BRUNNA CROCHES

PARTE 1

GIT

Artist Image :Descourtilz, Jean-Théodore | Dates:179?-1855

Guia iniciante: GIT

ABOUT ME

Brunna Croches

Developer Full Stack

Brunna Croches é Dev FullStack, advogada e empreendedora.

Apaixonada por tech, vem adquirido vasto conhecimento na área.

Desenvolveu projetos ricos em diversidade, buscando captar as próximas tendências e necessidades do mercado.

Neste e-book você aprenderá ou recapitulará de forma simplificada e otimizados conceitos de programação feito por ela.

let's share

SUMMARY
GIT

1 Introdução

Conteúdo de apoio

O Git é uma ferramenta criada com o intuito de facilitar o controle de versão dos arquivos de projetos de software, sejam eles da linguagem de programação que você utiliza, XML, JSON, HTML, CSS, enfim, todos eles. Com o Git é abstraído o esforço e a complexidade da execução de tarefas que podem prejudicar a produtividade de qualquer programador. Para isso, ele fornece comandos simples para tudo o que precisamos no que diz respeito ao controle de versão: da criação do repositório à criação de uma nova versão do código.

Para compreender a importância dessa ferramenta, imagine que você está trabalhando em um projeto e decide experimentar uma biblioteca indicada por um amigo. Muitas vezes, por descuido, ou por não utilizar uma ferramenta de controle de versão, fazemos o teste da biblioteca no código do nosso projeto, sem criar uma nova versão para o teste. Quando fazemos isso estamos assumindo um risco muito alto. No teste, podemos realizar várias mudanças no código e depois perder, até mesmo, horas, para conseguir voltar ao estado que gostaríamos, no caso de optarmos por não adotar a biblioteca. A depender do estágio do projeto, pode ser comum recomeçar do zero.

Agora, imagine que você está trabalhando em um projeto com uma equipe de cinco pessoas. Como vocês fariam para manter uma versão do projeto compatível? Como você faria para passar aos seus colegas as mudanças que você fez no código? Quão trabalhoso pode ser gerir o código de todas as pessoas?

O Git facilita tudo isso ao prover uma solução distribuída, leve e robusta. A **Figura 1** apresenta os principais elementos relacionados a essa ferramenta e como ela funciona.

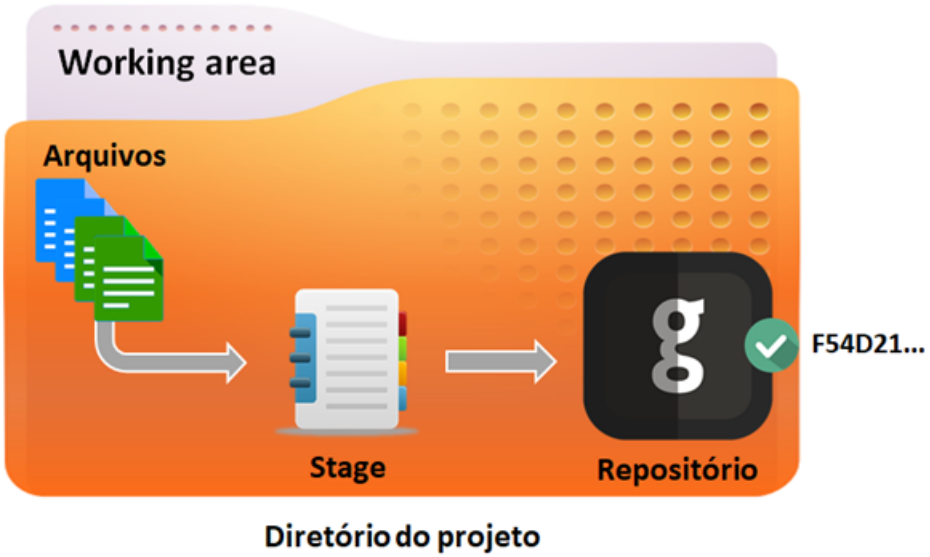


Figura 1. Funcionamento do Git

Note que a imagem principal representa uma pasta, o diretório do projeto. Esse nada mais do que o diretório em que nosso projeto será criado. Ao criarmos um repositório Git, fazemos desse diretório nossa Working Area, ou área de trabalho. Dentro dele, é criado um diretório oculto, de nome .git. Esse é nosso repositório de fato. Ele que guardará as informações, os metadados, referentes ao nosso código.

A partir disso, qualquer arquivo criado/modificado na working area do Git. Para que possamos adicionar esse arquivo criado/modificado ao repositório, precisamos lidar com outro conceito, o conceito de Stage (ou palco). Assim, devemos adicionar os arquivos nessa área e, somente a partir disso, executar o commit. O stage nada mais é que um espaço intermediário que irá conter o que deve ser adicionado ao repositório.

Quando executamos o commit, estamos adicionando os novos arquivos e as mudanças realizadas ao repositório, criando assim uma nova versão do nosso código, o que gera um código hash, representado na **Figura 1** pelo código de seis dígitos à direita. É utilizando esse código que conseguimos navegar entre as versões do projeto.

Sugestão de conteúdo

- Artigo Trabalhando com o Repositório Remoto no Git

Guia

- O que é?
- Git

Links

- Site oficial do Git
- Site para download do Visual Studio Code

1.1 Git na prática

Aprenda a utilizar o Git para controlar as versões do código do seu projeto. De forma simples, vamos analisar na prática os comandos mais básicos do Git. Assim, você saberá como criar um repositório, adicionar os arquivos ao stage, criar versões do código com o comando commit, criar branches para testar bibliotecas ou implementar novas funcionalidades, navegar entre branches, entre outros recursos.
07:05 min

Conteúdo de apoio

O uso do Git pode ser feito de várias formas. As IDEs e editores de código mais modernos, por exemplo, já trazem plugins e soluções nativas para utilização desse sistema de controle de versão. Nesse vídeo, no entanto, para apresentar os principais comandos independentemente do ambiente de desenvolvimento que você faz uso, optamos pelo terminal do sistema operacional.
Assim, após instalar o Git em sua máquina, os dois primeiros comandos a executar no prompt são:

```
git config --global user.name "Nome do usuario"
git config --global user.email e-mail@exemplo.com
```

O Git pede uma identificação, para que possa sinalizar o autor das mudanças realizadas no repositório. Logo após, no diretório em que você pretende criar ou já criou o projeto, execute:

```
git init
```

Isso é o que precisamos para inicializar o repositório e criar a pasta oculta .git. Agora, nosso diretório também é o que chamamos na linguagem Git de working area, o diretório de trabalho.
Qualquer arquivo criado/modificado neste diretório será monitorado pelo Git. Para que possamos adicionar esse arquivo em nosso repositório, primeiro precisamos adicioná-lo ao stage, o que é feito com o comando:

```
git add NomeDoArquivo.extensão
```

Dessa forma o arquivo estará referenciado nesse espaço intermediário. A partir disso, para confirmar as mudanças e adicionar o novo arquivo ao repositório, precisamos executar:

```
git commit -m “Mensagem descritiva das mudanças realizadas.”
```

É válido destacar a importância da mensagem especificado junto com o commit. É recomendado que a partir dela o programador saiba tudo o que foi feito naquela versão.
Outro conceito importante do Git é o conceito de branch. É aconselhado criar um branch sempre que você desejar testar alguma nova API/Biblioteca/Framework ou começar a criar uma nova funcionalidade no projeto. Quando inicializamos o repositório, automaticamente ele cria o branch master. Esse é o branch principal do projeto. Quando estamos trabalhando em equipe, principalmente, o ideal é que esse branch receba apenas versões do código com as funcionalidades já testadas e avaliadas.

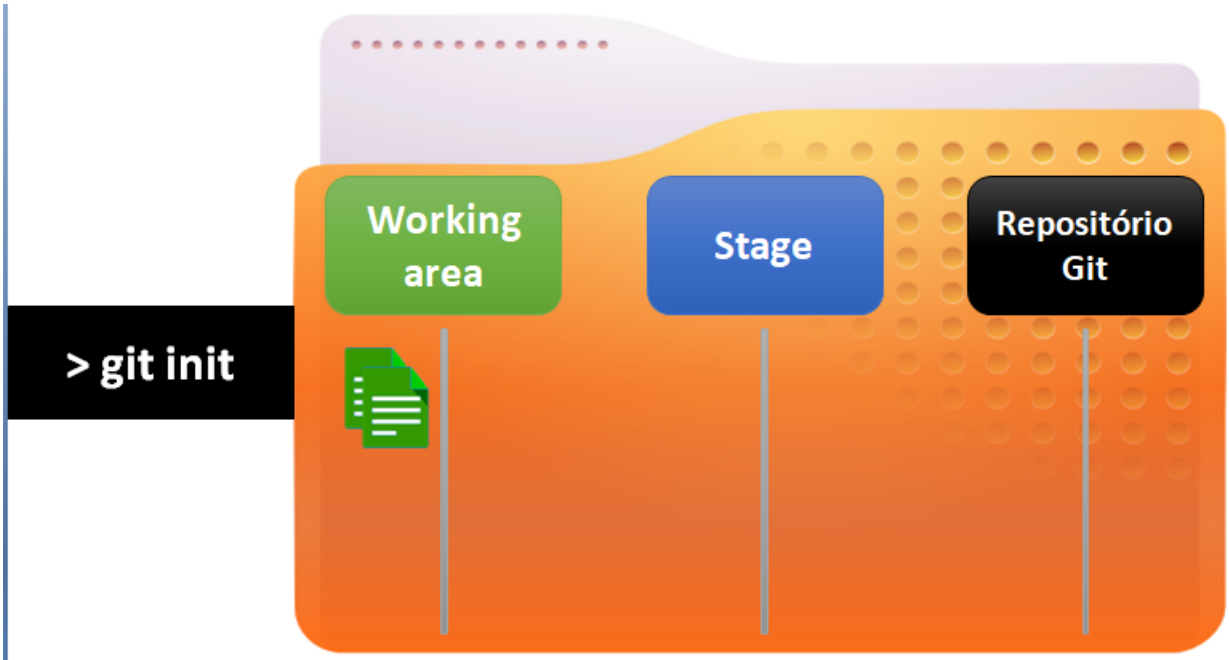
Para começar a criar uma nova funcionalidade, como já mencionado, o ideal é criar um novo branch. Ao fazer isso, lembre-se de dar um nome autoexplicativo. Para criar o branch, execute:

```
git branch nomeDoBranch
```

Ao fazer isso, é como se criássemos uma cópia de todos os arquivos do nosso projeto. A diferença é que essa cópia é gerenciada pelo Git. Portanto, você não verá arquivos duplicados. A partir disso, para acessar o novo branch, o nosso novo ramo, execute no terminal:

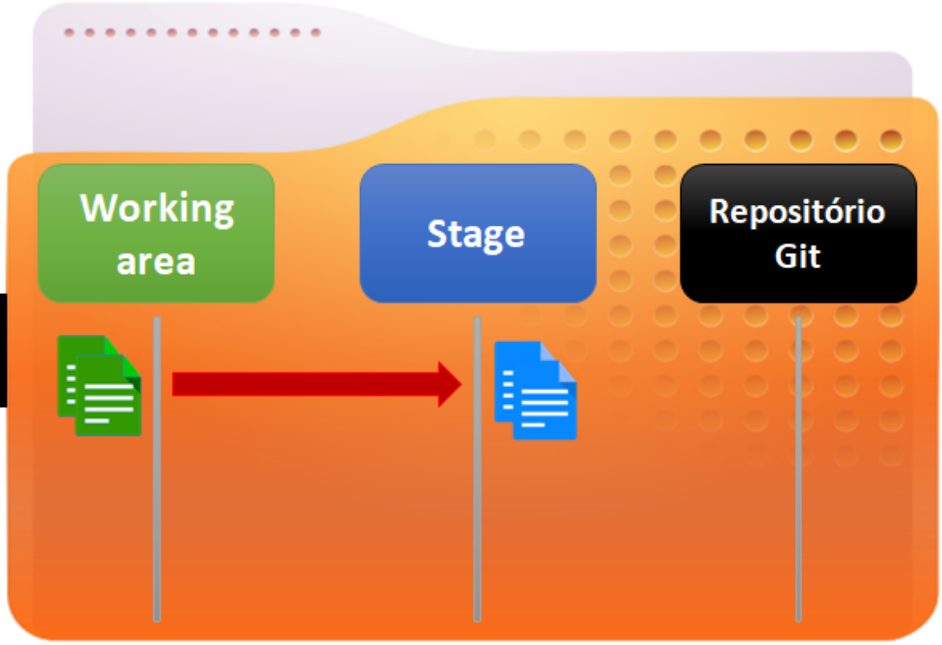
```
git checkout nomeDoBranch
```

Logo após, todas as mudanças realizadas a partir de agora estarão acessíveis apenas nesse branch. Portanto, se você criar um ou mais arquivos, ao voltar para o branch master (com o comando git checkout master) esses arquivos não serão encontrados, pois nesse ramo esses arquivos não existem.
Agora que entendemos como Git funciona, podemos resumir o fluxo básico de trabalho com ele à sequência de imagens abaixo:



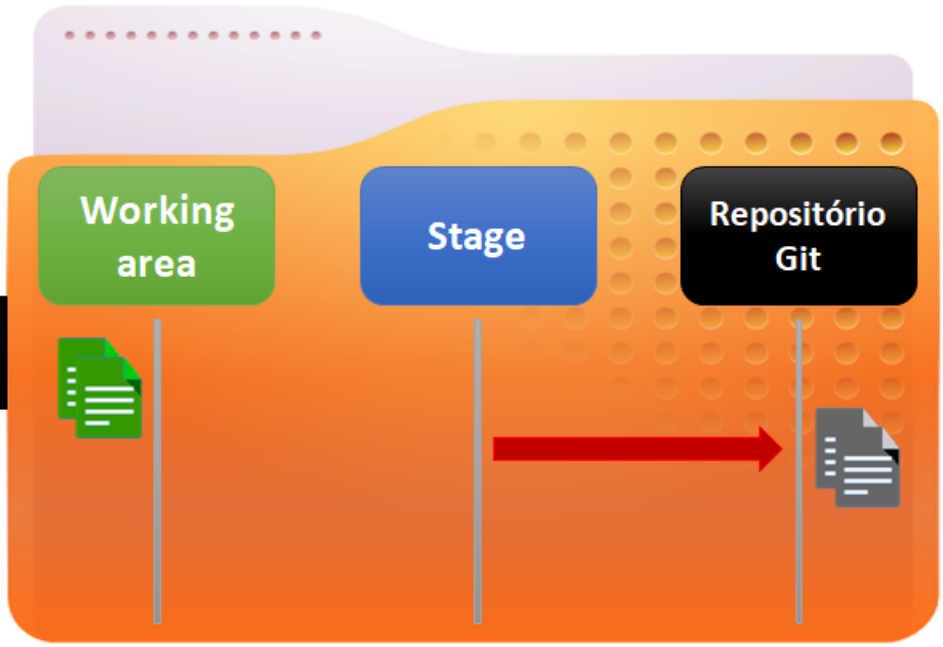
Diretório do projeto

> git add .

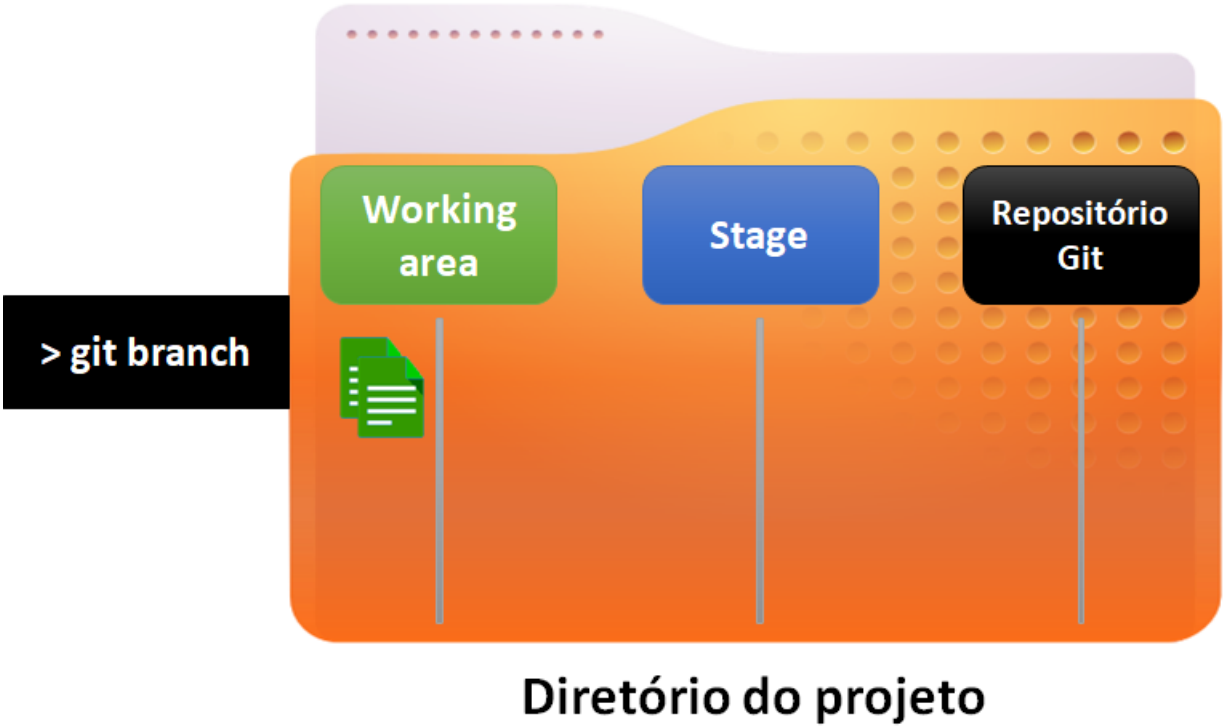


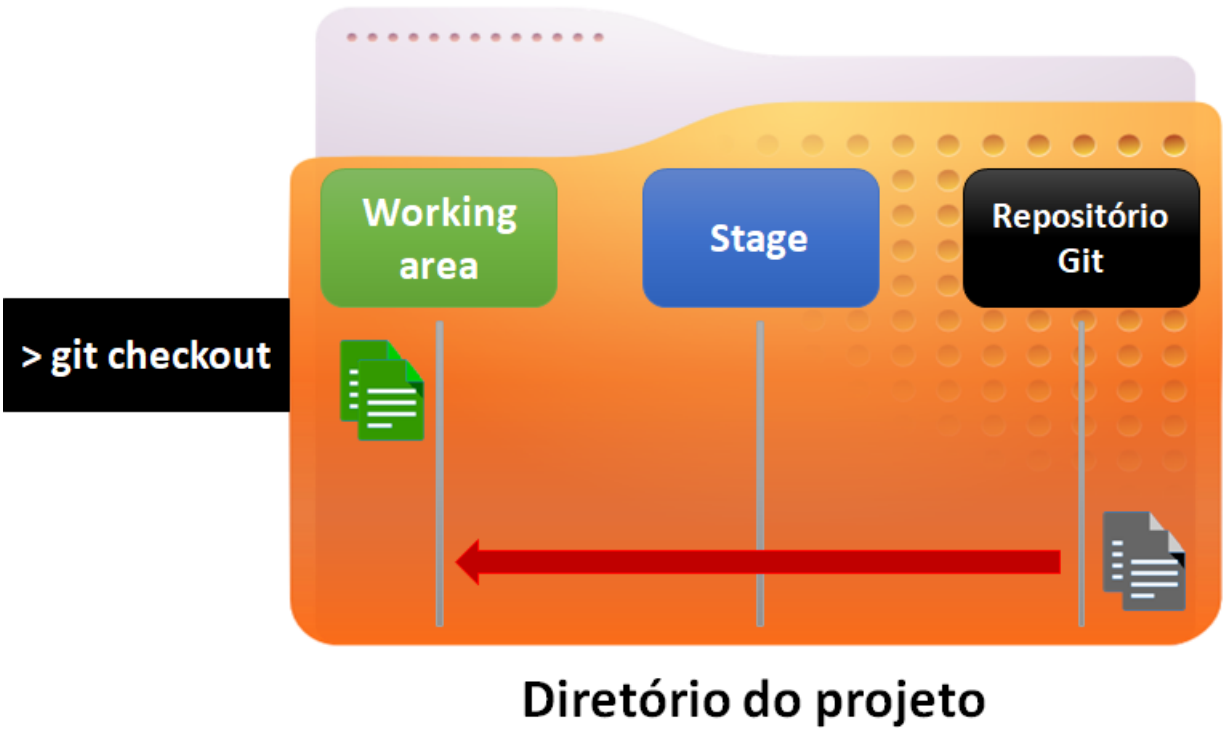
Diretório do projeto

> git commit



Diretório do projeto





#PraCegoVer - Transcrição dos Slides
Tela 1: Inicialização do repositório: git init

Tela 2: Arquivos foram criados/modificados: git add NomeDoArquivo.extensão

Tela 3: Adicionar arquivos ao repositório e criar nova versão: git commit -m “Mensagem descritiva”

Tela 4: Criar/testar algum recurso: git branch nomeDoBranch

Tela 5: Acessar outro branch: git checkout nomeDoBranch

1.2 Principais características do Git

Aprenda neste artigo as principais características do Git e como utilizá-las sem complicações.

Como usar o Git?
Seja no desenvolvimento de novos softwares ou na manutenção de sistemas, é comum ter mais de uma pessoa, às vezes grandes equipes, editando arquivos de código fonte. Muitas dessas edições necessitam ser realizadas paralelamente no mesmo arquivo, inclusive por pessoas em projetos diferentes. Nesses cenários, fica a dúvida: como gerenciar as contribuições de diferentes desenvolvedores ao longo do tempo?

O controle de versão há muito tempo se tornou imprescindível no desenvolvimento de software justamente por fornecer uma solução para esses dilemas. No entanto, a simples adoção de uma ferramenta de controle de versão não é suficiente para o sucesso.

Com o grande número de ferramentas, cada qual tecendo seus próprios conceitos e paradigmas no versionamento de artefatos, **a única maneira de ser bem-sucedido nessa tarefa é conhecendo bem o que cada ferramenta oferece, saber diferenciar os conceitos por trás de cada ferramenta e definir a que melhor se encaixa no contexto em questão**.

Este artigo, além de **explorar o correto uso do Git** – uma das ferramentas mais populares atualmente para controle de versão.
Em muitas equipes de desenvolvimento há dois sentimentos que vem se tornando muito comuns: o receio de fazer alterações no código que possam quebrar outras funcionalidades e o receio de compilar um projeto com código a mais ou código a menos.

Além disso, geralmente esses códigos são repletos de comentários documentando detalhadamente e de forma exaustiva aspectos que não deveriam ser registrados em comentários, tais como relacionar a inclusão de uma variável a um projeto específico.

Uma solução antiga para esses problemas é o **versionamento de controle**, que visa dar segurança aos programadores em relação às **versões corretas a serem compiladas** **e fornecer comentários relacionados as alterações implementadas na própria ferramenta de controle de versão, eliminando assim a complexidade de um código poluído** repleto de explicações e o receio dos desenvolvedores de estarem alterando versões erradas do sistema.
Entretanto, mesmo em projetos que fazem uso de versionamento, esses sentimentos insistem em fazer parte do cotidiano das pessoas envolvidas e não é raro a ocorrência de problemas decorrentes do controle de versão. Mesmo com ferramentas os problemas ainda ocorrem, o que evidenciam o fator humano como causa, em boa parte, devido à falta de entendimento por completo do fluxo de trabalho do controle de versão ou da ferramenta adotada pela equipe.

É comum, por exemplo, vermos pessoas acostumadas a usar o SVN sem entender seu fluxo de trabalho e **migrarem para o Git** pensando ser a mesma coisa e vice-versa. Esses equívocos são os grandes geradores de problemas em relação ao versionamento e alimentam o receio da equipe em mexer no código, especialmente no Git, onde o erro mais comum é fazer o commit pensando que as alterações serão enviadas para o servidor de controle de versão como acontece no SVN.

Tendo como ponto de partida a necessidade do entendimento, tanto conceitual quanto prático das ferramentas de controle de versão para a solução desses cenários, esse artigo abordará os diferentes tipos de controle de versão a fim de fornecer uma base conceitual para se entender o fluxo de trabalho e focará no Git, mostrando como o usá-lo da maneira correta.

Controle de Versão com Git

Boa parte das tarefas profissionais que temos no dia a dia giram em torno de uma série de interações com algum tipo de conteúdo, que varia de acordo com a função de cada indivíduo.

Esse conteúdo pode assumir muitas formas, tais com dados financeiros em uma planilha, textos e relatórios em documentos de texto, apresentações e, no caso dos programadores, código fonte em arquivos de texto. As tarefas diárias dos profissionais, em geral, podem ser resumidas em um ciclo de criar conteúdo, salvar, editar e salvar o conteúdo novamente, como ilustra a **Figura 1**.

Figura 1.Ciclo de vida de arquivos nas tarefas diárias da maioria dos profissionais

Esse ciclo pode se tornar muito complexo em pouco tempo em relação ao número de pessoas que compartilharão informações e necessidades em usar e alterar o mesmo conteúdo.

Imagine um projeto de software com uma equipe pequena de, por exemplo quatro desenvolvedores. Nesse cenário muito provavelmente os desenvolvedores, ao longo do projeto, precisarão alterar um conteúdo (código-fonte) gerado por outro desenvolvedor e, talvez, essa necessidade apareça justamente enquanto houver outro desenvolvedor fazendo alterações no código.

O que fazer nesse caso? O programador A que quer alterar o código só poderá fazê-lo quando o programador B que estiver alterando o código terminar suas tarefas? Mas, e se isso atrasar o programador A? Quanto mais gente no projeto e quanto mais longo for, maior a complexidade em lidar com questões como estas.

É aqui que entra o controle de versão, que nada mais é do que uma série de diretrizes para gerenciamento de conteúdo compartilhado. Por meio do controle de versão é possível saber quem fez as alterações, o motivo das mudanças e o que foi alterado em qualquer tipo de conteúdo.

A **Figura 2** ilustra as alterações em um arquivo de código-fonte feitas por um programador em diferentes momentos. Esse é o histórico do arquivo e fornece informações muito importantes sobre a evolução do mesmo, tais como as versões do arquivo ao longo do tempo e quais as mudanças em cada versão.

Figura 2. Exemplo de histórico de alterações em um arquivo de código fonte ao longo do tempo

Conhecer o histórico de um determinado arquivo é muito importante, pois fornece preciosas informações para o entendimento do arquivo em si. Contudo, o maior benefício do controle de versão é a colaboração.

A **Figura 3** ilustra as alterações em um arquivo de código-fonte feitas por quatro programadores, cada um representado por uma cor diferente.
Com esse histórico é possível saber quem foi o responsável pela criação de cada versão, quais mudanças ocorreram em cada versão, o motivo das alterações e, o recurso mais importante e poderoso no controle de versão, unir todas as contribuições de todos os indivíduos quando trabalharam em versões paralelas do arquivo em um determinado arquivo, contendo todas as alterações por meio do processo chamado merge.

Graças a ele é possível ter diferentes pessoas alterando o mesmo arquivo ao mesmo tempo e, quando essas pessoas terminarem suas tarefas, une suas alterações em um arquivo que contemple as tarefas de ambos.

Dessa forma, o trabalho se desenvolve muito mais rápido, pois não é mais preciso esperar que alguém que esteja trabalhando em um arquivo termine para que outra pessoa possa fazer suas alterações.

Figura 3.Exemplo de histórico de alterações em um arquivo de código-fonte feitas por diferentes programadores ao longo do tempo

Tudo o que foi falado até aqui diz respeito ao controle de versão. Existem muitas ferramentas que o implementam, cada qual a sua maneira, com algumas semelhanças e diferenças.

Atualmente, uma das ferramentas mais conhecidas e que vem tomando espaço a cada dia é o Git, que é usada por cerca de 68% dos desenvolvedores, de acordo com o Java Tools and Technologies Landscape Report 2016 (seção **Links**). Sendo assim, essa é uma ferramenta que vale a pena conhecer e dominar por completo e será o foco do próximo tópico, que apresenta o Git e suas particularidades, bem como do restante do artigo, que mostrará como trabalhar efetivamente com controle de versão utilizando esta ferramenta.

O que é Git?

***O Git implementa de forma automatizada as diretrizes de controle de versão**, fornecendo histórico de mudanças, facilitando a colaboração no manuseio de arquivos por diferentes pessoas ao mesmo tempo.*
Além do Git, existem outras ferramentas de controle de versão, tais como CVS e SVN, os mais utilizados nos anos anteriores ao Git. No entanto, são ferramentas cujo funcionamento e forma de trabalho são muito distintas e, para entender de forma mais fácil como cada uma funciona, o ideal é deixar de lado tudo o que se sabe das outras ferramentas a fim de evitar confusões entre termos e funcionalidades.

O Git é atualmente a ferramenta de controle de versão mais utilizada por desenvolvedores. Grande parte dessa adesão em massa está relacionada ao fato de o Git ser uma das ferramentas mais recente e, portanto, trazer melhorias significativas no versionamento em relação a ferramentas mais antigas.

Convém um breve comparativo entre o Git e ferramentas como o CVS e SVN para entender a evolução dessas ferramentas, suas vantagens e desvantagens e o motivo da popularidade do Git.

Git versus CVS e SVN

Diferentemente do Git, o CVS e o SVN são sistemas de versionamento centralizados. Isso quer dizer que eles possuem um **repositório central** em que os usuários podem enviar (commit) e receber (checkout) artefatos versionados.

Essa abordagem centralizada oferece como principal vantagem o controle sobre os projetos, facilitando a implementação de segurança de acesso e bloqueio de arquivos (lock). Entretanto, parece haver mais desvantagens do que vantagens nesse modelo.

Entre as inconveniências desse paradigma de versionamento podemos citar problemas de escalação, onde muitos usuários e projetos sob o mesmo repositório geralmente tornam essas ferramentas lentas e a impossibilidade de se trabalhar offline, obrigando os usuários a sempre estarem conectados no servidor para executar tarefas como criação de tags, branchs e merge.

Além disso, há diferenças entre o próprio CVS e o SVN, que também são diferenças significativas, sendo a mais relevante o tipo de commit, que no CVS é feito por arquivo enquanto que no SVN é possível fazer o commit agrupado de arquivos, facilitando o rollback e identificação de algum commit que possa quebrar o código.

Por outro lado, o Git implementa a abordagem de controle de versão descentralizada, que possui uma curva de aprendizado um pouco maior do que o paradigma centralizado. Entretanto, uma vez que se apreende os conceitos e passa-se a utilizar a ferramenta, todo o fluxo de trabalho se torna fácil e intuitivo.

A grande diferença entre os sistemas centralizados e os sistemas distribuídos, como o Git, é em relação ao commit. Em sistemas centralizados, o commit simplesmente envia todas as alterações nos artefatos para o repositório central. Nos sistemas descentralizados o commit apenas cria uma imagem das alterações em um repositório local e, só após o comando o push (que será abordado no decorrer do artigo) é que as alterações nos artefatos são enviadas para o repositório remoto.

Essa forma de trabalho descentralizada torna o Git muito mais rápido em relação as ferramentas centralizadas, além de promover o uso de branches locais para experimentos e commits frequentes, práticas comuns e desenvolvidas na disciplina de métodos ágeis.

Atualmente o **SVN** é a ferramenta de controle de versão centralizada mais popular. Ela é especialmente adotada em projetos individuais e de pequenas equipes, especialmente por ser muito simples de usar (devido ao paradigma de controle centralizado adotado).

Mas simplicidade de uso é um atributo relativo, afinal nada impede o uso do Git de forma simples, como o SVN, e ter a disposição muitas características e possibilidades de trabalho fornecidas pelo paradigma de versionamento distribuído.

Além disso, quando se conhece bem os principais conceitos e características de uma ferramenta, seu uso torna-se naturalmente mais fácil. Por isso, os próximos tópicos elucidam os principais conceitos exclusivos do Git e mostram como essa ferramenta trata os dados que devem ser versionados.

Snapshots no Git

A maioria dos outros controladores de versão gerenciam diferenças entre arquivos, associando a cada um deles uma lista de mudanças. Mas no Git as coisas são bem distintas, pois ao invés de uma lista de mudanças associada a cada arquivo, **ele usa snapshots para manter o histórico, ou seja, a cada commit o Git copia todos os arquivos do projeto e muda a referência dos mesmos marcando-os como sendo a versão atual.**
No caso de arquivos que não foram alterados, o Git apenas mantém um link para o arquivo sem copiá-lo novamente. Esse processo de funcionamento é ilustrado na **Figura 4**.

Nesse exemplo é possível observar o estado inicial onde todos os arquivos foram criados, Version 1.
Em seguida, após alterações nos arquivos A e B, uma nova versão é criada, a Version 2, com os arquivos A1 e C1 copiados literalmente para esse snapshot e uma referência ao arquivo B, que não sofreu nenhuma alteração.
Já na Version 3, quando apenas um arquivo é alterado (arquivo C2), apenas ele é copiado para o novo snapshot e os demais são links para os arquivos anteriores não alterados. As demais versões dessa linha do tempo seguem a mesma lógica.

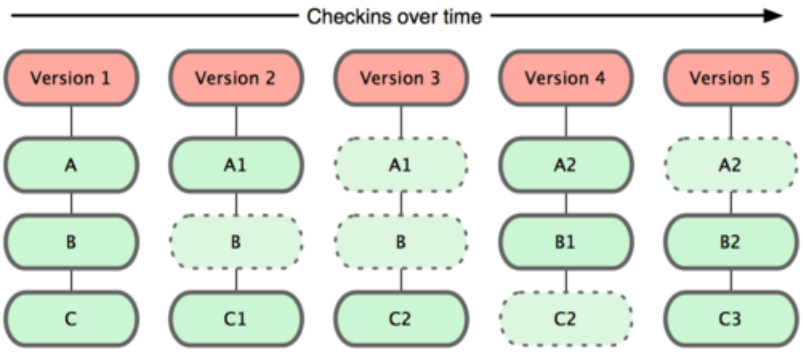


Figura 4. A linha do tempo de histórico de versão do Git baseada em snapshots

Essa forma de lidar com as versões como se fossem snapshots traz muitas vantagens, especialmente em relação a criação de branches. Como no Git um **branch** é um ponteiro para algum commit, poderíamos criar facilmente um branch de qualquer uma das versões da **Figura 4**.

O branch seria criado pelo Git simplesmente adicionando um ponteiro para algumas versões e, a partir de então, alterações e commits nesse novo branch seguiriam o mesmo processo de versões e snapshots a partir do ponto de criação desse novo branch em diante.

Trabalho local com Git

Outra característica única do Git é sua forma de trabalho local. Diferente de outros controladores de versão, a maior parte das operações do Git pode ser executada localmente, já que o Git mantém um histórico do projeto localmente. Isso faz do Git uma ferramenta extremamente rápida, que não sofre com a latência da rede para a execução da grande maioria de suas operações. As principais vantagens dessa característica são a rapidez nas operações (imagine ter que

comparar diferentes versões de um arquivo de um ou dois meses atrás usando recursos locais versus recursos remotos) e a possibilidade de se trabalhar e ter a disposição quase todas as operações mesmo quando não se está conectado na rede.

Estados do Git

Um dos conceitos mais importantes do Git são os estados que podem ser aplicados em cada arquivo. São três estados fundamentais:

- **Commited** – arquivos armazenados na base local.
- **Modified** – arquivos que sofreram mudanças, mas não foram enviados para a base local (commit).
- **Staged** – arquivos modificados marcados para que façam parte do próximo commit.

A **Figura 5** ilustra a transição de estados de arquivos versionados pelo Git em seu fluxo de trabalho. Além dos estados, pode-se observar seções que estão relacionadas ao estado de cada arquivo.
O Git os organiza no diretório de trabalho (working directory), área de preparação (staging area, também chamada de index) e repositório (git directory).

O repositório é o local onde são armazenados os metadados e o banco de objetos do projeto. Essa é a parte copiada quando um projeto é clonado.
O diretório de trabalho é um checkout de alguma versão do projeto. Os arquivos desse diretório são obtidos a partir do banco de dados comprimido do Git no repositório e disponibilizados no disco, possibilitando sua utilização e edição.

A área de preparação é, na verdade, um arquivo chamado de index dentro do repositório que especifica o conteúdo do próximo commit.
Em linhas gerais, o fluxo de trabalho com o Git pode ser descrito da seguinte forma:

1. **Faz-se o checkout de um projeto do repositório, torna-se o diretório de trabalho.**
2. **Todas as alterações (correções, desenvolvimento, etc.) são realizadas no diretório de trabalho. As alterações vão para a área de preparação.**
3. **Quando o commit é feito, todos os arquivos da área de preparação são armazenados no repositório.**

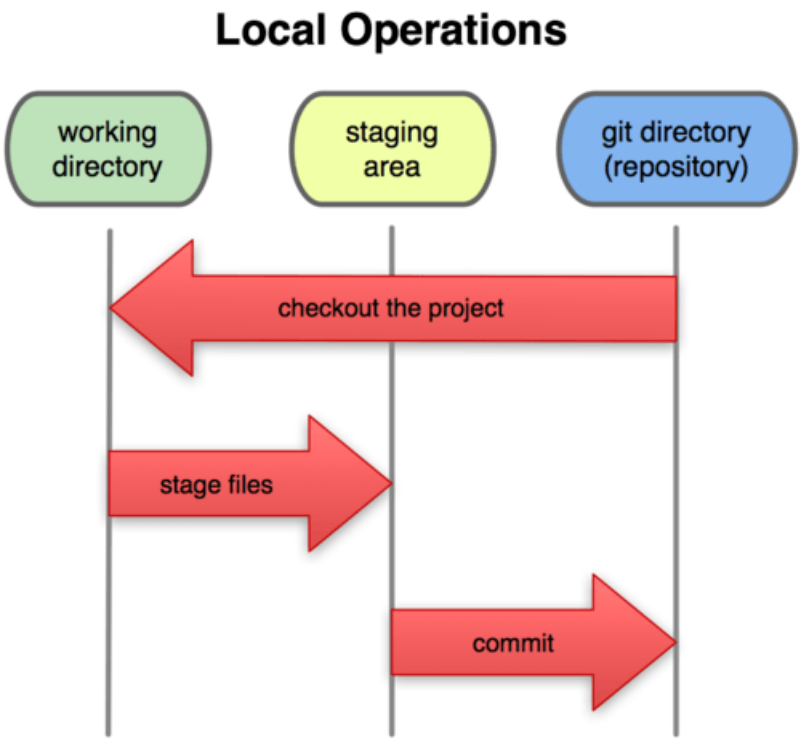


Figura 5. Transição de estados de arquivos versionados pelo Git
O entendimento dos estados, da forma de trabalho local e dos snapshots são fundamentais para se utilizar o Git de forma fácil e correta, pois essas três características são a base que define o versionamento de controle distribuído e o próprio Git.

Os próximos tópicos se concentram no uso prático do Git conforme o fluxo de trabalho apresentado na **Figura 5**, demonstrando os principais comandos para lidar com versionamento. Veja na seção **Links** o link para fazer o download, instalar e configurar o Git. Partiremos da ideia que o mesmo está pronto em sua máquina.

Comandos do Git

Depois de conhecer tantos detalhes a respeito do Git, seu fluxo de trabalho e termos comumente utilizados ao se trabalhar com controle de versão, convém agora dominar o uso do Git e seus comandos para colocar em prática os conceitos vistos até o momento, tais como configurar o Git, criar repositórios, rastrear arquivos, verificar alterações, fazer commit, entre outras atividades de controle de versão. Ao longo dos exemplos, arquivos simples de código-fonte serão criados para serem versionados pelo Git.

Como configurar Git

O comando git config permite configurar diversas características do Git. A maioria delas tem um valor default que se encaixa na grande maioria dos casos. Entretanto, ao menos o usuário deve ser configurado para que o Git associe, por exemplo, seu nome e e-mail em seus commits.

Para fazer isso, basta executar os comandos git config –global user.name <name> e git config –global user.email <email> .

git config –-global user.name <brunnacroches>

git config –global brunna.brunnacroches <brunnacroches@gmail.com> .

Criando um repositório no Git

O primeiro passo para se trabalhar com controle de versão usando o Git é criar um repositório. Isso pode ser feito com o comando git init dentro do diretório que se deseja transformar em repositório, conforme exemplo a seguir:

```
[user@localhost repository]$ git init
Initialized empty Git repository in /home/user/repository/.git/
```

Esse comando inicializa um diretório em um repositório vazio. O diretório oculto criado .git contém todas as configurações e dados do repositório em questão, tais como links para repositórios remotos, banco de dados com os arquivos versionados, etc. Uma vez criado o repositório, todo o fluxo de trabalho apresentado anteriormente se torna possível.

Rastreando arquivos no Git

O repositório recém-criado ainda não contém nenhum arquivo para ser versionado (rastreado). Com o comando git add pode-se adicionar todos os arquivos do repositório no index (área de preparação) para o próximo commit e, quando finalmente o referido comando for executado, os arquivos passarão a ser rastreados.
O index é um snapshot das alterações do diretório de trabalho corrente e dita o conteúdo do próximo commit, por isso, após qualquer alteração é necessário adicioná-la no index, caso contrário a mesma não será gravada no repositório após commit. Esse comando adiciona todas as alterações nos diretórios abaixo do repositório, mas é possível especificar a adição de arquivos únicos informando o nome e o caminho do arquivo ao invés do ponto, como por exemplo, git add /home/user/repository/MinhaClasse.java.
Como exemplo, ao criar o arquivo MinhaClasse.java e executar o comando git status , que informa o atual estado do diretório de trabalho, obtém-se o resultado da **Listagem 1**.
Listagem 1. Estado atual do arquivo MinhaClasse.java

```
[user@localhost repository]$ git status

On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

          MinhaClasse.java

nothing added to commit but untracked files present (use "git add" to track)
```

Isso significa que houve alterações no diretório de trabalho, mas essas mudanças não foram adicionadas no index e, portanto, nada será adicionado ao repositório em caso de commit. Para fazer o commit desse arquivo é necessário, antes de tudo, adicioná-lo por meio do comando git add. Após adicionar o arquivo MinhaClasse.java o status agora é o mesmo apresentado na **Listagem 2**.

Listagem 2. Estado atual do arquivo MinhaClasse.java após o comando add

```
[user@localhost repository]$ git add MinhaClasse.java
[user@localhost repository]$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   MinhaClasse.java
```

Agora há alterações no index prontas para serem adicionadas ao repositório. MinhaClasse.java se tornou um arquivo pronto para ser adicionado ao repositório e ter seu versionamento iniciado.

O próximo tópico detalhará o comando de commit, mas antes note a dica do Git para remover arquivos do index: git rm --cached <file>. Caso o arquivo seja removido do index, ele não será adicionado ao repositório.

Git Commit

O comando git commit armazena o atual conteúdo do index em um novo commit no repositório juntamente com uma mensagem definida pelo usuário descrevendo as alterações realizadas.

Em linhas gerais, a partir do momento em que o commit é realizado, o arquivo passa a ter suas alterações rastreadas, sendo possível visualizar as mesmas ao longo do tempo, comparar diferentes versões e até voltar para versões anteriores.

Para fazer o commit do arquivo MinhaClasse.java descrevendo as alterações com uma mensagem indicando se tratar do primeiro commit, é preciso executar o comando git commit -m “primeiro commit”. Na **Listagem 3** é apresentado o resultado.

Listagem 3. Resultado do comando commit

```
[user@localhost repository]$ git commit -m "primeiro commit"
[master (root-commit) 9026f18] primeiro commit
 1 file changed, 3 insertions(+)
 create mode 100644 MinhaClasse.java
```

Nesse caso o commit foi executado com sucesso e um arquivo foi alterado (MinhaClasse.java que foi adicionado ao repositório), já que houve três inserções (este arquivo contém três linhas e, cada linha inserida entra nessa contagem).

A partir de agora esse arquivo passa a ser rastreável e está no repositório local do Git, onde é possível consultar todo o seu histórico.

Histórico: Git commit history

Um dos usos mais importantes para controladores de versão é possibilitar o controle das versões, mostrando o histórico de evolução de um arquivo ou de um conjunto de arquivos por commits.

O comando git log <file> possibilita visualizar o histórico dos commits por arquivo e o comando git log mostra todos eles.

Após alteração e commit da classe MinhaClasse.java, na qual foi adicionada uma variável, o comando git log MinhaClasse.java mostra o seguinte histórico, apresentado a **Listagem 4**.

Listagem 4. Histórico após o comando git

```
[user@localhost repository]$ git log MinhaClasse.java
commit 10a1f9cb0a490a012dcb625c0571a1307aa41624
Author: Gabriel Amorim <gabriel@email.com>
Date:   Tue May 13 17:42:52 2017 -0300
```

```
    //adição de constante na MinhaClasse.java

commit 9026f184ed12772e8480ca59056eb3c29223a808
Author: Gabriel Amorim <gabriel@email.com>
Date:   Tue May 13 17:14:02 2017 -0300

    primeiro commit
```

Note que cada commit é seguido por um código hash, que é gerado considerando o conteúdo de todo o commit. Dessa forma, cada um tem seu próprio hash e, consequentemente, é impossível perder qualquer alteração ou ter arquivos corrompidos sem que o Git seja capaz de detectar. Voltando ao histórico, veja que cada commit é apresentado mostrando o autor do mesmo, a data e um comentário descrevendo o seu motivo. Assim, é muito fácil consultar o histórico e a evolução de um arquivo ou de toda a base de código.

Comandos do Git ao se trabalhar em equipe

Os comandos apresentados nos tópicos anteriores são bastante úteis ao se trabalhar localmente e fazem parte dos comandos core do Git. No entanto, quando se trabalha em uma equipe que mantém um repositório remoto para versionamento, são necessários novos comandos para atuar nesse cenário. Os próximos tópicos apresentam os comandos mais úteis para se trabalhar em equipe no Git.

Git Clone

O cenário mais comum é uma equipe ter um repositório remoto ao qual todos têm acesso para enviar seus commits. Nesse caso, antes de começar a desenvolver, é necessário baixar o repositório remoto e isso é feito com o comando git clone <caminho_para_o_repositório>. Este faz uma cópia do trabalho local de um repositório. O exemplo da **Listagem 5** faz o clone de um repositório do GitHub – serviço que mantém diversos repositórios online – com o comando git clone https://github.com/gabrielamorim/Java_repository.git.

Listagem 5. Clone de um repositório do GitHub

```
[user@localhost repository]$ git clone https://github.com/gabrielamorim/Java_repository.git
Cloning into 'Java_repository'...
remote: Counting objects: 126, done.
remote: Total 126 (delta 0), reused 0 (delta 0), pack-reused 126
Receiving objects: 100% (126/126), 28.53 KiB | 0 bytes/s, done.
Resolving deltas: 100% (35/35), done.
Checking connectivity... done.
```

Após realizar o clone do repositório, pode-se realizar quaisquer alterações nos códigos baixados. Quando as alterações forem finalizadas, o comando git status mostrará as mesmas no diretório de trabalho, que devem ser adicionadas ao index com o comando git add e, finalmente deve-se fazer o commit com o comando git commit -m “mensagem do commit” para armazenar as alterações no repositório local e poder rastrear localmente o trabalho realizado.

Enviando e recebendo alterações no repositório remoto

As alterações realizadas e que forem adicionadas no repositório por meio do commit estarão apenas no repositório local. Como o trabalho é em equipe, as alterações locais devem ser enviadas para o repositório remoto para que todos os membros da equipe também tenham acesso e possam atualizar seus repositórios locais com as últimas alterações de outros membros da equipe.

Para isso utiliza-se o comando git push, que envia todos os commits do repositório local para o repositório remoto. O exemplo da **Listagem 6** mostra o resultado da execução do git push. Para enviar os commits para o repositório remoto é necessário fornecer as credencias de acesso.

Listagem 6. Resultado da execução do git push

```
[gamorim@localhost Java_repository]$ git push
Username for 'https://github.com': username
Password for 'https://novais.amorim@gmail.com@github.com':
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 491 bytes | 0 bytes/s, done.
Total 5 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
To https://github.com/gabrielamorim/Java_repository.git
   bf03c8d..35ffd16  master -> master
```

Da mesma forma que alterações são enviadas para o repositório remoto, em algum momento será necessário fazer o update do repositório local para baixar as últimas alterações disponíveis no repositório remoto. Isso é feito com o

comando git pull, que simplesmente baixa todo o conteúdo atualizado do repositório remoto para o repositório local. Ao tentar executar o git pull no repositório https://github.com/gabrielamorim/Java_repository.git, obtém-se o resultado a seguir:

```
[user@localhost Java_repository]$ git pull
Already up-to-date.
```

Esse resultado significa que não há alterações no repositório remoto e que o local já está atualizado. Entretanto, caso houvessem alterações, elas seriam baixadas e o Git mostraria os arquivos alterados.

Git Branches e tags

Branches são ramificações, como uma cópia de um diretório de trabalho, criados para se desenvolver funcionalidades de forma isoladas uns das outros. Todo repositório Git se inicia com um branch padrão (master). O principal fluxo de trabalho consiste em criar novos branches para desenvolver uma funcionalidade e, quando finalizar o desenvolvimento, mesclar (merge) o branch criado com o principal.

Um branch pode ser criado com o comando git checkout -b <nome_do_novo_branch>.

O exemplo da **Listagem 7** mostra a criação de um novo branch.

Listagem 7. Criação de um novo branch

```
[user@localhost repository]$ git branch
* master
[user@localhost repository]$ git checkout -b funcionalidade_xyz
Switched to a new branch 'funcionalidade_xyz'
[user@localhost repository]$ git branch
* funcionalidade_xyz
  master
```

Primeiro foi executado o comando git branch para identificar o branch atual. Nesse caso, o único existente era o master, marcado com um asterisco para indicar o branch corrente. O comando git checkout -b funcionalidade_xyz criou um novo branch chamado funcionalidade_xyz. O Git então mudou para o novo branch e, uma nova execução do comando git branch mostrou a existência de dois branches, sendo o atual (aquele em que as alterações serão aplicadas) como sendo a funcionalidade_xyz.

Ao fazer uma alteração em algum arquivo desse novo branch, como o MinhaClasse.java, todas as mudanças serão adicionadas a esse branch.

Para voltar ao branch master utilize o comando git checkout master. Ao verificar o arquivo MinhaClasse.java do branch master, pode-se ver que não há nenhuma alteração, pois as mudanças foram feitas no funcionalidade_xyz. Agora é preciso fazer o merge dos dois branches, que é feito com o comando git merge <nome_do_branch>. Este mescla as alterações realizadas no branch onde ocorreram as alterações com o branch corrente que, no caso, é o principal. Ao avaliar o arquivo MinhaClasse.java agora, é possível ver que as alterações do funcionalidade_xyz agora estão no branch master. O Git sabe exatamente como fazer o merge caso não haja conflitos, no entanto, quando houver divergências que ele não saiba como resolver, o mesmo exibirá as diferenças entre os branches e solicitar o merge manual, como mostra a **Listagem 8.**

Listagem 8. Diferenças entre os branches

```
[user@localhost repository]$ git merge funcionalidade_xyz
Updating 10a1f9c..a98668a
Fast-forward
 MinhaClasse.java | 1 +
 1 file changed, 1 insertion(+)
 create mode 160000 Java_repository
```

Os branches auxiliam no desenvolvimento paralelo de funcionalidades, não impactando no código do sistema em produção ou a versão mais próxima de produção, que geralmente é o que está no branch master. O merge de um branch paralelo com o principal geralmente é devido a inclusão de uma funcionalidade ou correção de bug, então convém criar um ponto de release, ou seja, uma marcação indicando a versão do software a partir daquele commit, chamada de tag, e sua criação é feita com o comando git tag <nome_da_tag> <identificador_do_commit>, como no exemplo a seguir:

```
[user@localhost repository]$ git tag 1.0.0 a98668ab880ba9059cf587de851b989f09c71fbb
```

O nome da tag é o identificador da mesma para consultas, que geralmente é o número de versão seguindo a convenção da equipe. O identificador do commit é o hash do commit que originou a criação dessa tag, que pode ser obtido por meio do histórico.

O fluxo de trabalho das ferramentas controladoras de versão tendem a ser mal interpretadas, especialmente quando acontece a migração de uma ferramenta para outra, fazendo com que os desenvolvedores continuem a utilizar a nova ferramenta com os mesmos conceitos da antiga. Isso acontece especialmente com o Git, que tem um fluxo de trabalho um pouco diferente das ferramentas mais utilizadas até então, fazendo o Git ser taxado até mesmo de muito complexo. Os recursos apresentados nesse artigo são muito úteis para facilitar e melhorar o controle de versão e possibilitam o uso básico e corriqueiro da ferramenta de forma correta. Entender o Git corretamente é o primeiro passo de um longo caminho para o aperfeiçoamento do ambiente de desenvolvimento.

1.3 Lista de comandos úteis do GIT

Web Clip

leocomelli / **git.md**

Last active 2 hours ago • Report abuse

Code | Revisions 5 | Stars 1,548 | Forks 791

Lista de comandos úteis do GIT

git.md — Raw

GIT

Estados

- Modificado (modified);
- Preparado (staged/index)
- Consolidado (comitted);

Ajuda

Geral

```
git help
```

Comando específico

```
git help add
git help commit
git help <qualquer_comando_git>
```

Configuração

Geral

As configurações do GIT são armazenadas no arquivo **.gitconfig** localizado dentro do diretório do usuário do Sistema Operacional (Ex.: Windows: C:\Users\Documents and Settings\Leonardo ou *nix /home/leonardo).

As configurações realizadas através dos comandos abaixo serão incluídas no arquivo citado acima.

Setar usuário

```
git config --global user.name "Leonardo Comelli"
```

Setar email

```
git config --global user.email leonardo@software-ltda.com.br
```

Setar editor

```
git config --global core.editor vim
```

Setar ferramenta de merge

```
git config --global merge.tool vimdiff
```

Setar arquivos a serem ignorados

```
git config --global core.excludesfile ~/.gitignore
```

Listar configurações

```
git config --list
```

Ignorar Arquivos

Os nomes de arquivos/diretórios ou extensões de arquivos listados no arquivo **.gitignore** não serão adicionados em um repositório. Existem dois arquivos .gitignore, são eles:

- Geral: Normalmente armazenado no diretório do usuário do Sistema Operacional. O arquivo que possui a lista dos arquivos/diretórios a serem ignorados por **todos os repositórios** deverá ser declarado conforme citado acima. O arquivo não precisa ter o nome de **.gitignore**.
- Por repositório: Deve ser armazenado no diretório do repositório e deve conter a lista dos arquivos/diretórios que devem ser ignorados apenas para o repositório específico.

Repositório Local

Criar novo repositório

```
git init
```

Verificar estado dos arquivos/diretórios

```
git status
```

Adicionar arquivo/diretório (staged area)

Adicionar um arquivo em específico

```
git add meu_arquivo.txt
```

Adicionar um diretório em específico

```
git add meu_diretorio
```

Adicionar todos os arquivos/diretórios

```
git add .
```

Adicionar um arquivo que esta listado no .gitignore (geral ou do repositório)

```
git add -f arquivo_no_gitignore.txt
```

Comitar arquivo/diretório

Comitar um arquivo

```
git commit meu_arquivo.txt
```

Comitar vários arquivos

```
git commit meu_arquivo.txt meu_outro_arquivo.txt
```

Comitar informando mensagem

```
git commit meuarquivo.txt -m "minha mensagem de commit"
```

Remover arquivo/diretório

Remover arquivo

```
git rm meu_arquivo.txt
```

Remover diretório

```
git rm -r diretorio
```

Visualizar histórico

Exibir histórico

```
git log
```

Exibir histórico com diff das duas últimas alterações

```
git log -p -2
```

Exibir resumo do histórico (hash completa, autor, data, comentário e qtde de alterações (+/-))

```
git log --stat
```

Exibir informações resumidas em uma linha (hash completa e comentário)

```
git log --pretty=oneline
```

Exibir histórico com formatação específica (hash abreviada, autor, data e comentário)

```
git log --pretty=format:"%h - %an, %ar : %s"
```

- %h: Abreviação do hash;
- %an: Nome do autor;
- %ar: Data;
- %s: Comentário.

Verifique as demais opções de formatação no Git Book

Exibir histório de um arquivo específico

```
git log -- <caminho_do_arquivo>
```

Exibir histórico de um arquivo específico que contêm uma determinada palavra

```
git log --summary -S<palavra> [<caminho_do_arquivo>]
```

Exibir histórico modificação de um arquivo

```
git log --diff-filter=M -- <caminho_do_arquivo>
```

- O pode ser substituido por: Adicionado (A), Copiado (C), Apagado (D), Modificado (M), Renomeado (R), entre outros.

Exibir histório de um determinado autor

```
git log --author=usuario
```

Exibir revisão e autor da última modificação de uma bloco de linhas

```
git blame -L 12,22 meu_arquivo.txt
```

Desfazendo operações

Desfazendo alteração local (working directory)

Este comando deve ser utilizando enquanto o arquivo não foi adicionado na **staged area**.

```
git checkout -- meu_arquivo.txt
```

Desfazendo alteração local (staging area)

Este comando deve ser utilizando quando o arquivo já foi adicionado na **staged area**.

```
git reset HEAD meu_arquivo.txt
```

Se o resultado abaixo for exibido, o comando reset *não* alterou o diretório de trabalho.

```
Unstaged changes after reset:
M       meu_arquivo.txt
```

A alteração do diretório pode ser realizada através do comando abaixo:

```
git checkout meu_arquivo.txt
```

Repositório Remoto

Exibir os repositórios remotos

```
git remote

git remote -v
```

Vincular repositório local com um repositório remoto

```
git remote add origin git@github.com:leocomelli/curso-git.git
```

Exibir informações dos repositórios remotos

```
git remote show origin
```

Renomear um repositório remoto

```
git remote rename origin curso-git
```

Desvincular um repositório remoto

```
git remote rm curso-git
```

Enviar arquivos/diretórios para o repositório remoto

O primeiro **push** de um repositório deve conter o nome do repositório remoto e o branch.

```
git push -u origin master
```

Os demais **pushes** não precisam dessa informação

```
git push
```

Atualizar repositório local de acordo com o repositório remoto

Atualizar os arquivos no branch atual

```
git pull
```

Buscar as alterações, mas não aplica-las no branch atual

```
git fetch
```

Clonar um repositório remoto já existente

```
git clone git@github.com:leocomelli/curso-git.git
```

Tags

Criando uma tag leve

```
git tag vs-1.1
```

Criando uma tag anotada

```
git tag -a vs-1.1 -m "Minha versão 1.1"
```

Criando uma tag assinada

Para criar uma tag assinada é necessário uma chave privada (GNU Privacy Guard - GPG).

```
git tag -s vs-1.1 -m "Minha tag assinada 1.1"
```

Criando tag a partir de um commit (hash)

```
git tag -a vs-1.2 9fceb02
```

Criando tags no repositório remoto

```
git push origin vs-1.2
```

Criando todas as tags locais no repositório remoto

```
git push origin --tags
```

Branches

O **master** é o branch principal do GIT.

O **HEAD** é um ponteiro *especial* que indica qual é o branch atual. Por padrão, o **HEAD** aponta para o branch principal, o **master**.

Criando um novo branch

```
git branch bug-123
```

Trocando para um branch existente

```
git checkout bug-123
```

Neste caso, o ponteiro principal **HEAD** esta apontando para o branch chamado bug-123.

Criar um novo branch e trocar

```
git checkout -b bug-456
```

Voltar para o branch principal (master)

```
git checkout master
```

Resolver merge entre os branches

```
git merge bug-123
```

Para realizar o *merge*, é necessário estar no branch que deverá receber as alterações. O *merge* pode automático ou manual. O merge automático será feito em arquivos textos que não sofreram alterações nas mesmas linhas, já o merge manual será feito em arquivos textos que sofreram alterações nas mesmas linhas.

A mensagem indicando um *merge* manual será:

```
Automerging meu_arquivo.txt
CONFLICT (content): Merge conflict in meu_arquivo.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Apagando um branch

```
git branch -d bug-123
```

Listar branches

Listar branches

```
git branch
```

Listar branches com informações dos últimos commits

```
git branch -v
```

Listar branches que já foram fundidos (merged) com o master

```
git branch --merged
```

Listar branches que não foram fundidos (merged) com o master

```
git branch --no-merged
```

Criando branches no repositório remoto

Criando um branch remoto com o mesmo nome

```
git push origin bug-123
```

Criando um branch remoto com nome diferente

```
git push origin bug-123:new-branch
```

Baixar um branch remoto para edição

```
git checkout -b bug-123 origin/bug-123
```

Apagar branch remoto

```
git push origin:bug-123
```

Rebasing

Fazendo o **rebase** entre um o branch bug-123 e o master.

```
git checkout experiment

git rebase master
```

Mais informações e explicações sobre o Rebasing

###Stash

Para alternar entre um branch e outro é necessário fazer o commit das alterações atuais para depois trocar para um outro branch. Se existir a necessidade de realizar a troca sem fazer o commit é possível criar um **stash**. O Stash como se fosse um branch temporário que contem apenas as alterações ainda não commitadas.

Criar um stash

```
git stash
```

Listar stashes

```
git stash list
```

Voltar para o último stash

```
git stash apply
```

Voltar para um stash específico

```
git stash apply stash@{2}
```

Onde **2** é o indíce do stash desejado.

Criar um branch a partir de um stash

```
git stash branch meu_branch
```

Reescrevendo o histórico

Alterando mensagens de commit

```
git commit --amend -m "Minha nova mensagem"
```

Alterar últimos commits

Alterando os três últimos commits

```
git rebase -i HEAD~3
```

O editor de texto será aberto com as linhas representando os três últimos commits.

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added catfile
```

Altere para edit os commits que deseja realizar alterações.

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added catfile
```

Feche o editor de texto.

Digite o comando para alterar a mensagem do commit que foi marcado como *edit*.

```
git commit –amend –m “Nova mensagem”
```

Aplique a alteração

```
git rebase --continue
```

Atenção: É possível alterar a ordem dos commits ou remover um commit apenas mudando as linhas ou removendo.

Juntando vários commits

Seguir os mesmos passos acima, porém marcar os commtis que devem ser juntados com **squash*

Remover todo histórico de um arquivo

```
git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
```

Bisect

O bisect (pesquisa binária) é útil para encontrar um commit que esta gerando um bug ou uma inconsistência entre uma sequência de commits.

Iniciar pequinsa binária

```
git bisect start
```

Marcar o commit atual como ruim

```
git bisect bad
```

Marcar o commit de uma tag que esta sem o bug/inconsistência

```
git bisect good vs-1.1
```

Marcar o commit como bom

O GIT irá navegar entre os commits para ajudar a indentificar o commit que esta com o problema. Se o commit atual não estiver quebrado, então é necessário marca-lo como **bom**.

```
git bisect good
```

Marcar o commit como ruim

Se o commit estiver com o problema, então ele deverá ser marcado como **ruim**.

```
git bisect bad
```

Finalizar a pesquisa binária

Depois de encontrar o commit com problema, para retornar para o *HEAD* utilize:

```
git bisect reset
```

Contribuições

Sinta-se a vontade para realizar adicionar mais informações ou realizar correções. Fork me!

 **dalmosantos** commented on 3 Apr 2016 •••

Opa, tudo bem?

Por gentileza, se possível, teria como adicionar o comando, para altear a url do git remote:

git remote set-url origin git@github.com:leocomelli/outro-curso-git.git

Abraços

 **clebiovieira** commented on 19 Feb 2017 •••

Ótimo guia.

ezequiel88 commented on 7 Mar 2017

como saber o tamanho de um repositorio? tem comando pra isso?

rrrose commented on 18 Apr 2017

Seu guia está perfeito! Me ajudou muito!!! Vou divulgá-lo!

fontnelle commented on 20 Apr 2017

Ótimo guia parabéns!!

maellson commented on 24 Apr 2017

TOP!! muito bom!

lcarlosilva commented on 14 Jun 2017

Parabéns pela iniciativa **@leocomelli** ! Muito TOP esse guia.

tuchinski commented on 27 Jun 2017

Muito bom!! Me ajudou e ainda ajuda muito. Parabéns!

carlosvasconcelosjr commented on 11 Jul 2017

Mão na roda! Muito bom. Parabéns pela iniciativa.

GraniteConsultingReviews commented on 28 Aug 2017

Thanks for sharing this help me a lot

pedromoraesh commented on 12 Sep 2017

Adicione o comando "git config --get remote.origin.url", é util para pegar a URL do repositório, principalmente no uso em Shell Scripts

noglipe commented on 12 Sep 2017

Muito Bom

ArieltonPires commented on 16 Sep 2017

very nice!

nuneswesley commented on 1 Oct 2017

Muito bom!
Mas gostaria de deixar uma observação: sobre o comando **git fecth** seria **git fetch**.
Obrigado

silv4b commented on 19 Oct 2017

Muito bom o seu guia amigo, aprendi bastante! Irei divulgar.

leylson commented on 31 Oct 2017

Ótimo trabalho!

RickieSouza commented on 11 Jan 2018

Muito bom!

wmsilva commented on 24 Jan 2018

Muito bom material me ajudou muito.

DanielDonizetttibatista commented on 3 Feb 2018

mo bom mas o meu está escrevendo: to many arguments
o que estou fazendo de errado?

RodrigoTenorio86 commented on 21 Feb 2018

mto grato.

paulloandrade commented on 14 Mar 2018

Muito bom esse guia de comandos, obrigado !!

demacdolincoln commented on 23 Mar 2018

primeiramente: simples, direto, bem completo, realmente parabéns

segundamente: meus 2 dedos de contribuição ~> exemplo de como adicionar um alias no git:

```
git config --global alias.graph 'log --graph --oneline --all --decorate'
```

daí basta ir num repositório cheio de branches e fazer `git graph` para ver um log mostrando os caminhos de cada branch

(vi isso na archwiki)

erikrossetti commented on 27 Mar 2018 • edited

Obrigado por compartilhar!
Faltou o comando para remover o diretório adicionado por engano:
`git rm --cached -r directory-name`

samuelrvg commented on 3 Apr 2018

Existe algum comando que me possibilita enviar apenas um trecho de código de um arquivo, e não o arquivo todo que foi modificado?

No sourcetree existe essa possibilidade, tem como fazer no git ?

BrunoRebelato commented on 23 Apr 2018

Obrigado por compartilhar , grato !!!!

sabrinabm commented on 5 May 2018

Muito útil, obrigada !

faellesales commented on 9 May 2018

Parabéns, muito bom!

lemenezes commented on 17 May 2018

Valeu demais!!!!

filipenickel commented on 21 May 2018

TOp

cianortetem commented on 27 May 2018

Olá tudo bem?
como eu consigo resolver esse problema.
tenho uma pasta, dentro dela existe outras pastas e arquivos variados,
em uma determinada pasta está o conteúdo que eu preciso compartilhar
mais quando eu faço o push, o conteúdo desta pasta não sobe,
e no github fica um ícone de duas pastinhas com o nome do commit,
mais a pasta esta vazia, não sobe o conteúdo dela e no git status não mostra
modificações desta pasta e apresenta

pasta(modified content, untracked content)

se puder me ajudar eu agradeço.

Diogo-Nobrega commented on 29 May 2018

Muito boa essa lista de comandos. Obrigado por compartilhar.

isweluiz commented on 7 Jun 2018

Tks

NicolasWoitchik commented on 2 Jul 2018

Muito bom a lista de comandos.
Teria algum comando para enviar um único arquivo quando o "push to master" em um servidor remoto fosse disparado ?
Tipo em um site se eu fizesse uma alteração no arquivo index.html e fizesse um commit deste arquivo e logo após o commit eu fizesse um push to master ao invés de enviar todos os arquivos, enviasse somente a index.html ?

Não sei se deu pra entender, qualquer coisa eu tento explicar melhor...
Obrigado desde já :)

NicolasWoitchik commented on 3 Jul 2018

Muito bom a lista de comandos.
Teria algum comando para enviar um único arquivo quando o "push to master" em um servidor remoto fosse disparado ?
Tipo em um site se eu fizesse uma alteração no arquivo index.html e fizesse um commit deste arquivo e logo após o commit eu fizesse um push to master ao invés de enviar todos os arquivos, enviasse somente a index.html ?

Não sei se deu pra entender, qualquer coisa eu tento explicar melhor...
Obrigado desde já :)

rafaelturquia commented on 12 Jul 2018

O que é pull request? Valeu

carolpandim commented on 17 Jul 2018

Adorei, obrigada!!!

Alexlopesdev commented on 22 Jul 2018

muito bom

DiegoSantosWS commented on 7 Aug 2018

Guia me ajuda muito, ele está como uma das referências no manual interno aqui da empresa parabéns!

AdrianoRabello commented on 12 Aug 2018

Parabéns amigo. A melhor explicação e post que já vi sobre git !

leticiacamposs2 commented on 18 Aug 2018

Que guia maravilhoso, já salvei salvei nos meus favoritos <3

mauroscs commented on 23 Aug 2018

Parabéns Leonardo!!! Seu documento me ajudou muito!! Estou iniciando agora com o Git e confesso que a ferramenta é top!!!
Por gentileza, por acaso você sabe como posso utilizar os comandos "Ctrl C e Ctrl V" no terminal Git?

cleefsouza commented on 24 Aug 2018

Parabéns, ótimo guia

rnrnshn commented on 28 Aug 2018

como adicionar files/diretorios num commit ja feito??

BergSabbath commented on 24 Sep 2018

esse guia eh meu salvador.. sempre recorro a ele.. obrigado!

marcelopoars commented on 24 Sep 2018

Show de bola! Parabéns.

lfcfernando commented on 28 Sep 2018

Muito bom, obrigado por compartilhar seu conhecimento!

felipesilvadesign commented on 10 Jan 2019

cool

sartorileonardo commented on 17 Jan 2019

Gostei da lista, parabéns!

StarleyDev commented on 1 Feb 2019

Amigo o seguinte codigo esta errado...

`git fecth`

Corrigindo :

`git fetch`

lucasblk92dev commented on 1 Feb 2019

olá eu consigo abrir o sublime 3 normalmente com o comando subl, mas na hora de fazer o commit, o editor que abre ainda e o 'vim', ja tentei varios tutoriais na net, consegue me ajudar a resolver? obrigado!

sukenn commented on 24 Feb 2019

vlw man

felipesudrj commented on 14 Mar 2019

Alguem conhece uma forma de empacotar as mudanças de um commite e gerar um zip com elas?
partindo do cenário que tenho que enviar tudo por FTP apos concluir minhas atividades.

RonanUFPa commented on 2 Apr 2019

Excelente, Obrigado!

RobertaMelo commented on 4 Apr 2019

Muito top mesmo! Obrigada :)

galloaleonardo commented on 11 Apr 2019

Uma pequena contribuição aqui:

Puxar um commit específico de um branch para outro:
git cherry-pick <commit hash (SHA-1)>

Exemplo:
`git cherry-pick 8f801338069f66e3c06cfed93a9eaff43bda8jn`

marcuspereiradev commented on 27 Apr 2019

Muito bom!
Tem algum comando onde eu possa ver, depois de adicionar a chave ssh no github, se estou conectado ou não?

jhonalves1026 commented on 15 May 2019

Como posso executar o código do github, para ver-lo funcionando???

caiomyrapereira commented on 15 Jun 2019

vlw ai man.

Maxuelreis commented on 19 Jul 2019

Conteúdo de muitíssima utilidade. vlw bro.

RodrigoStuani commented on 1 Sep 2019

Me ajudou bastante.. Muito obrigado! o>

alexandresys commented on 2 Sep 2019

show ajudou bastante.. Muito obrigado! valeu

thiagotancredi commented on 9 Sep 2019

Muito bom!! Grato pela lista.

maykonsousa commented on 13 Sep 2019

T acessando mais essa página que o facebook. apanho demais nesse github veeei. Muito top o seu guia

alanrepo commented on 19 Sep 2019

Sugiro uma pequena correção na linha 106 :

alanrepo revised this gist 9 minutes ago. 1 changed file with 1 addition and 1 deletion.

2 git.md

```
@@ -103,7 +103,7 @@ Os nomes de arquivos/diretórios ou extensões de arquivos listados no arquivo *
    git rm -r diretorio
-  ### Visualizar hitórico
+  ### Visualizar histórico
   ##### Exibir histórico
```

https://gist.github.com/alanrepo/e750cccd2aa01379468fcfe59dbc9e75

ThiagoFranco25 commented on 14 Nov 2019

vlw meu consagrado

eddymaxs commented on 3 Jan 2020

Ajuda muito no meu aprendizado! Obrigado e Parabéns !

filipenonato commented on 6 Jan 2020

Parabéns pela contribuição, vai ajudar muita gente no aprendizado.

Felipe-Marques commented on 7 Jan 2020

Obrigado pela lista, me ajudou muito no curso.

ghost commented on 14 Jan 2020

Muito bom

flads commented on 17 Jan 2020

Thank's!!!

mauricio178 commented on 18 Jan 2020

Ótimo guia de comandos

agpinheiro commented on 27 Jan 2020

Muito top!! Vlw pela ajuda!

Alexsandra127 commented on 6 Feb 2020

Thanks!!!

Felipe-Marques commented on 7 Feb 2020

@Fabricio-Guima uma lista de comandos que pode ajudar muito.

Felipe-Marques commented on 7 Feb 2020

> Parabéns Leonardo!!! Seu documento me ajudou muito!! Estou iniciando agora com o Git e confesso que a ferramenta é top!!!
> Por gentileza, por acaso você sabe como posso utilizar os comandos "Ctrl C e Ctrl V" no terminal Git?

@mauroscs boa tarde, você consegue usar o CTRL C para copiar quando seleciona o trecho desejado, caso contrário, CTRL C pula para outra linha de comando, o CTRL V funciona normalmente para colar. Estou baseando no Bash.

Felipe-Marques commented on 7 Feb 2020

> Como posso executar o código do github, para ver-lo funcionando???

@jhonalves1026 boa tarde, vamos dizer que você tenha um repositório no github com o html, css e uma pasta de imagens da sua página web, seu site está pronto e você deseja visualiza-lo pela hospedagem do github, você precisará ir no settings do seu repositório em seguida achar o github pages, em sources selecione a branch do seu projeto (normalmente a master branch), o github irá liberar um link para visualização do site hospedado: https://jhonalves1026.github.io/repositório/arquivo.html (se seu arquivo html estiver nomeado como index.html será reconhecido automaticamente ex: https://jhonalves1026.github.io/repositório/).

Espero ter ajudado em algo.
Abraço.

alexandresys commented on 7 Feb 2020

Fala meu amigo td bem cara esse código, funciona mas ele foi feito em caráter de estudo. Então eata bem básico. Valeu abc

Em 7 de fev de 2020 12:18, Felipe Marques <notifications@github.com> escreveu:

Como posso executar o código do github, para ver-lo funcionando???

@jhonalves1026<https://github.com/jhonalves1026> boa tarde, vamos dizer que você tenha um repositório no github com o html, css e uma pasta de imagens da sua página web, seu site está pronto e você deseja visualiza-lo pela hospedagem do github, você precisará ir no settings do seu repositório em seguida achar o github pages, em sources selecione a branch do seu projeto (normalmente a master branch), o github irá liberar um link para visualização do site hospedado: https://jhonalves1026.github.io/repositório/arquivo.html<https://jhonalves1026.github.io/reposit%C3%B3rio/arquivo.html> (se seu arquivo html estiver nomeado como index.html será reconhecido automaticamente ex: https://jhonalves1026.github.io/repositório/<https://jhonalves1026.github.io/reposit%C3%B3rio/>).

Espero ter ajudado em algo.
Abraço.

—
You are receiving this because you commented.
Reply to this email directly, view it on GitHub<https://gist.github.com/2545add34e4fec21ec16?email_source=notifications&email_token=AH2IIRWFVEOEBQ4VRUECH43RBV3UJA5CNFSM4HI5HSDKYY3PNVWWK3TUL52HS4DFVNDWS43UINXW23LFNZ2KUY3PNVWWK3TUL5UWJTQAGBORU#gistcomment-3169562>, or unsubscribe<https://github.com/notifications/unsubscribe-auth/AH2IIRQP5X42IFNEQ2OWLEDRBV3UJANCNFSM4HI5HSDA>.

alexandresys commented on 7 Feb 2020

Fala meu amigo blz ...cara se não me engano tem uma lista de comandos aí no guit.. abc

Em 7 de fev de 2020 12:02, Felipe Marques <notifications@github.com> escreveu:

Parabéns Leonardo!!! Seu documento me ajudou muito!! Estou iniciando agora com o Git e confesso que a ferramenta é top!!!
Por gentileza, por acaso você sabe como posso utilizar os comandos "Ctrl C e Ctrl V" no terminal Git?

@mauroscs<https://github.com/mauroscs> boa tarde, você consegue usar o CTRL C para copiar quando seleciona o trecho desejado, caso contrário, CTRL C pula para outra linha de comando, o CTRL V funciona normalmente para colar. Estou baseando no Bash.

—
You are receiving this because you commented.
Reply to this email directly, view it on GitHub<https://gist.github.com/2545add34e4fec21ec16?email_source=notifications&email_token=AH2IIRTQQJF5E2ICQCUHXHDRBVZYBA5CNFSM4HI5HSDKYY3PNVWWK3TUL52HS4DFVNDWS43UINXW23LFNZ2KUY3PNVWWK3TUL5UWJTQAGBORA#gistcomment-3169552>, or unsubscribe<https://github.com/notifications/unsubscribe-auth/AH2IIRSLQZPXYULRSJIIQ2LRBVZYBANCNFSM4HI5HSDA>.

mramalho commented on 13 Mar 2020 • edited

Olá, alguém poderia me ajudar?

Estou automatizando a geração de versão usando o Jenkins para um software que temos aqui na empresa, mas estou precisando listar os arquivos que foram integrados (merged) na branch master para extraí-los para uma pasta para deploy.

Estes arquivos são .DLL, .RPT, etc... pois o software que utilizamos foi desenvolvido em VB6 e VB.Net, e não estou conseguindo encontrar o comando do git que lista os arquivos que foram integrados na branch master.

Alguém consegue ajudar?

Aproveitando, parabéns pelo guia de comandos.

Marcos Ramalho

Maxmiller-Nunes commented on 31 Mar 2020

Cara, muito obrigado pelo quia, muito simples e entender, direto ao ponto, me ajudou muito. VLW!

romulomoraesti commented on 1 Apr 2020

Muito bom! Parabéns!
Me ajudou bastante, estava precisando.

rogeriodelphi commented on 22 Apr 2020

Tem como eu alterar um projeto de privado para público ou vice-versa?

csferreiraa commented on 23 Apr 2020

> Tem como eu alterar um projeto de privado para público ou vice-versa?

Tem sim **@rogeriodelphi**, segue link detalhando o passo a passo necessário para execução.

https://help.github.com/pt/github/administering-a-repository/setting-repository-visibility

renatoflavio13 commented on 27 Apr 2020

thx bro

weversondf commented on 5 May 2020

Guia prático e funcional! Valew.

01101-kairo commented on 9 May 2020

agradecido man

pamellabiotec commented on 13 May 2020 • edited

Ótimo guia prático do git!

Ferodrigueshk commented on 22 May 2020 • edited

O comando Touch index.html é usado no Linux...
qual é o comando equivalente para o Windows?

steffesonlira commented on 21 Jun 2020

Muito bom, parabéns!!

jadsonecomp commented on 26 Jun 2020

Parabéns, excelente conteúdo

tatmorenno commented on 3 Jul 2020

@Ferodrihueshk

> O comando Touch index.html é usado no Linux...
> qual é o comando equivalente para o Windows?

Você pode utilizar o comando:
copy NUL nome.txt

Ou então se quiser inserir o conteúdo manualmente ainda existe o:
copy con nome.txt

Após dar enter nesse comando copy con, o prompt ficará vazio e tudo o que você digitar será adicionado no arquivo.
Cada enter será uma quebra de linha e quando terminar, dê um F6 e um enter que o arquivo é salvo.

thomas-ferraz commented on 6 Aug 2020 • edited

Estou começando... fiz um diretório teste transformei em repositório e vinculei este repositório local com um repositório remoto. Agora queria apagar tudo da minha máquina! Como faço? Eu desvinculei os repositórios com "git remote rm origin". Alguém sabe como fazer essa exclusão?

Maxmiller-Nunes commented on 6 Aug 2020

Opa beleza?
Se você quer somente tirar o "git" do projeto é so apagar a past .git que ele cria dentro de sua pasta, fazendo isso o git para de monitorar o seu projeto na pasta.

thomas-ferraz commented on 6 Aug 2020

> Opa beleza?
> Se você quer somente tirar o "git" do projeto é so apagar a past .git que ele cria dentro de sua pasta, fazendo isso o git para de monitorar o seu projeto na pasta.

Funcionou! Mas só fazendo pelo cmd! Valeu!

Maxmiller-Nunes commented on 6 Aug 2020

Show maravilha.
Essa pasta .git ele fica ocultada, tem uma opção para mostrar essas pasta, so habilitar que da certo.

gefersoneds commented on 14 Aug 2020

Muito bom.
I loved it, congratulations

claudiomf1 commented on 16 Aug 2020

Qual comando para definir o diretorio inicial do git? alguem poderia escrever um exemplo?

samuelikz commented on 2 Sep 2020

Obrigado!

IsaqueDiniz commented on 13 Sep 2020

Muito obrigado, amigo.
Sempre que esqueço algo venho aqui consultar.

magnamoz commented on 14 Sep 2020

Ótimo guia, obrigada.

danielbarrozim commented on 27 Sep 2020

Muito bom!

silmarasilva commented on 5 Oct 2020

Muito útil, obrigada.

thomaztorres commented on 5 Oct 2020

Muito bom de verdade, obrigado amigo!

xcarlosr commented on 16 Oct 2020

Muito obrigado, um guia bem resumido e direto!

andersonrocha79 commented on 24 Oct 2020

nó... ajudou demais... obrigado!

Raimos03 commented on 14 Dec 2020

Parabéns pela explicação.

newcode25 commented on 14 Jan

Me ajudou esse guia obrigado por cria-lo. Que Deus te ajude a ser uma pessoa que contribui, sempre.

GiliardGodoi commented on 21 Jan

Seria interessante colocar um índice nessa lista?

1. Configuração
2. Repositório Local
3. Repositório Remoto
4. Tags
5. Branches
6. Rebase
7. Stash
8. Reescrevendo o histórico
9. Bisect

mariotlemes commented on 26 Jan • edited

Muito boa a lista mesmo. Parabéns! Concordo com o colega GiliardGodoi, um sumário cairia muito bem. Existe uma ferramenta (doctoc) que faz isso automaticamente no arquivo markdown. Basta instalá-la e depois fazer um "**git doctoc .**". Dê uma olhada, é simples e vai enriquecer teu trabalho.

ptSouthier commented on 8 Feb

> Opa, tudo bem?
>
> Por gentileza, se possível, teria como adicionar o comando, para altear a url do git remote:
>
> git remote set-url origin git@github.com:leocomelli/outro-curso-git.git
>
> Abraços

Vim até aqui buscando exatamente isso, cara! Muito obrigado pela contribuição!!

adriano458 commented on 10 Feb

Muito bom. Obrigado por compartilhar o seu conhecimento

josethiagodev commented on 21 Mar

O melhor GUIA que já vi...
Parabéns, muito bem explicado!!

jsvenancio commented on 4 Apr

Bom tuturial.
Vai ajudar muito.

Re04nan commented on 7 Apr

Parabéns pelo conteúdo, fez um ótimo trabalho.

PedroTPFreitas commented on 8 Apr

Obrigado, sempre que der, vou postar algo novo.

Em qua., 7 de abr. de 2021 às 21:54, Renan Marques ***@******>
escreveu:
...

RonisonMaria commented on 11 Apr

> Muito bom a lista de comandos.
> Teria algum comando para enviar um único arquivo quando o "push to master" em um servidor remoto fosse disparado ?
> Tipo em um site se eu fizesse uma alteração no arquivo index.html e fizesse um commit deste arquivo e logo após o commit eu fizesse um push to master ao invés de enviar todos os arquivos, enviasse somente a index.html ?
>
> Não sei se deu pra entender, qualquer coisa eu tento explicar melhor...
> Obrigado desde já :)

Olá!
Antes de dar o comando git push, no git add você pode especificar somente o arquivo que quer. Para facilitar você pode usar o git gui, na interface do gui você consegue adicionar o arquivo desejado e até mesmo fazer commit e push.

PedroTPFreitas commented on 11 Apr

Boa noite
Tem que fazer o git push, que é para poder atualizar os arquivos no git
hub, senão não dá certo

Em dom, 11 de abr de 2021 21:28, Ronison Matos ***@***.***>
escreveu:
...

pyTonyc commented on 7 May

Top... Tenho usado como guia de referência.

lyssacavalcanti commented 21 days ago

Oi Leonardo! Excelente lista. Vou aproveitar e indicar seu link no meu Instagram @techly.com.br .

educalixto commented 16 days ago

Perfeito ! <3

PedroTPFreitas commented 16 days ago

Vlw
Conhecimento deve ser compartilhado

Em sáb, 22 de mai de 2021 08:53, Eduardo Calixto ***@***.***>
escreveu:
...

Meurer-SG commented 10 days ago

Muito obrigado por compartilhar a lista, Leonardo Comelli (**@leocomelli**), ajudou muito, parabéns!

AleKnot commented 7 days ago • edited

> Adicione o comando "git config --get remote.origin.url", é util para pegar a URL do repositório, principalmente no uso em Shell Scripts

Valeu, me ajudou aqui!

ótima lista **@leocomelli** parabéns

© 2021 GitHub, Inc. Terms Privacy Security Status Docs
Contact GitHub Pricing API Training Blog About

CONTACT

Brunna Croches

Developer Full Stack

brunnacroches.dev

linkedin.com/brunnacroches

github.com/brunnacroches

@brunnacroches.dev

discord.com/brunnacroches

brunnacroches@gmail.com

let's share