork e pull request é fundamental para contribuições em projetos de código aberto, permitindo que outras pessoas sugiram alterações sem afetar diretamente o repositório original.

- **Fork:** Em um projeto de código aberto, clique no botão "Fork" no canto superior direito da página. Isso cria uma cópia do repositório original na sua conta.

- **Clonando o Fork para o Repositório Local:**

Feito isso, para fazer alterações no código, é preciso ter esse código em sua máquina de forma local, que é possível da seguinte forma:

```
git clone https://github.com/seu-usuario/nome-do-fork.git  
cd <caminho/até/pasta-com-o-nome-do-fork>
```

Em seguida, é uma boa prática você criar uma ramificação para armazenar suas alterações, sem prejudicar o andamento do projeto principal, dessa forma:

```
git checkout -b feature/minha-contribuicao
```

Realize as alterações desejadas, faça commits e, se necessário, crie novas branches para diferentes funcionalidades ou correções.

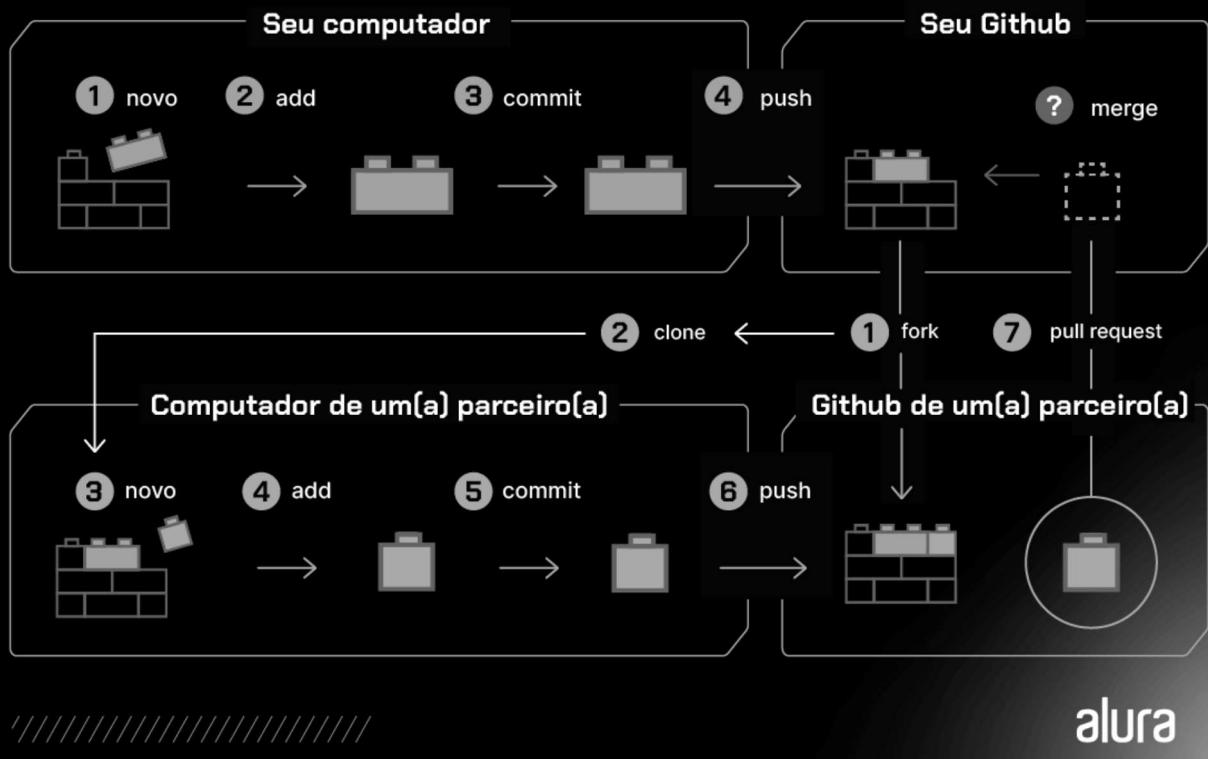
Feito isso, para enviar as alterações para seu repositório com a versão do fork, faça:

```
git push origin feature/minha-contribuicao
```

Por fim, para concretizar suas alterações no projeto, no GitHub, vá até o seu fork e clique em "New Pull Request". Escolha a branch com suas alterações e sugira a incorporação ao repositório original.

Os colaboradores do projeto original vão revisar suas alterações por meio do pull request, onde será possível discutir elas por ali também. Se tudo estiver em ordem, eles poderão mesclar suas alterações no projeto principal.

Git / GitHub Como fazer um Pull Request



alura

Utilizar forks e pull requests promove uma colaboração estruturada, permitindo contribuições externas enquanto mantém o controle sobre o repositório original. Esse fluxo é fundamental em projetos de código aberto e ambientes colaborativos.

Integração com IDEs

Como já escrevemos código em Ambientes de Desenvolvimento Integrado (IDEs), integrar todos esses passos com Git e GitHub, simplifica significativamente o ciclo de vida do desenvolvimento de software.

Vamos explorar como essa integração pode ser alcançada em IDEs populares.

Visual Studio Code (VSCode):

1. Instalação do Git:

Certifique-se de ter o Git instalado na sua máquina. O VSCode geralmente detecta automaticamente a instalação do Git. Para ver mais detalhes de como iniciar com o Git no VSCode, confira a [documentação](#).

2. Configuração do GitHub:

Se ainda não o fez, é preciso configurar as credenciais do GitHub no Git usando os seguintes comandos no terminal (Linux e macOS) ou no GitBash (Windows):

```
git config --global user.name "Seu Nome"  
git config --global user.email "seu-email@example.com"
```

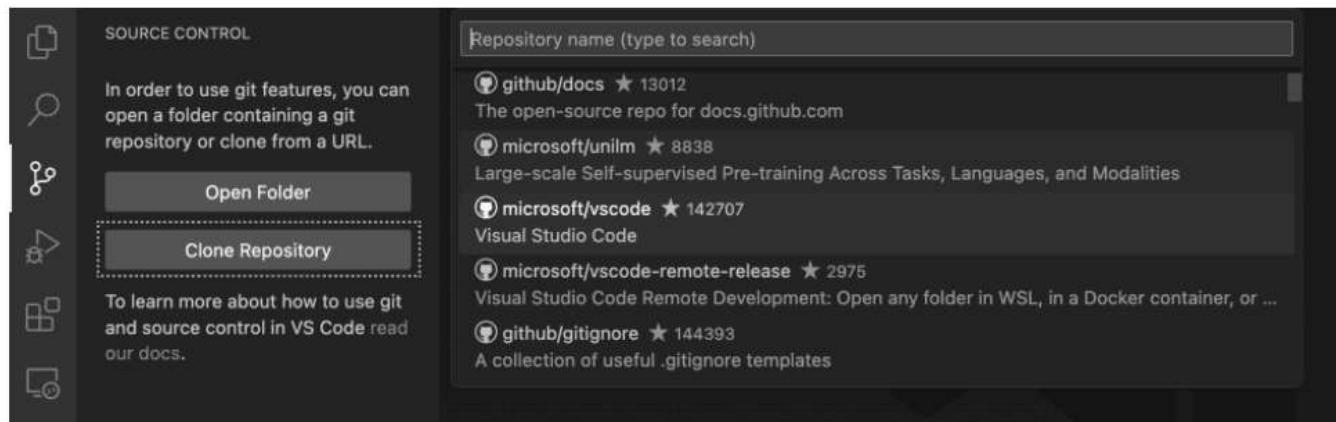
3. Autenticação no GitHub:

Configure a autenticação com o GitHub, usando SSH ou token para segurança aprimorada. E para conferir o passo a passo para cada sistema operacional, confira no nosso artigo

[Nova exigência do Git de autenticação por token, o que é e o que devo fazer?](#)

4. Clonando um Repositório:

Feito essas configurações, use a opção de clonagem no VSCode para copiar um repositório do GitHub para o seu ambiente de desenvolvimento.



5. Controle de Versão Direto no Editor:

Tendo o repositório Git no VSCode, você terá uma interface gráfica para operações do Git, como commit, push, pull e merge, facilitando o controle de versão diretamente no editor.

Outras IDEs:

1. Eclipse:

- Para o Eclipse, você pode utilizar o [plugin EGit](#) para integrar o Git. Configure suas credenciais do GitHub e clone repositórios diretamente do ambiente de desenvolvimento.

2. IntelliJ IDEA:

Assim como o VSCode, o IntelliJ IDEA possui suporte nativo ao Git. Configure o GitHub nas configurações do Git e clone repositórios usando a interface gráfica.

3. Visual Studio (VS):

O Visual Studio também oferece integração direta com o Git. Portanto, da mesma forma como citado para o VSCode, configure suas credenciais e clone repositórios do GitHub sem sair do ambiente de desenvolvimento.

Além dessas interfaces gráficas, essas IDEs também tem um terminal de comandos incluso, o que facilita bastante também. Essa integração do Git e GitHub com IDEs simplifica os fluxos de trabalho, tornando o controle de versão e a colaboração em equipe mais acessíveis diretamente no ambiente de codificação.

Github Desktop: usar Git sem precisar configurar e manter um servidor

Outra opção de utilização do Git e Github, sem utilizar o terminal de comandos ou extensões nas IDEs, é o Github Desktop, que possui uma interface gráfica específica como um "sincronizador de código".

Assim, é possível facilitar as visualizações, o envio e o recebimento das modificações, além dos famosos conflitos de merge, que você não precisa se preocupar nesse primeiro instante.

Nesse vídeo, o Felipe, da Alura, te explica como dar os passos no Github Desktop: [GitHub sem linhas de comando | #AluraMais](#)

Além do Controle de Versão: Serviços e Recursos Adicionais do GitHub

O Github é bastante popular, tanto que é apelidado de rede social da pessoa desenvolvedora, pois além de oferecer os recursos de controle de versão, também é comumente utilizado para compartilhar código e, até mesmo, ser utilizado como um portfólio.

Além disso, na plataforma você consegue baixar projetos abertos pela comunidade por meio do git clone, em que você pode ver passo a passo no artigo [Clonando um repositório com Git e GitHub](#).

[Projetos de código aberto, ou open source](#), frequentemente envolvem colaboração de desenvolvedores de todo o mundo, para isso, existem eventos como o [Hacktoberfest](#) para incentivar a contribuição da comunidade.

Para garantir a contribuição harmoniosa de diversos colaboradores, geralmente é utilizado o Github com práticas de ramificação e colaboração, como diretrizes para criar forks (bifurcações) que é uma cópia independente do repositório de código-fonte.

Além da convenção de abrir issues (problemas) para discussão e fazer pull requests (solicitações de subir alterações) para propor alterações e revisões de código rigorosas para garantir a qualidade do código.

Mas o Github vai muito além disso: essa tecnologia possui diversas outras ferramentas que otimizam nosso trabalho. Confira na sequência:

GitHub Actions: Automatização de Fluxos de Trabalho

Com o Github Actions, é possível automatizar, personalizar e executar fluxos de trabalho no nosso repositório no próprio Github.

Então, por exemplo, qualquer teste que você precisa executar a cada alteração no projeto, você consegue fazer de forma automatizada com o Github Actions.

GitHub Pages: Hospedagem de Sites Estáticos

É possível compartilhar um site estático feito com HTML, CSS e Javascript por meio do Github Pages, que usa os arquivos diretamente do seu repositório, executa os arquivos e publica um site, e fornece um link para você publicar. Para saber mais como é feito esse processo, confira o artigo [Como colocar seu projeto no ar com o Github Pages?](#).

GitHub Codespaces: Ambiente de Desenvolvimento Online

O GitHub Codespaces oferece um ambiente de desenvolvimento online e hospedado na nuvem.

Com essa ferramenta, pessoas desenvolvedoras podem acessar um ambiente de desenvolvimento funcional diretamente do navegador, eliminando a necessidade de configurações locais complexas.

Além disso, oferece integração perfeita com o GitHub, permitindo que os desenvolvedores acessem seus repositórios, issues e pull requests diretamente do ambiente de desenvolvimento.

E para acessar esse ambiente, basta você acessar um repositório no Github e apertar a tecla  (ponto final), que irá abrir automaticamente o ambiente do Codespaces.

GitHub Discussions: Comunicação da Comunidade

O GitHub Discussions é uma plataforma de comunicação colaborativa destinada à comunidade que se reúne em torno de um projeto, seja ele de código aberto ou interno.

Dentro desse espaço, as pessoas da comunidade têm a oportunidade de fazer perguntas, fornecer respostas, compartilhar atualizações e realizar discussões abertas, permitindo o acompanhamento de decisões que impactam o funcionamento coletivo da comunidade.

GitHub Security: Ferramentas de Segurança Integradas

O GitHub Security engloba um conjunto de ferramentas integradas voltadas para a segurança dos projetos.

Essas ferramentas abrangem análise de código, verificações de segurança e testes para verificar se as dependências do código estão seguras, com o Dependabot.

Elas trabalham juntas para manter o código seguro e atualizado, proporcionando uma camada de proteção e detecção de vulnerabilidades diretamente na plataforma do GitHub.

GitHub Insights: Análise e Estatísticas de Repositórios

O GitHub Insights é uma funcionalidade que oferece uma variedade de análises e estatísticas relacionadas aos repositórios hospedados no GitHub.

Com ela, você pode obter informações valiosas sobre o desempenho do seu projeto, incluindo métricas de contribuição, atividade da comunidade e tráfego do repositório.

Essas estatísticas proporcionam uma visão detalhada sobre como o seu projeto está sendo utilizado e como a comunidade está interagindo com o seu código, auxiliando na avaliação e no aprimoramento do desenvolvimento e da colaboração.

Git e Github para sobrevivência

Nessa nova **websérie** da Alura, o Mario Souto, conhecido como DevSoutinho, traz importantes pontos do **uso dessa plataforma**:

Git e Github para Sobrevivência #01: Como o Git funciona? <https://www.youtube.com/watch?v=BAmvmaKQklQ>

O segundo episódio também está no ar: **Como funciona o merge?** Aqui você vai entender os **branches**, a **master** e como juntamos o trabalho de diversas pessoas e equipes:

Git e Github para Sobrevivência #02: Como o merge funciona? https://www.youtube.com/watch?v=t_UND1if4eI

Mais referências e conteúdo

O Fabio Akita tem um canal que admiro muito e explica o Git e sua importância sem ser um tutorial:

Entendendo GIT | (não é um tutorial!) <https://www.youtube.com/watch?v=6Czd1Yetaac>

Se você quer ouvir um guia para iniciantes em Git e Github, eu gravei esse episódio justo para facilitar esses passos:

Guia do Iniciante em Github | Hipsters #184

Temos um artigo **tutorial** sobre começar com Git, aprenda a versionar para você fazer seus primeiros commits.

E nosso principal curso de Git e Github é extremamente elogiado.

E a partir daí, você pode entrar em merges e branches. Enquanto isso, experimente fazer os primeiros pushes e pulls, sincronizando com o Github Desktop. é um excelente caminho.

Perguntas Frequentes:

Como usar o Git?

Você vai usar o Git para guardar o versionamento de todo seu sistema de maneira segura e distribuída. Deve sempre fazer pequenos commits e 'pushar' suas modificações em grupos que fazem sentido.

Do outro lado, deve fazer os 'pulls' para sincronizar com as modificações de todo time. O Git é uma ferramenta, você vai precisar de um servidor principal (mesmo que seja distribuído) para facilitar seu trabalho. É aí que entra o Github, como sendo uma das opções já prontas.

O que é commit git?

Cada commit registra um momento de vida do seu projeto. É uma foto (snapshot) daquele instante dos arquivos. Com isso, você consegue se referenciar àquele momento do projeto, podendo voltar a cada “fase” dele, analisá-lo, ver as diferenças em um outro momento.

Não se esqueça que os commits são feitos no seu repositório local e você precisa ‘pushar’ (empurrá-los) de volta para a origem, para que outras pessoas tenham acesso a essas fotos e atualizem-se no tempo de vida.

O que é um repositório em git?

Um repositório git nada mais é um diretório, ou pasta, que contém os arquivos de configuração da ferramenta contidos na pasta `.git`, gerada a partir do comando `git init` na pasta do projeto.

Com isso, o diretório passa a ser monitorado pelo Git e guardar todo o histórico de alterações do projeto, o que dá toda a mágica do controle de versões pois possibilita voltar no tempo no projeto, além de criar ramificações para facilitar a vida de devs que trabalham nele.

Qual a diferença de Git e GitHub?

Git é a ferramenta que Linus criou. Se você quer utilizá-la, pode fazer tudo sozinho e hospedar um repositório principal para ter seu software versionado. Contudo, isso pode ser uma tarefa complicada.

Algumas empresas, como Github, Bitbucket e Gitlab, já oferecem esse serviço, controlam o acesso dos usuários e oferecem ferramentas extras de gerenciamento de projetos.

O Github é uma opção para que você use o Git de maneira mais simples, mas não é obrigatório, apesar de ser extremamente utilizado.

Como criar um arquivo no git?

Quando utilizamos o comando `git init`, que inicializa um repositório dentro de uma pasta, é criada uma subpasta `.git`, que armazena todas as informações sobre o histórico do projeto.

A partir desse momento, o git monitora todas as alterações feitas nos arquivos do projeto, bem como a criação de novos arquivos e pastas.

Nesse caso, o git não é responsável pela criação dos arquivos, mas por monitorar suas alterações e gerenciá-las.

Quais são as alternativas ao Github?

O Bitbucket e o Gitlab são as duas principais alternativas de serviço Git, além do Github.

Conclusão

O Git e GitHub se solidificaram em uma posição fundamental no universo do desenvolvimento de software e na colaboração entre equipes de pessoas programadoras. Hoje, essas tecnologias estão presentes em inúmeros projetos.

A flexibilidade e facilidade de uso têm sido essenciais para dar maior agilidade ao desenvolvimento de software. Afinal de contas, melhoram a colaboração entre pessoas desenvolvedoras e garantem a organização e a segurança de versões de código.