



## Controle de Versionamento

Prof. Marcondes Alexandre

### Descrição

Apresentação do conceito de controle de versionamento e do uso do software GitHub para armazenamento e controle do código-fonte em um ambiente de desenvolvimento.

### Propósito

Em um contexto de necessidades empresariais que se alteram com uma velocidade cada vez maior, o uso de controle de versão proporcionará instrumentos para que os membros do time de desenvolvimento possam trabalhar no código-fonte de maneira colaborativa, automatizada e otimizada em suas atividades.

## Objetivos

Módulo 1

### Controle de versão

Identificar o conceito de controle de versão de software.

## Módulo 2

## Introdução ao Git

Reconhecer o software Git e sua instalação.

## Módulo 3

## Comandos básicos do Git

Empregar os comandos básicos do Git.

## Módulo 4

## Operações de Merge, Diff, Branching e Merging

Aplicar o Git para controle de versões de software.



## Introdução

Os desenvolvedores de software utilizam várias teorias, técnicas e ferramentas na elaboração de seus sistemas de informação e essas escolhas são definições feitas pela empresa ou equipe de trabalho em um projeto. No entanto, há vários desafios que os programadores enfrentam no projeto. O versionamento do código-fonte é um dos principais. O uso de um sistema de controle de versão de código-fonte consiste em uma boa prática na engenharia de software porque busca sempre manter o código desenvolvido atualizado no contexto da equipe de desenvolvimento.



## 1 - Controle de versão

Ao final deste módulo, você será capaz de identificar o conceito de controle de versão de software.

# Desenvolvimento de software e controle de versões

Na dinâmica imposta de mudanças e atualizações cada vez mais frequentes e urgentes, a equipe de desenvolvimento de software busca encontrar meios que auxiliem sua produtividade, eficiência e qualidade das entregas mediante alterações de requisitos pelo cliente. Nesse contexto, a adoção de uma ferramenta que pudesse favorecer as atividades de cada membro da equipe de desenvolvimento de maneira a organizar as tarefas e redução parece algo intangível, não é mesmo?

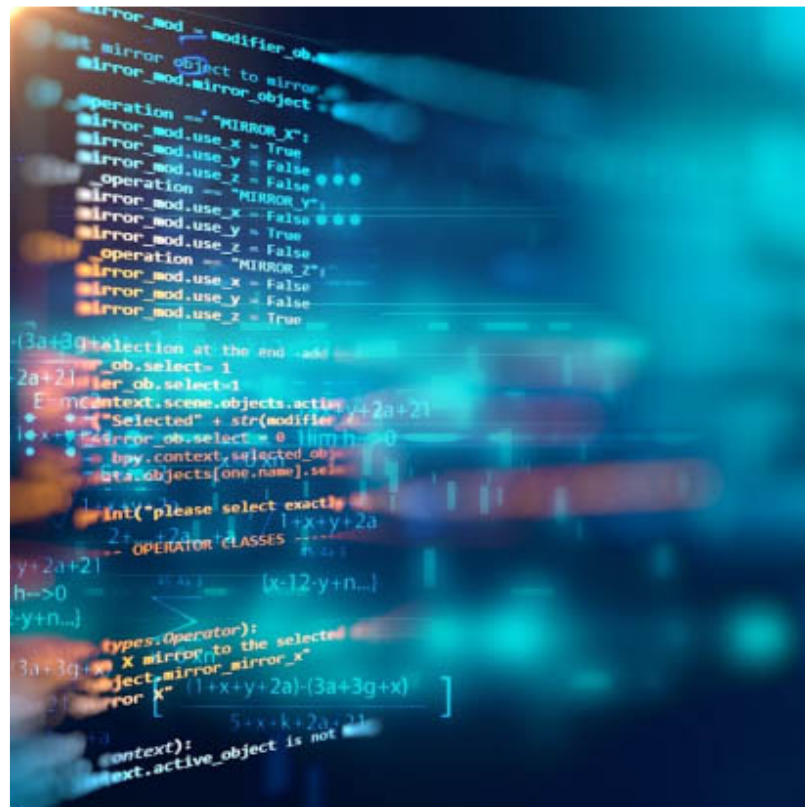
Contudo, uma abordagem ágil na concepção do projeto de software proporciona meios que podem resolver essa questão, dentre deles, encontra-se o **controle de versão**. Com ele, é possível que médias e grandes organizações possam conduzir de maneira harmônica os projetos de desenvolvimento de software seguindo padrões, fluxo de processos e mitigando riscos de segurança em seus sistemas de informação. Neste módulo, discutiremos sobre os conceitos, os

termos e o emprego do controle de versão em equipes de desenvolvimento.

### Comentário

Os sistemas de controle de código-fonte (ou sistemas de controle de versão) permitem que os desenvolvedores colaborem no código e acompanhem as alterações. É possível utilizar o controle de versão para salvar o trabalho realizado por um desenvolvedor, bem como coordenar as alterações de código em uma equipe. O gerenciamento de código-fonte é uma ferramenta essencial para projetos em que, principalmente, haja uma grande quantidade de programadores.

Os **sistemas de controle de versão** mantêm histórico de arquivos, para que você possa revisar facilmente e até mesmo reverter para qualquer versão do seu código. Além disso, ajuda a resolver conflitos ao mesclar contribuições de várias fontes.



Para a maioria das equipes de software, o código-fonte é um repositório de conhecimento. Logo, podemos entender que o uso de controle de versão para um sistema de informação protegerá o código-fonte contra eventuais desastres e erros acidentais por parte dos desenvolvedores, pois, sem controle de versão, possivelmente cada desenvolvedor teria uma cópia do código-fonte em seu próprio computador, desassociado, portanto, de um ponto central de controle e gestão do código produzido. Essa abordagem é extremamente ruim, além de ser perigosa no tocante às alterações individuais feitas no código.

Imagine um cenário em que tenhamos 50 desenvolvedores e cada um tenha uma cópia individualizada do código-fonte de determinado projeto de desenvolvimento de software e que não haja um controle central desse código. Imagine o caos em tratar semelhanças de implementações, como também para consolidar esse código. Só de pensar já ficamos exaustos, não é mesmo? Por isso, adotando um sistema de controle de versão, essa questão estaria plenamente resolvida, porque cada desenvolvedor teria uma versão vinculada a uma cópia principal e haveria mecanismos de sincronização e de resolução de conflitos do código-fonte caso houvesse discrepâncias.

### Atenção!

É evidente que a utilização de uma ferramenta de controle de código sem o alinhamento da condução das atividades de desenvolvimento do software numa abordagem ágil é insuficiente, e que a implementação da integração contínua e entrega são fundamentais num contexto de DevOps. Portanto, o controle de versão consiste em acompanhar todas as alterações nos ativos de software, rastreando e gerenciando quem fez, o que fez e quando fez alterações no código.

O controle de versão é o primeiro passo necessário para garantir a qualidade do código-fonte, garantir o fluxo e extrair valor e concentrar-se no processo. Tudo isso cria valor não apenas para as equipes de software, mas também para o cliente. Sendo assim, inferimos que o controle de versão é uma solução para gerenciar e salvar as alterações feitas em quaisquer ativos criados manualmente. Se forem feitas alterações no código-fonte, você poderá voltar no tempo e reverter facilmente para versões de trabalho anteriores.

As ferramentas de controle de versão permitirão que você veja **quem fez as alterações, quando e o que exatamente foi alterado**. O controle de versão também facilita a experimentação e, mais importante, torna a colaboração possível. Sem controle de versão, a colaboração no código-fonte seria uma operação dolorosa como citamos anteriormente.

## Benefícios do uso do controle de versão

Quer você escreva código profissional ou pessoalmente, você deve sempre usar um sistema de gerenciamento de controle de código-fonte para controlar a versão do seu código.

**Algumas das vantagens de usar o controle de origem são:****Criação de um fluxo de trabalho**

Os fluxos de trabalho de controle de versão evitam confusão para todos que trabalham com seu processo de desenvolvimento com ferramentas diferentes e incompatíveis. Os sistemas de controle de versão fornecem aplicação de processo e permissão para que todos permaneçam com o projeto atualizado.


**Utilização de versão**

Cada versão tem uma descrição na forma de um comentário. Essas descrições ajudam a rastrear alterações de código por versão, em vez de alterações de arquivos individuais. O código armazenado nas versões pode ser visualizado e restaurado do controle de versão a qualquer momento, conforme necessário. Isso facilita o início de novos trabalhos com base em qualquer versão de código.

**Colaboração entre membros**

O controle de versão sincroniza versões e garante que suas alterações não entrem em conflito com outras alterações em sua equipe. Sua equipe conta com o controle de versão para ajudar a resolver e evitar conflitos, mesmo quando as pessoas fazem alterações ao mesmo tempo.

**Preservação do histórico de alterações**



O controle de versão rastreia as alterações quando sua equipe salva uma nova versão do seu código. Você pode visualizar esse histórico para descobrir quem mudou, por que e quando. A história lhe dá a confiança para experimentar porque você sempre pode reverter para uma boa versão anterior. O histórico permite que sua base trabalhe com qualquer versão do código, como corrigir bugs em versões anteriores.

### Automatização de tarefas

Os recursos de automação de controle de versão economizam o tempo da sua equipe e geram resultados consistentes.

## Métodos para controle de versão

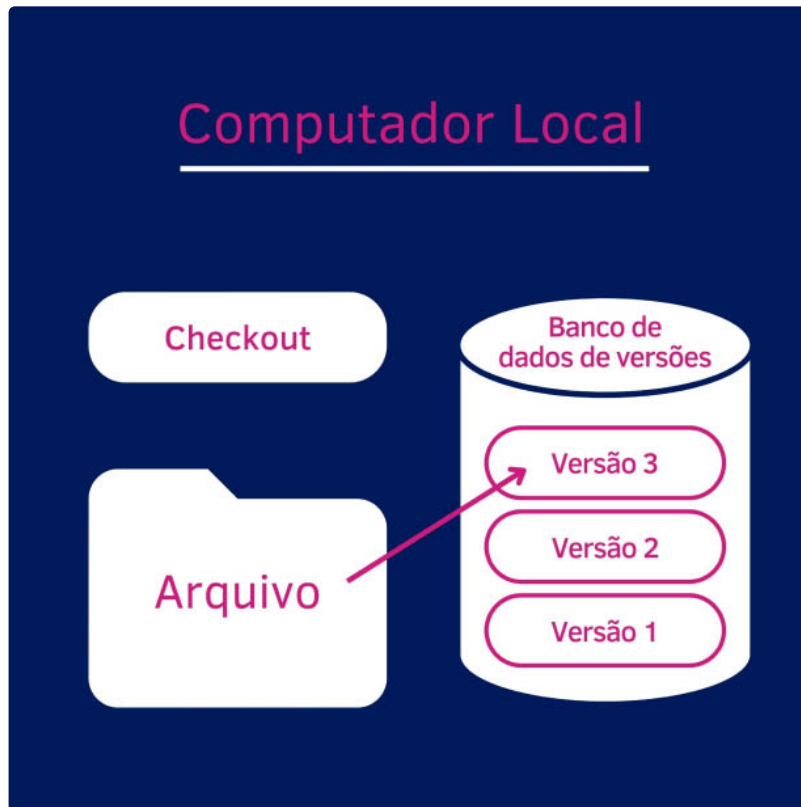
O método de controle de versão preferido de algumas pessoas é copiar arquivos em outro diretório para que possam trabalhar localmente em suas estações de trabalho. Essa abordagem é muito comum porque é muito simples, contudo, também é incrivelmente propensa a erros, além do que a probabilidade de esquecer em qual diretório você está e acidentalmente escrever no arquivo errado ou copiar arquivos que você não queria, indubitavelmente, é maior.

Para lidar com esse problema, os programadores há muito tempo desenvolveram controle de versão local que possui um banco de dados simples e mantém todas as alterações nos arquivos sob controle de revisão.

Uma das ferramentas mais populares foi um sistema chamado Revision Control System (RCS), que é mantido pela GNU software livre, que ainda é disponibilizado para uso, sendo possível baixar o programa no site [gnu.org](https://gnu.org). A figura a seguir ilustra seu uso em um computador local. A ferramenta usa o conceito de checkout para controlar, em seu banco de dados, o armazenamento das versões daquele arquivo.

gnu.org

<https://www.gnu.org/software/rcs/>



Controle de versão em um computador local.

Como citamos anteriormente, um grande desafio era a necessidade de colaborar com desenvolvedores em outros sistemas. Para lidar com esse problema, foram desenvolvidos Sistemas de Controle de Versão Centralizados (CVCS). Esses sistemas, como CVS, Subversion e Perforce, têm um único servidor que contém todos os arquivos versionados e vários clientes que retiram arquivos desse local central. Por muitos anos, esse tem sido o padrão para controle de versão.

Essa configuração oferece muitas vantagens, especialmente sobre controle de versão local. Por exemplo, todos sabem até certo ponto o que todos os outros no projeto estão fazendo. Os administradores têm controle refinado sobre quem pode fazer o quê; e é muito mais fácil administrar um CVC do que lidar com bancos de dados locais em cada estação de trabalho dos desenvolvedores.

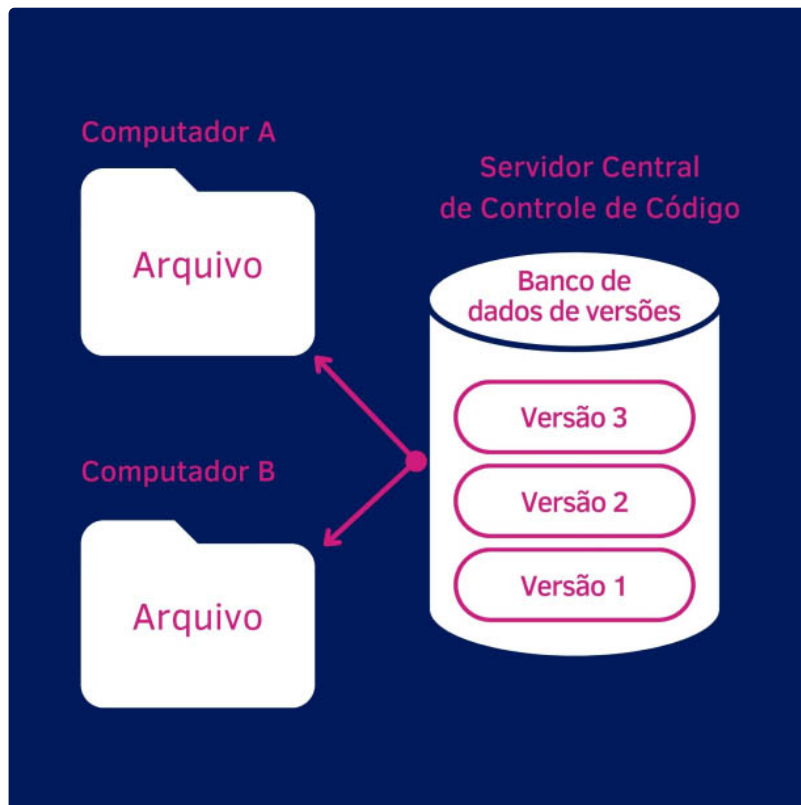
#### Atenção!

No entanto, essa configuração também tem algumas desvantagens sérias. O mais óbvio é o ponto único de falha que o servidor centralizado representa. Se esse servidor ficar inativo por uma hora, durante essa hora ninguém poderá colaborar ou salvar alterações de versão em qualquer coisa em que esteja trabalhando. Se o disco rígido em que está o banco de dados central for corrompido e os backups adequados não



tiverem sido mantidos, você perderá absolutamente tudo – todo o histórico do projeto, exceto os instantâneos únicos que as pessoas tenham em suas máquinas locais.

Os sistemas de controle de código-fonte local sofrem desse mesmo problema – sempre que você tem todo o histórico do projeto em um único lugar, corre o risco de perder tudo. Na figura abaixo, mostra-se como é organizado o sistema de controle de versionamento de código quando há um servidor central para exercer essa função que possui versões locais em computadores dos desenvolvedores de maneira independente.



Controle de versão em um servidor central.

Visando resolver a questão do ponto de falha sendo ainda um servidor central para manutenção dos códigos-fonte, essa abordagem evolui para um contexto mais amplo, sendo, portanto, projetado para funcionar no modelo de sistemas distribuídos que possibilitem o controle de versão. Há muitas ferramentas que atendem essa filosofia no tratamento de código-fonte, como: Git, Mercurial, Bazaar ou Darcs.

Assim, se algum servidor ficar indisponível e esses sistemas estiverem colaborando por meio dele, qualquer um dos repositórios do desenvolvedor poderá ser copiado de volta para o servidor para restaurá-lo. Cada clone é realmente uma cópia completa de todos os dados. Além disso, muitos desses sistemas lidam muito bem com vários repositórios remotos com os quais podem trabalhar, para que você possa colaborar com diferentes times de programadores de maneiras diversas simultaneamente dentro do mesmo projeto. Isso

permite configurar vários tipos de fluxos de trabalho que não são possíveis em sistemas centralizados, como modelos hierárquicos.



## Veja agora os meios para controle de versão

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

### Questão 1

Analise as afirmativas:

I. Os sistemas de controle de versionamento de código-fonte podem ser organizados em dois modelos de repositórios: centralizado e distribuído.

Porque

II. Uma ferramenta para versionamento de código-fonte consiste, principalmente, em um local para armazenamento de artefatos gerados durante o desenvolvimento de sistemas.

Dito isso, assinale a alternativa correta.

A

I é verdadeira, II é verdadeira e a II justifica a I.

B

I é verdadeira, II é verdadeira, porém a II não justifica a I.

C

I e II são falsas.

D

I é falsa e II é verdadeira.

E

I é verdadeira e II é falsa.

Parabéns! A alternativa A está correta.

Em um controle de versionamento de código-fonte, o repositório é o local no qual os arquivos serão armazenados durante o processo de desenvolvimento, podendo ser centralizado ou, ainda, distribuído.

## Questão 2

Analise as afirmativas:

I. Ao usar o modelo centralizado para controle de versão de código-fonte, existe um repositório distribuído com várias cópias de trabalho nas estações de trabalho dos desenvolvedores.

Porque

II. O controle de versão é formado unicamente por dois componentes: o repositório e a área de trabalho. A área de trabalho apresenta todas as versões dos documentos envolvidos no projeto de software.

Dito isso, assinale a alternativa correta.

A

I é verdadeira, II é verdadeira e a II justifica a I.

B

I é verdadeira, II é verdadeira, porém a II não justifica a I.

C

I e II são falsas.

D

I é falsa e II é verdadeira.

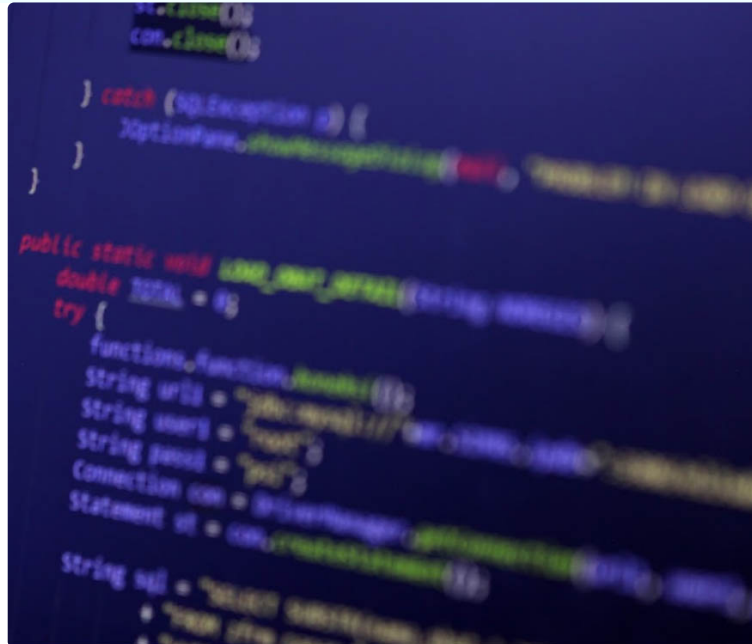
E

I é verdadeira e II é falsa.

**Parabéns! A alternativa E está correta.**

O controle de versão de código-fonte apresenta o repositório como unidade de integração dos artefatos desenvolvidos pelos

programadores, sendo o local que apresenta todas as versões de todos os arquivos envolvidos no projeto.



## 2 - Introdução ao Git

Ao final deste módulo, você será capaz de reconhecer o software Git e sua instalação.

# Software GitHub

O surgimento do Git como sistema de controle de versionamento de código encontra-se entrelaçado com a história do projeto de código aberto do sistema operacional Linux. Durante a maior parte da vida útil da manutenção do kernel do Linux (1991–2002), as alterações no software eram passadas como patches e arquivos armazenados. Em 2002, o projeto do kernel Linux começou a usar um sistema distribuído para versionamento de código-fonte proprietário chamado BitKeeper.

Em 2005, o relacionamento entre a comunidade que desenvolveu o kernel Linux e a empresa comercial que desenvolveu o BitKeeper foi desfeito, e o status gratuito da ferramenta foi revogado. Isso levou a comunidade de desenvolvimento do Linux (e em particular Linus Torvalds, o criador do Linux) a desenvolver sua própria ferramenta com base em algumas das lições que aprenderam ao usar o BitKeeper.

**Alguns dos objetivos do novo sistema foram os seguintes:**

- Velocidade;

- Design simples;
- Forte suporte para desenvolvimento não linear (milhares de ramificações paralelas);
- Totalmente distribuído; e
- Capacidade de lidar com grandes projetos, como o kernel Linux, de forma eficiente (velocidade e tamanho dos dados).

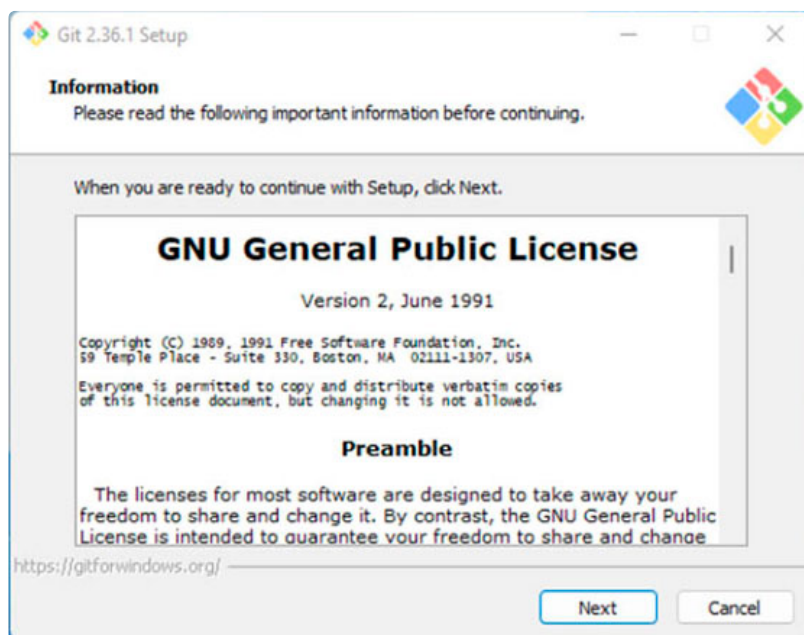
Assim, no ano de 2005, nasce a ferramenta Git, que evoluiu em face das características iniciais da ferramenta BitKeeper, tornando-se incrivelmente rápida e muito eficiente com grandes projetos, além de possuir um incrível sistema de ramificação para desenvolvimento de código-fonte.

## Instalando o GitHub

### Instalando o Git no Windows

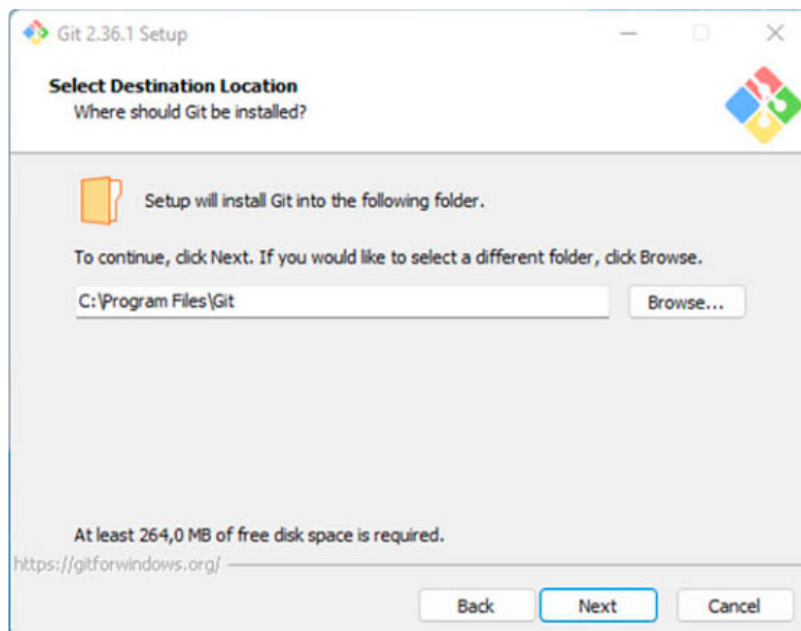
Você pode proceder a instalação no sistema operacional Windows após o download da versão do Git versão 64 bits seguindo os seguintes passos.

Na imagem a seguir, clique em Next para iniciar a instalação.



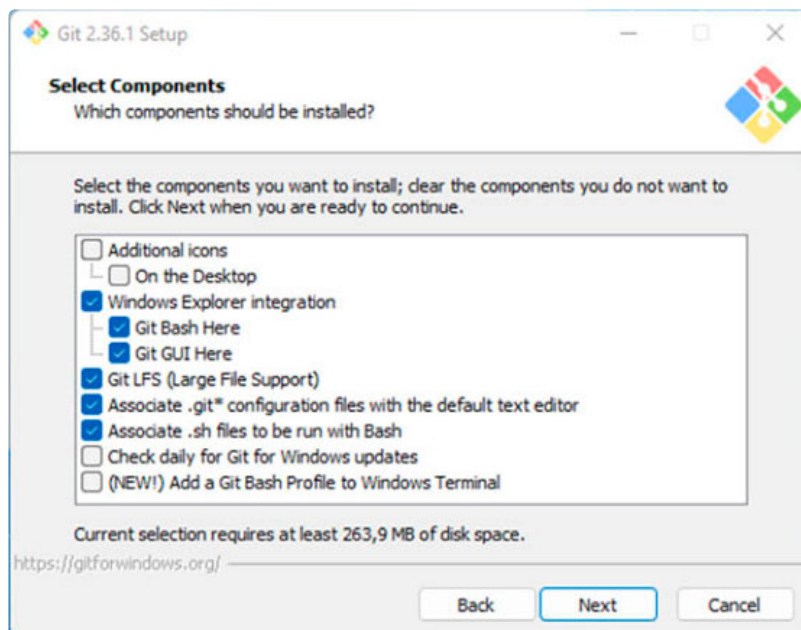
Tela de instalação do Git no Windows.

Na próxima imagem, você pode escolher o local de instalação do Git em seu disco rígido.



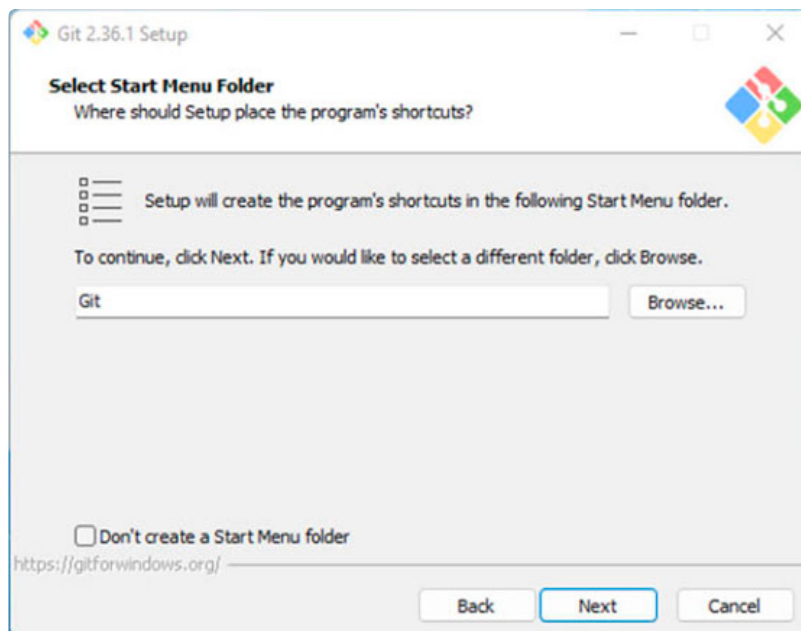
Definindo o local de instalação do Git.

Já na próxima tela, selecione os componentes para instalação do Git.



Escolhendo componentes para instalação do Git.

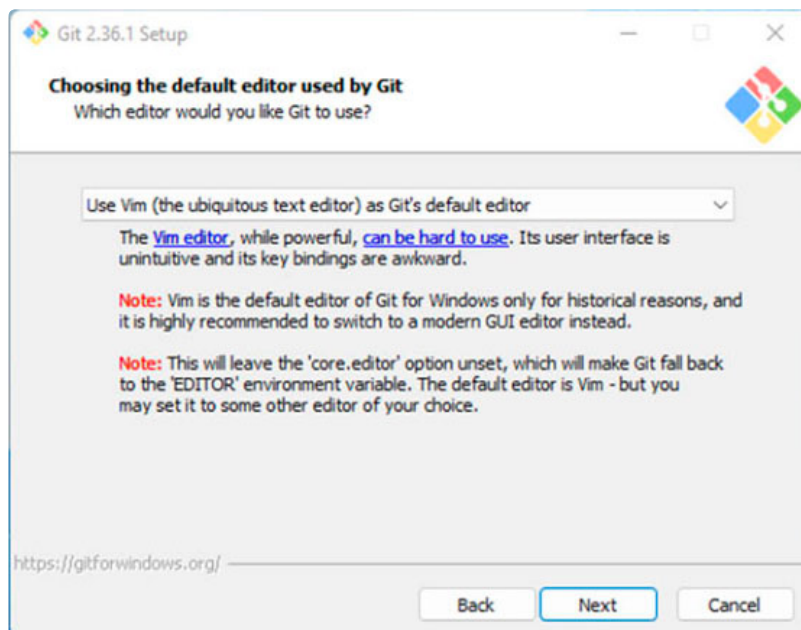
A seguir, escolha a pasta menu **Iniciar**, onde serão adicionados os atalhos dos programas do Git.



Selecionando a pasta menu Iniciar para adição dos atalhos do Git.

---

Na próxima etapa, selecione o editor padrão do Git.

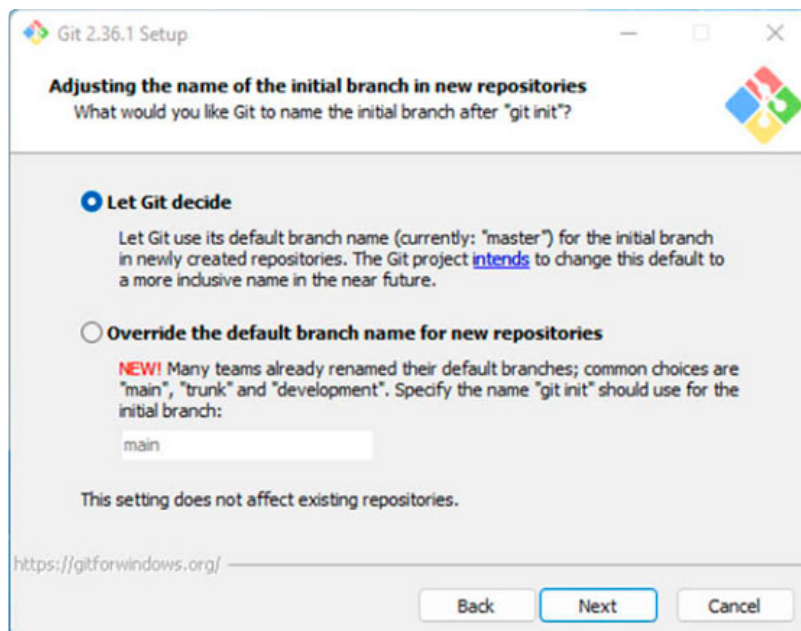


Escolhendo o editor padrão do Git.

---

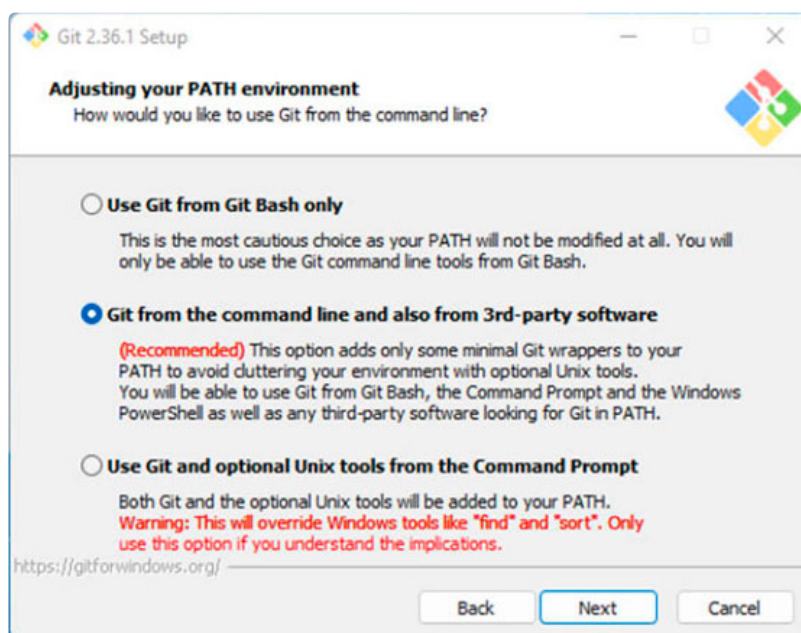
Na tela a seguir, escolha a opção para o nome inicial do repositório.





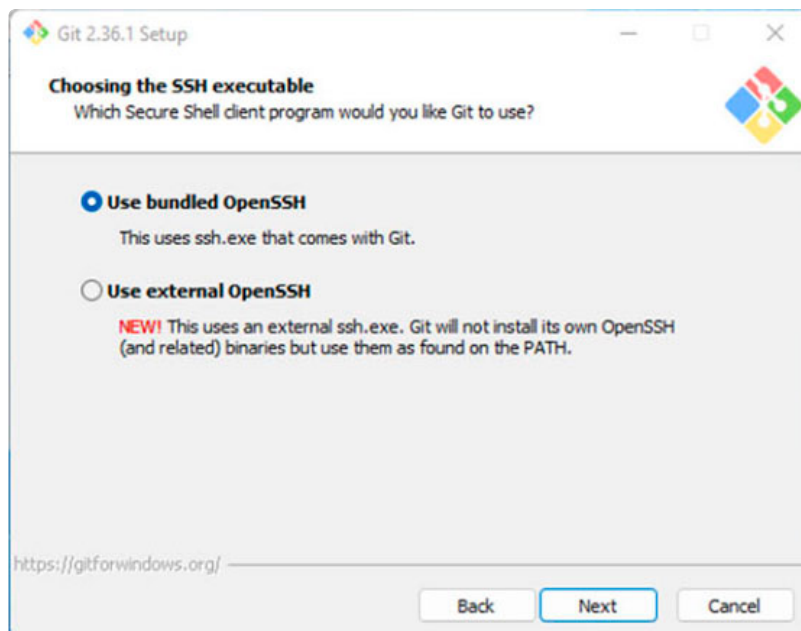
Nome padrão para criação de novo repositório no Git.

Você pode personalizar as variáveis de ambiente que serão carregadas no prompt de comando de linha.



Ajustando as variáveis de ambiente para o Git.

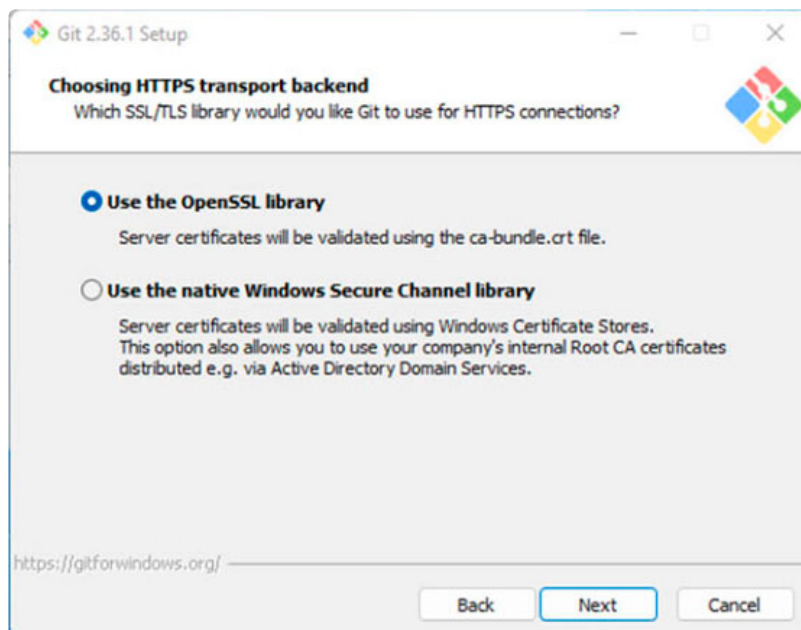
Na próxima etapa, escolha o programa cliente de acesso ao Git no Windows.



Selecionando o comando cliente para executar o Git.

---

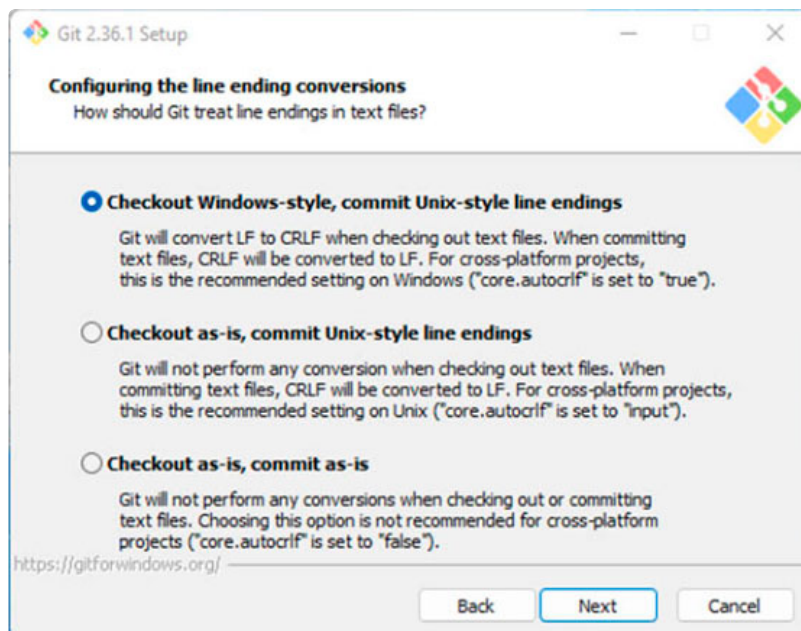
Novamente, escolha o programa cliente de acesso ao Git no Windows.



Selecionando o comando cliente para executar o Git.

---

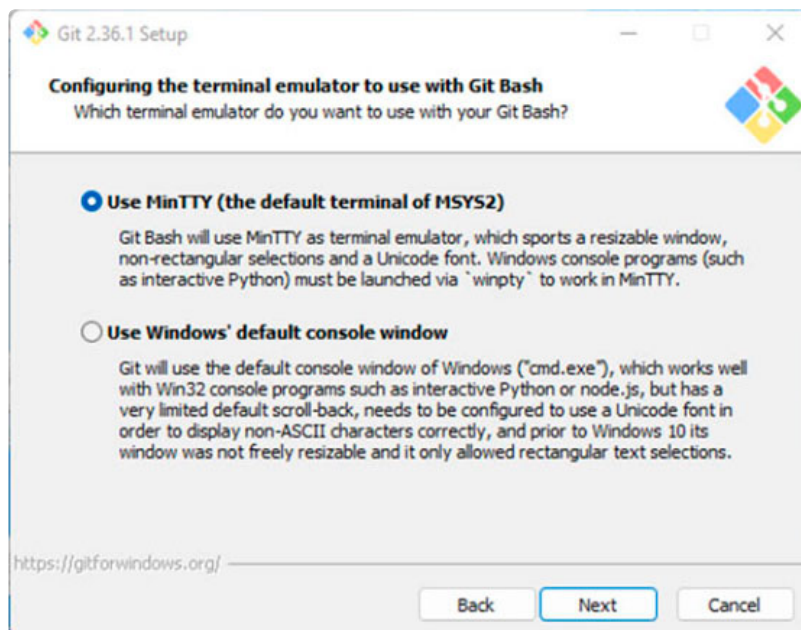
Agora, selecione o estilo de checkout dos arquivos no Git.



Selecionando o tipo de estilo de checkout para os arquivos no Git.

---

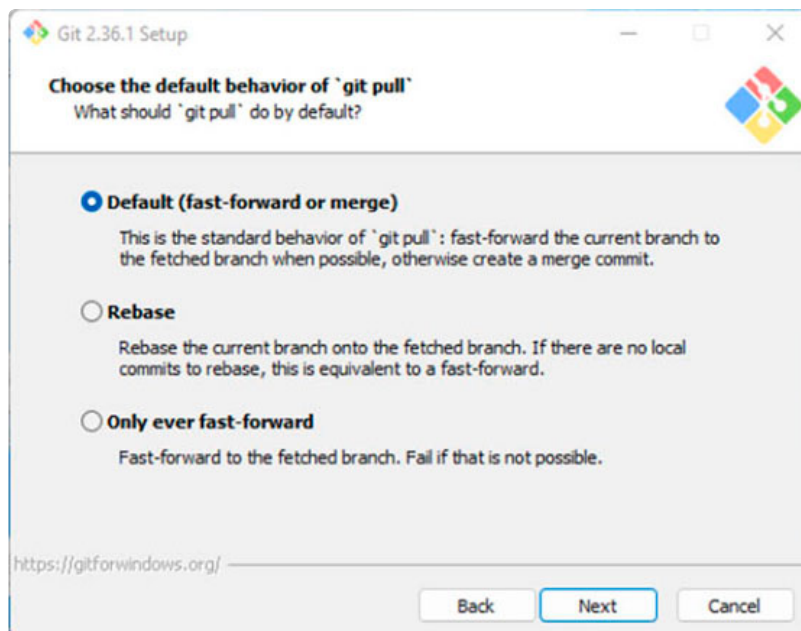
Escolha o tipo de terminal para ser usado no Git.



Selecionando o tipo de terminal para uso do Git.

---

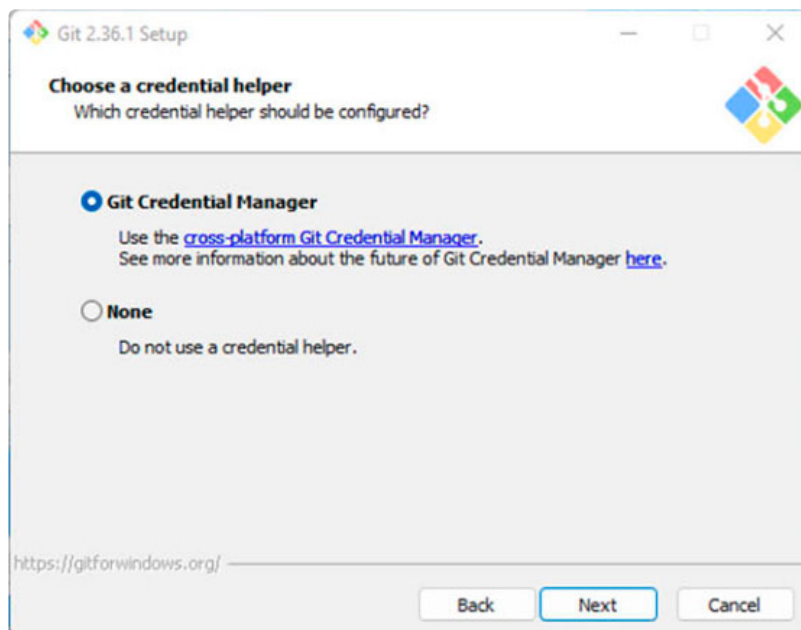
Escolha o comportamento padrão do git pull.



Escolhendo o tipo de comportamento padrão do git pull.

---

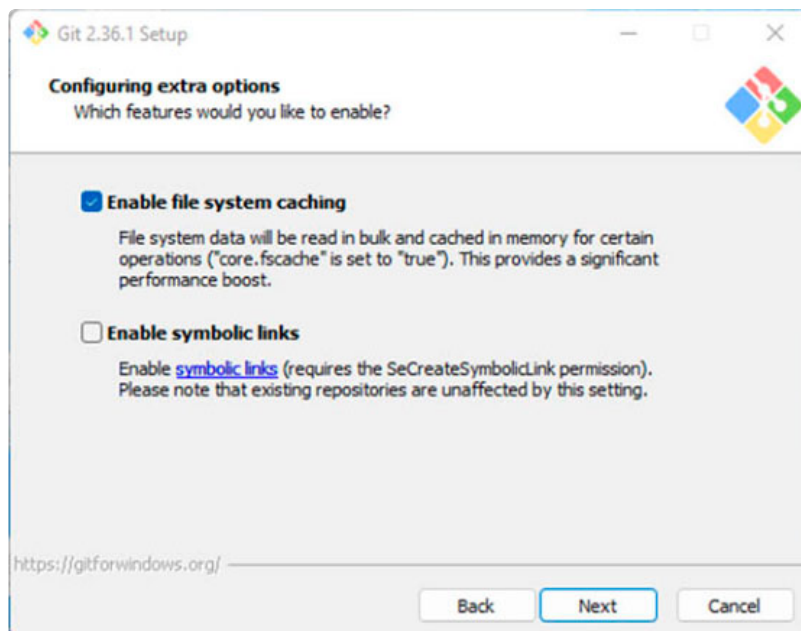
Escolha o gerenciador de credenciais para acesso ao Git.



Selecionando o gerenciador de credenciais para o Git.

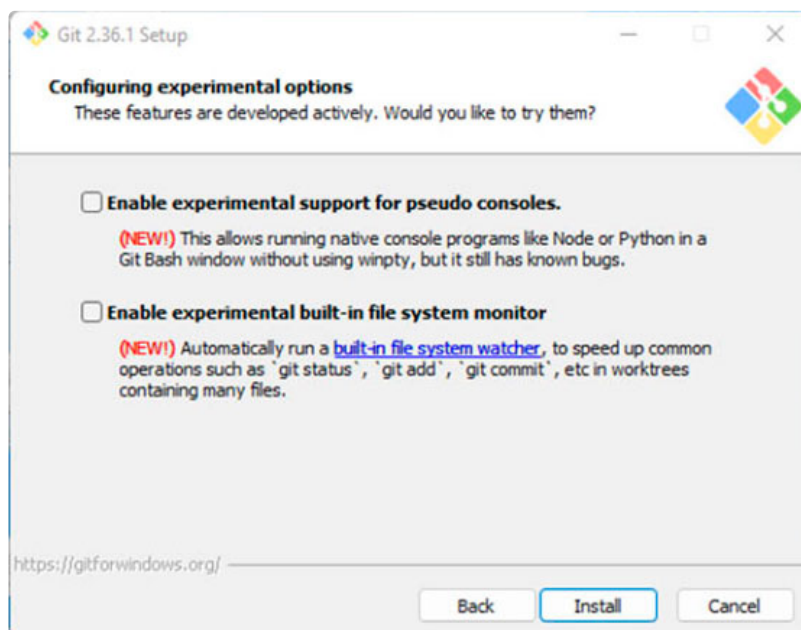
---

Habilite a opção extra de caching para melhor desempenho do Git.



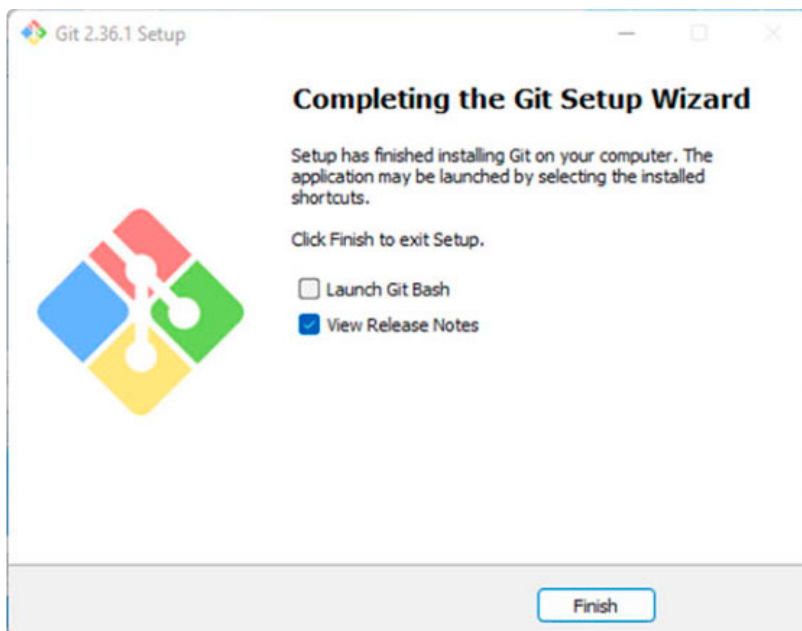
Adicionando a opção extra de caching.

As opções experimentais podem ser usadas para estudo do desenvolvedor e, caso seja oportuno, poderá ser adicionada durante a instalação.



Adicionando opções experimentais do Git.

A imagem a seguir apresenta que a instalação do Git foi concluída com sucesso no sistema operacional Windows.

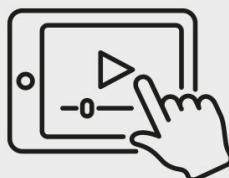


Tela de conclusão da instalação do Git.



## Veja agora a instalação do GitHub no ambiente Windows.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



### Instalando o Git no Linux

Há diversas distribuições Linux disponíveis atualmente no mercado de tecnologia. No entanto, sempre é recomendável para ambientes corporativos usar distribuições desse sistema operacional das quais você possa obter suporte e que tenham um modelo maduro do desenvolvimento desse projeto, como Ubuntu, Debian, CentOS, Red Hat.

Agora, vamos aprender as instruções necessárias para a instalação do Git no sistema operacional Ubuntu. Você pode usar o gerenciador de pacotes de distribuição para baixá-lo e o instalar; um comando `sudo apt-get install git` ou equivalente fornecerá o Git e todas as dependências necessárias em segundos, conforme mostrado a seguir.



```
azureuser@vm-estudo-ensiname-001:~$ sudo apt-get install git
Reading package lists... Done
Building dependency tree
Reading state information... Done
git is already the newest version (1:2.25.1-1ubuntu3.4).
git set to manually installed.
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
azureuser@vm-estudo-ensiname-001:~$
```

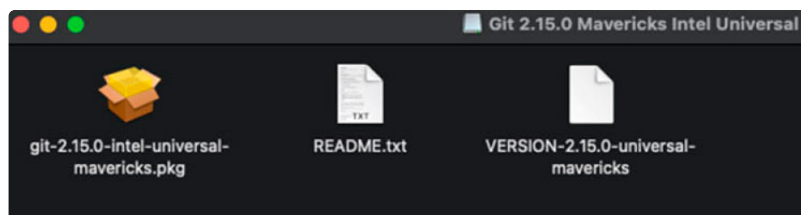
Download do Git.

---

## Instalando o Git no MacOS

O sistema operacional MacOS, desenvolvido pela empresa de tecnologia Apple, é baseado no projeto FreeBSD Unix e que vem pré-instalado em seus computadores desde 2002. Agora, vamos aprender como proceder a **instalação do Git de forma gráfica no MacOS**. No entanto, a instalação pode ser feita também de forma programática. Acesse o seguinte endereço eletrônico para baixar o arquivo para proceder com a instalação: <https://sourceforge.net/projects/git-osx-installer/>.

Após o download, execute o pacote da versão mais recente do Git que você baixou a fim de iniciar o assistente de instalação, conforme mostrado na imagem a seguir. É importante mencionar também que os computadores da Apple possuem processadores próprios chamados de M1 Apple, além da opção do processador Intel.



Download do Git.

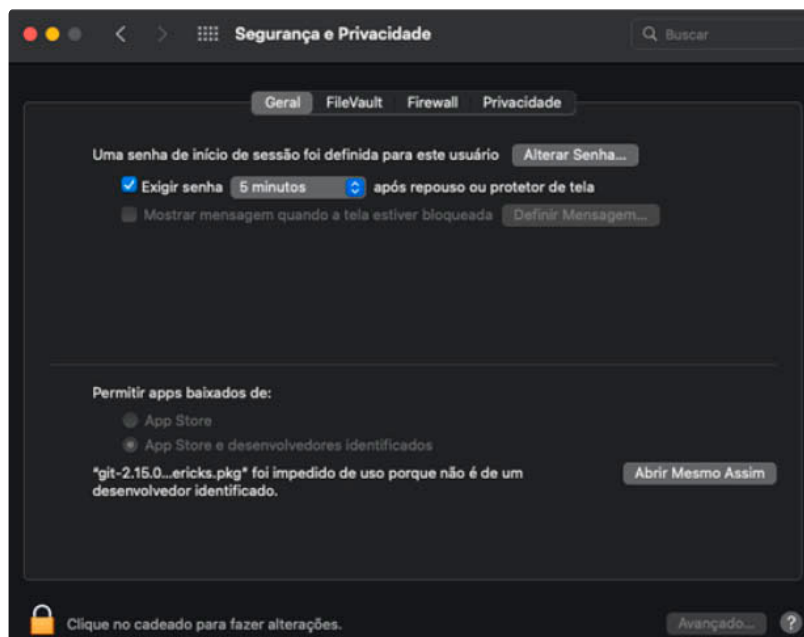
---

Caso seja apresentada uma mensagem, conforme mostrado na imagem abaixo, você precisará fazer uma mudança de configuração no sistema operacional do MacOS. Esse aviso é mostrado pelo MacOS em decorrência de um recurso que vem incorporado a ele, chamado Gatekeeper. Ele garante que o computador Mac apenas executará software que seja confiável, e, para Apple, o local mais seguro para obtenção de aplicativos consiste na loja de aplicativos, App Store.



Mensagem de aplicação não suportada nativamente pelo MacOS.

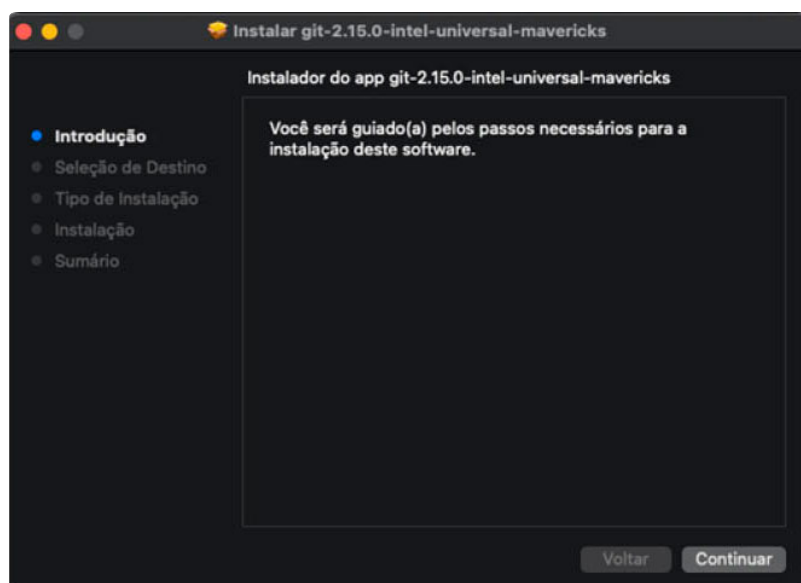
Para proceder com a instalação, no aplicativo de Preferências do Sistema, selecione Segurança e Privacidade e, em seguida, a Aba Geral, conforme mostrado abaixo. Clique na opção Abrir Mesmo Assim, para que o assistente de instalação do Git seja iniciado.



Configuração de execução do aplicativo.

Será apresentado o assistente de instalação do Git, de acordo com a imagem a seguir. Clique em Continuar para prosseguir.





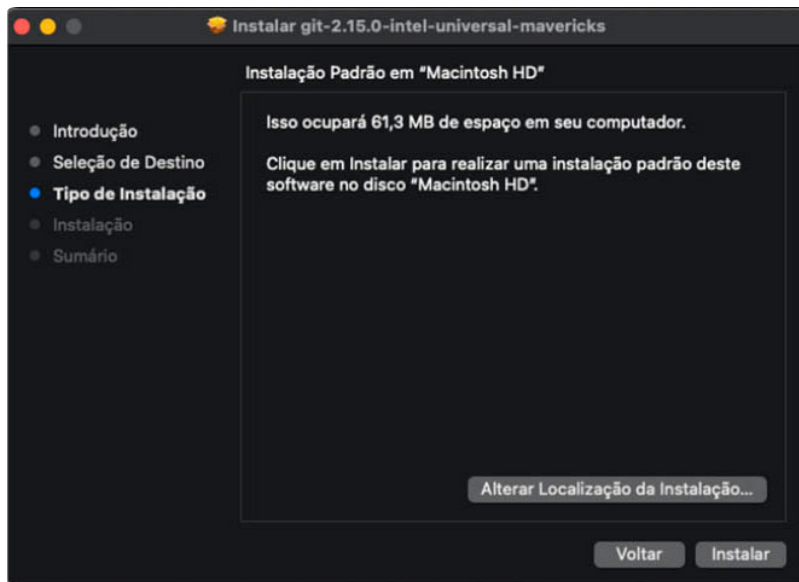
Tela inicial de instalação do Git.

Mude o local de instalação do programa Git, se você possui mais de um disco rígido em seu equipamento. Clique em Continuar para dar continuidade à instalação.



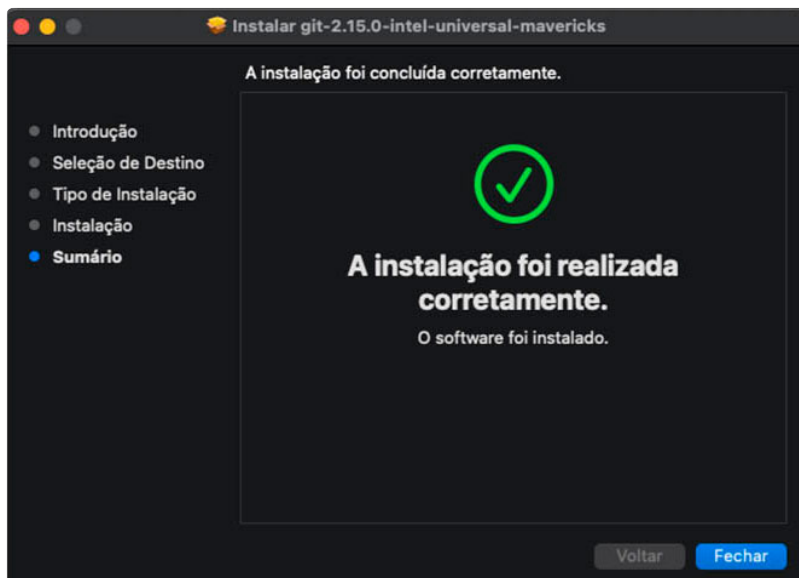
Selecione o local de instalação do Git.

Por fim, clique na opção Instalar para executar esse processo, de acordo com a imagem a seguir:



Selecionando o tipo de instalação.

Será apresentada a seguinte tela informando que a instalação foi executada com sucesso.



Tela de conclusão da instalação do Git.

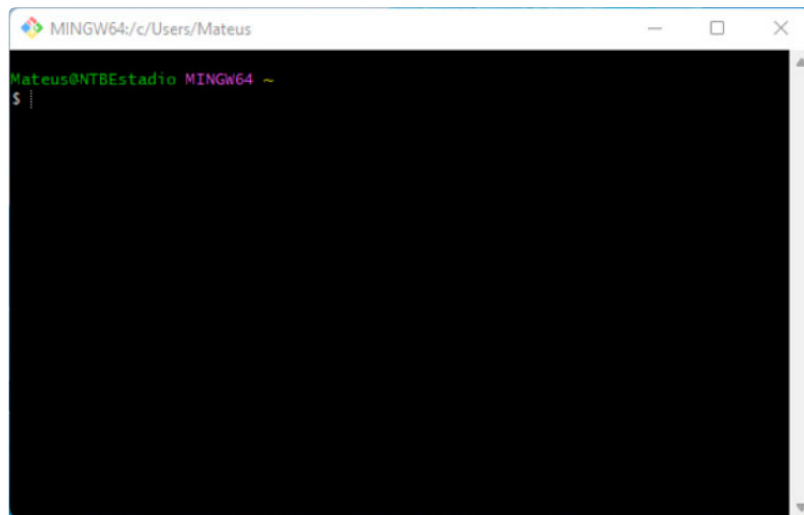
## Configuração e Teste do Git

A partir de agora, por conveniência, usaremos o Windows como nossa plataforma de referência. Nossas capturas de tela sempre se referirão a essa plataforma. De qualquer forma, todos os comandos principais do Git que usaremos funcionarão nas plataformas que mencionamos anteriormente.

É hora de testar nossa instalação. O Git está pronto para arrasar? Vamos descobrir!

Usando a integração do shell, clique com o botão direito do mouse em um local vazio na área de trabalho e escolha o novo item de menu Git

Bash. Ele aparecerá como um novo shell MinTTY, fornecendo a você um bash pronto para Git para Windows, de acordo com a imagem a seguir.



Shell MinTTY para uso no Git.

Este é um prompt típico do Bash, em que podemos ver o usuário, Mateus, e o host, NTBEstadio. Depois, há uma string MinGW64, que se refere à plataforma real que estamos usando, chamada Minimalist GNU for Windows.

Se o Git estiver instalado corretamente, digite simplesmente git sem especificar nada e será apresentada uma lista de comandos comuns. Caso contrário, precisará ser feita a reinstalação do Git.

```

MINGW64/c/Users/Mateus
$ git
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
          [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
          [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
          [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
          [--super-prefix=<path>] [--config-env=<name>=<envvar>]
          <command> [<args>]

These are common Git commands used in various situations:


start a working area (see also: git help tutorial)
  clone      Clone a repository into a new directory
  init       Create an empty Git repository or reinitialize an existing one


work on the current change (see also: git help everyday)
  add        Add file contents to the index
  mv         Move or rename a file, a directory, or a symlink
  restore    Restore working tree files
  rm         Remove files from the working tree and from the index


examine the history and state (see also: git help revisions)
  bisect     Use binary search to find the commit that introduced a bug
  diff       Show changes between commits, commit and working tree, etc
  grep       Print lines matching a pattern
  log        Show commit logs
  show       Show various types of objects
  status     Show the working tree status


grow, mark and tweak your common history
  branch     List, create, or delete branches
  commit     Record changes to the repository
  merge      Join two or more development histories together
  rebase     Reapply commits on top of another base tip
  reset      Reset current HEAD to the specified state
  switch     Switch branches
  tag        Create, list, delete or verify a tag object signed with GPG


collaborate (see also: git help workflows)
  fetch      Download objects and refs from another repository
  pull       Fetch from and integrate with another repository or a local branch
  push       Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
See 'git help git' for an overview of the system.

Mateus@NTBEstadio MINGW64 ~
$ |

```

Validando a instalação do Git.

Então, temos o Git funcionando! Você está animado? Vamos começar a entender como configurar o Git. Na próxima imagem, digite `git config`, para que sejam mostradas as opções de configurações que você poderá realizar.

```

MINGW64/c/Users/Mateus
Mateus@NTBEstadio MINGW64 ~
$ git config
usage: git config [<options>]

Config file location
  --global      use global config file
  --system      use system config file
  --local       use repository config file
  --worktree    use per-worktree config file
  -f, --file <file>  use given config file
  --blob <blob-id>  read config from given blob object

Action
  --get          get value: name [value-pattern]
  --get-all     get all values: key [value-pattern]
  --get-regexp   get values for regexp: name-regex [value-pattern]
  --get-urlmatch get value specific for the URL: section[.var] URL
  --replace-all replace all matching variables: name value [value-pattern]

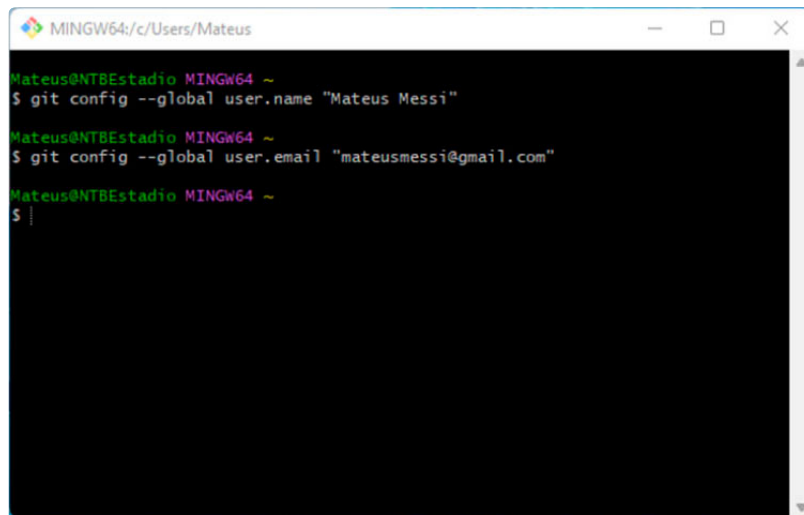
  --add          add a new variable: name value
  --unset        remove a variable: name [value-pattern]
  --unset-all    remove all matches: name [value-pattern]
  --rename-section rename section: old-name new-name

```

Validando a instalação do Git.

O Git precisa saber quem você é. Por isso é que no Git toda modificação que você faz em um repositório tem que ser assinada com o nome e e-mail do autor. Então, antes de fazer qualquer outra coisa, temos que

informar ao Git esse dado. Na próxima imagem, você poderá fazer essa configuração, utilizando o comando `git config --global`.

A screenshot of a Windows terminal window titled 'MINGW64: c:/Users/Mateus'. The prompt is 'Mateus@NTBEstadio MINGW64 ~'. The user enters the command '\$ git config --global user.name "Mateus Messi"'. The prompt changes to 'Mateus@NTBEstadio MINGW64 ~' and the user enters '\$ git config --global user.email "mateusmessi@gmail.com"'. The prompt changes again to 'Mateus@NTBEstadio MINGW64 ~' and the user enters '\$'.

```
Mateus@NTBEstadio MINGW64 ~  
$ git config --global user.name "Mateus Messi"  
Mateus@NTBEstadio MINGW64 ~  
$ git config --global user.email "mateusmessi@gmail.com"  
Mateus@NTBEstadio MINGW64 ~  
$
```

Validando a instalação do Git.

Neste módulo, você pôde aprender como proceder com a instalação do Git em diversos tipos de sistema operacional, bem como validar se a instalação transcorreu como esperado. No próximo módulo, conheceremos os principais comandos que você poderá usar no Git. Vamos lá?

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

### Questão 1

Sobre o sistema de controle de versão Git, marque a assertiva verdadeira.

**A**

Pode ser instalado apenas nos sistemas operacionais Windows e Linux.

**B**

A instalação do Git no sistema operacional MacOS pode ser feita usando unicamente a loja de aplicativo da Apple.

**C**

É necessário realizar durante o processo de instalação do Git no Linux ou MacOS a escolha do ambiente de shell para uso do Git.

**D**

Os comandos para uso do Git são independentes do sistema operacional escolhido para uso.

**E**

Torna-se fundamental que o programador opte pela instalação unicamente por linha de comando do Git no Windows, buscando uma maior segurança de uso.

**Parabéns! A alternativa D está correta.**

Podem ser usados os comandos do Git independentemente do sistema operacional escolhido, garantindo, dessa forma, maior flexibilidade pelo time de programadores da plataforma e sistema operacional de trabalho. O Git pode ser instalado nos sistemas operacionais Windows, Linux e MacOS, podendo ainda serem escolhidos dois métodos de instalação: gráfico e por linha de comando. Em ambos os casos, não há diferença no resultado final, seja no tocante à segurança ou desempenho do Git no referido sistema operacional.

## Questão 2

Analise as afirmativas:

I. O sistema de controle de versão Git surgiu a partir de outra ferramenta de versionamento de código-fonte chamada BitKeeper.

Porque

II. Ocorreu uma quebra na parceria, o que impulsionou Linus Torvalds a implementar um sistema de controle de código que pudesse dar sustentação ao desenvolvimento do sistema operacional Linux entre os programadores.

**A**

I é verdadeira, II é verdadeira e a II justifica a I.

**B**

I é verdadeira, II é verdadeira, porém a II não justifica a I.

**C**

I e II são falsas.

D I é falsa e II é verdadeira.

E I é verdadeira e II é falsa.

Parabéns! A alternativa A está correta.

Linus Torvalds teve um problema na ruptura da parceria com a empresa que detinha o sistema de controle de versão BitKeeper, que poderia impedir o desenvolvimento do kernel do Linux pelos programadores de forma colaborativa. Com isso, surgiu o projeto de código aberto Git para resolver esse impasse.



### 3 - Comandos básicos do Git

Ao final deste módulo, você será capaz de empregar os comandos básicos do Git.

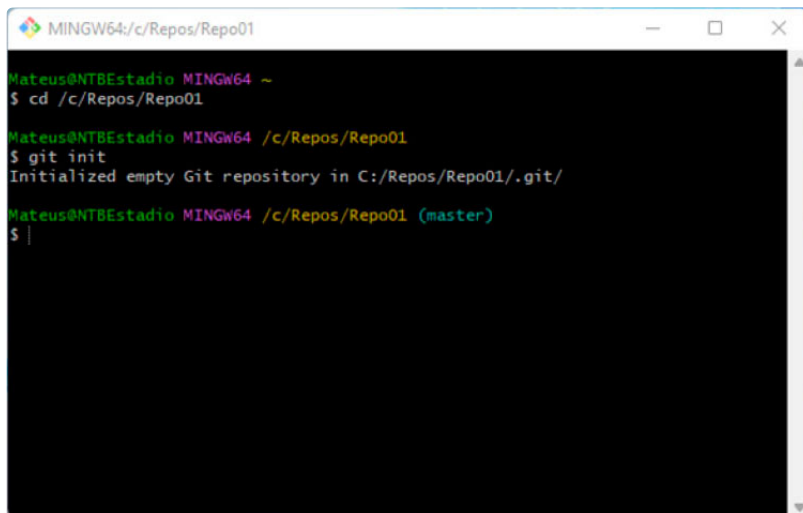
## Usando o Git

### Criando projeto no Git

Inicialmente, precisamos configurar um novo repositório. Um repositório é um contêiner para todo o projeto, portanto, cada arquivo ou subpasta dentro dele pertence a esse repositório. Fisicamente, um repositório nada mais é do que uma pasta que contém uma pasta especial chamada Git. Nessa pasta é que acontece a “mágica” na ferramenta.



Vamos começar criando o nosso primeiro repositório. Escolha uma pasta de sua preferência (por exemplo, C:\Repos\Repo01) e digite o comando git init, conforme apresentado na imagem a seguir.



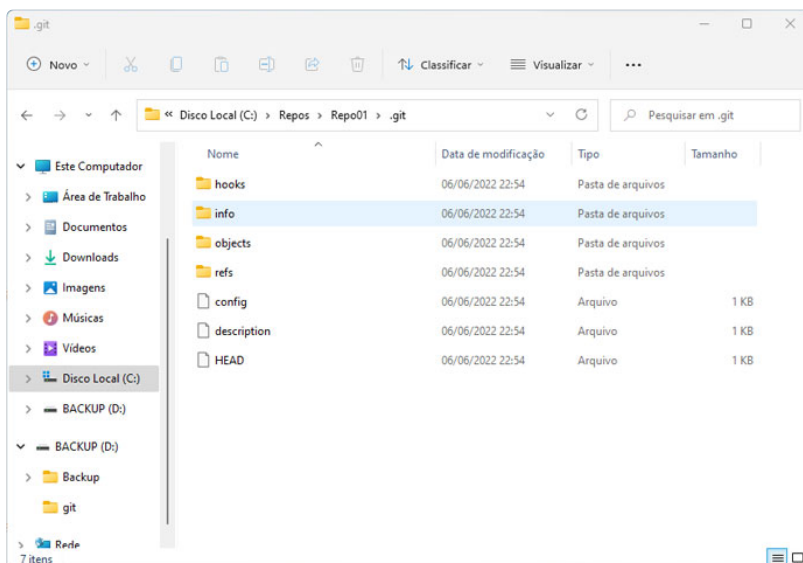
```
MINGW64/c:/Repos/Repo01
Mateus@NTBEstadio MINGW64 ~
$ cd /c:/Repos/Repo01

Mateus@NTBEstadio MINGW64 /c:/Repos/Repo01
$ git init
Initialized empty Git repository in C:/Repos/Repo01/.git/

Mateus@NTBEstadio MINGW64 /c:/Repos/Repo01 (master)
$
```

Criação de repositório no Git.

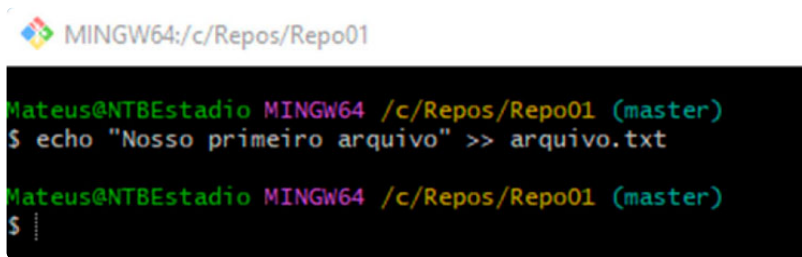
O que aconteceu dentro da pasta Repo01? O Git criou uma subpasta chamada .git. Essa subpasta geralmente fica oculta no Windows e, para vê-la, use o Windows Explorer e selecione a opção Mostrar itens ocultos. Após essa configuração, será possível identificar alguns outros arquivos e pastas, conforme mostrado na próxima imagem.



Estrutura do repositório Git após a criação.

Nesse momento, não é importante compreendermos o que está dentro dessa pasta. A única coisa que você precisa saber é que você não precisa modificá-la. Se você a excluir ou, ainda, modificar os arquivos manualmente, você pode ter problemas no repositório criado. Agora que temos um repositório, podemos começar a colocar arquivos dentro dele. O Git pode rastrear o histórico de qualquer tipo de arquivo, baseado em texto ou binário, pequeno ou grande, com a mesma eficiência.

Para ilustrar a criação de um arquivo no repositório Repo01, inicialmente vamos gerar o arquivo.txt contendo uma linha com o texto “Nosso primeiro arquivo”.



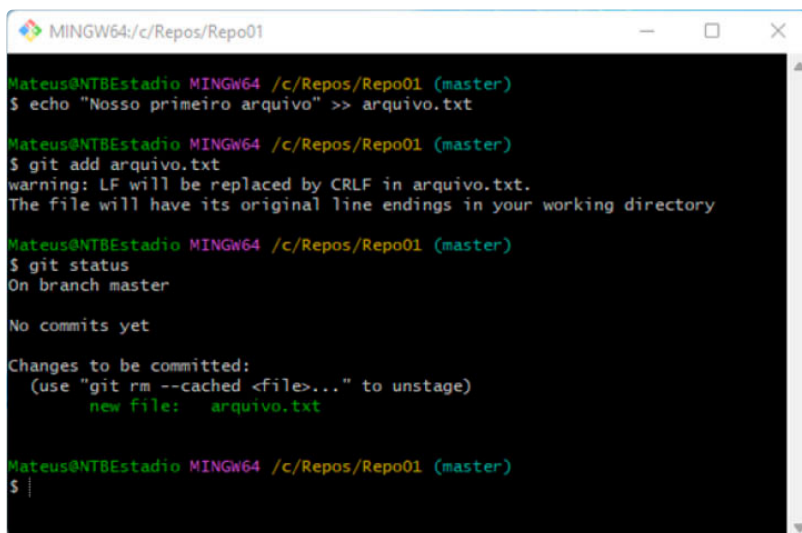
```
MINGW64:/c/Repos/Repo01

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ echo "Nosso primeiro arquivo" >> arquivo.txt

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$
```

Geração de arquivo no repositório.

E agora, Isso é tudo? Não! Temos que dizer explicitamente ao Git para colocar esse arquivo em seu repositório. Para fazer isso, execute a instrução: `git add arquivo.txt`. O comando **git add** diz ao Git que queremos que ele cuide desse arquivo e verifique se há modificações futuras, além do que, ao executar a instrução **git status**, é possível verificar que o arquivo precisa ser confirmado no **branch master** do repositório, de acordo com o que é apresentado na imagem a seguir.



```
MINGW64:/c/Repos/Repo01

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ echo "Nosso primeiro arquivo" >> arquivo.txt

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ git add arquivo.txt
warning: LF will be replaced by CRLF in arquivo.txt.
The file will have its original line endings in your working directory

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ git status
On branch master

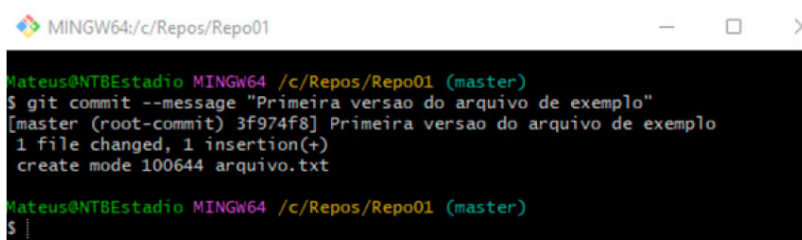
No commits yet

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

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$
```

Adição de arquivo no repositório Git.

Nesse ponto, o Git conhece o arquivo.txt. No entanto, temos que executar a instrução **git commit** para que o arquivo possa ser usado por outro desenvolvedor quando ele proceder com a cópia do repositório. Para isso, é necessário colocar uma mensagem no comando **git commit** sinalizando a motivação do commit desse arquivo para o repositório, conforme mostrado na imagem a seguir:



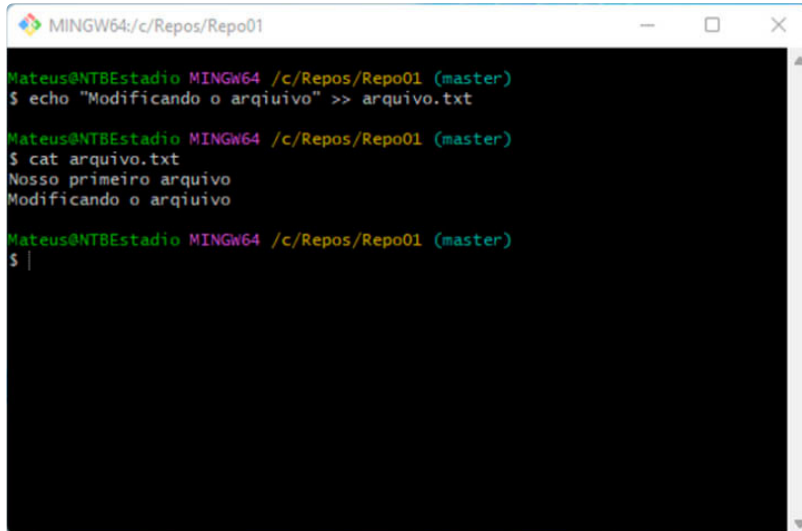
```
MINGW64:/c/Repos/Repo01

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ git commit --message "Primeira versao do arquivo de exemplo"
[master (root-commit) 3f974f8] Primeira versao do arquivo de exemplo
1 file changed, 1 insertion(+)
 create mode 100644 arquivo.txt

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$
```

Commit de arquivo no Git.

Agora, podemos tentar fazer algumas modificações no arquivo e ver como o Git trabalha as alterações. Da mesma forma que geramos o arquivo, utilize o comando **echo** para adicionar uma segunda linha no arquivo.txt. Você pode ainda digitar o comando **cat** arquivo.txt para visualizar o conteúdo inteiro do arquivo.txt.



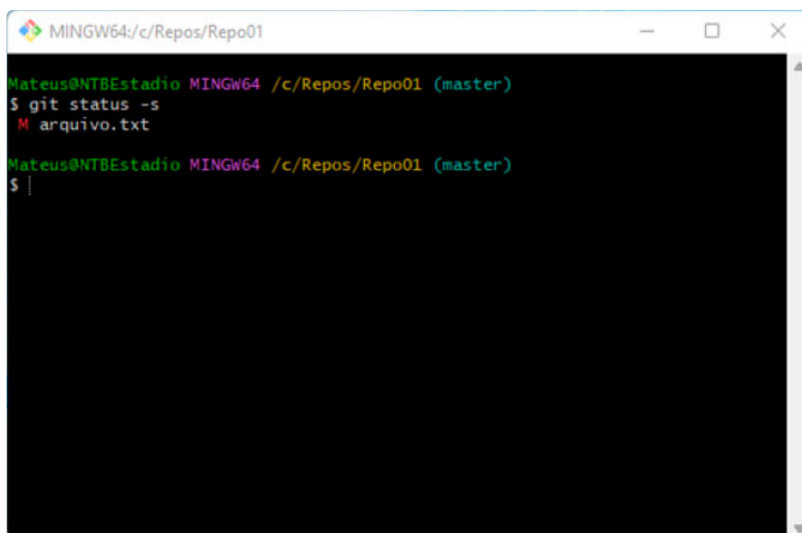
```
MINGW64/c/Repos/Repo01
Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ echo "Modificando o arquivo" >> arquivo.txt

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ cat arquivo.txt
Nosso primeiro arquivo
Modificando o arquivo

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ |
```

Modificando o arquivo no repositório Git.

Na sequência, se usarmos o comando: **git status -s**, será possível identificar qual(is) arquivo(s) foram modificados no repositório Git. Como você pode ver a seguir, o status do arquivo que foi modificado é sinalizado com a letra **M** na cor vermelha, indicando que alterações foram feitas e que precisa ser realizado o **git commit** para persistência das mudanças no repositório, a fim de que, caso algum outro desenvolvedor realize uma cópia atualizada do repositório, essa modificação esteja presente.



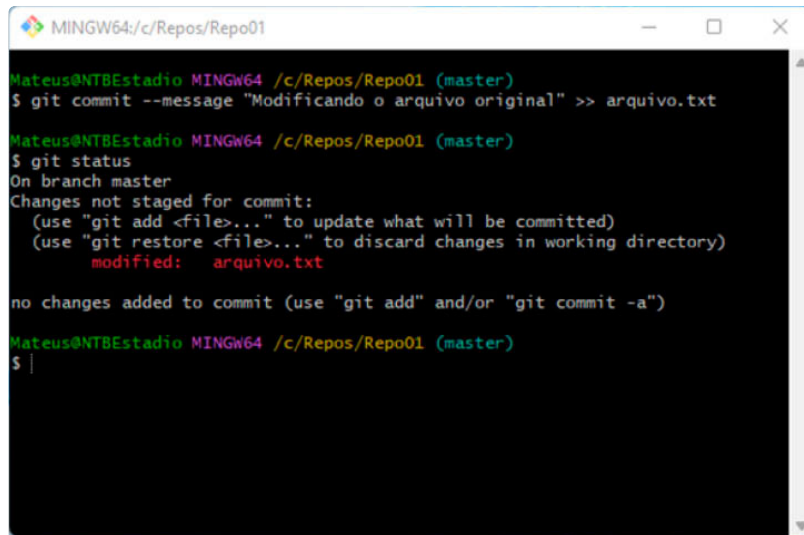
```
MINGW64/c/Repos/Repo01
Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ git status -s
M arquivo.txt

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ |
```

Verificando o status dos arquivos no repositório.

Agora precisamos executar o comando: `git commit --message "Modificando o arquivo original" >> arquivo.txt` para realizar o commit da mudança do arquivo, informando por meio do parâmetro `--message` uma anotação do que representa essa alteração, com o intuito de compreensão por parte do desenvolvedor desejar entender a motivação da mudança no código-fonte, conforme mostrado na imagem.

Em seguida, execute o comando `git status`, para verificar se há alguma pendência de arquivos que necessitam ser comitados.

A terminal window titled 'MINGW64:/c/Repos/Repo01' shows the following commands and output:

```
Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ git commit --message "Modificando o arquivo original" >> arquivo.txt

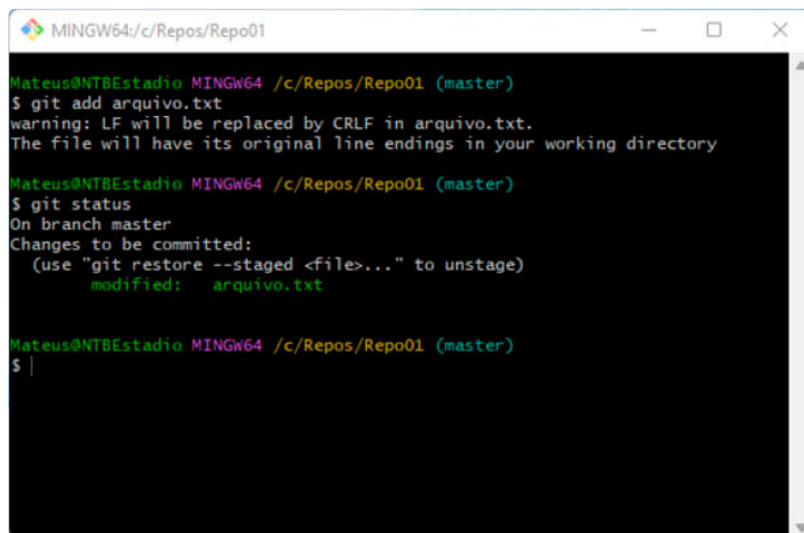
Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   arquivo.txt

no changes added to commit (use "git add" and/or "git commit -a")

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ |
```

Modificando o arquivo no repositório Git.

Na sequência, vamos adicionar o arquivo ao repositório com o intuito de que ele possa ser atualizado usando o comando `git add arquivo.txt`, vide imagem a seguir.

A terminal window titled 'MINGW64:/c/Repos/Repo01' shows the following commands and output:

```
Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ git add arquivo.txt
warning: LF will be replaced by CRLF in arquivo.txt.
The file will have its original line endings in your working directory

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   arquivo.txt

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ |
```

Modificando o arquivo no repositório Git.

Ao digitar o comando **git status**, nesse momento, será aberto o editor de texto Vim, solicitando que você digite a mensagem de commit do arquivo.

[illegible]

Modificando o arquivo no repositório Git.

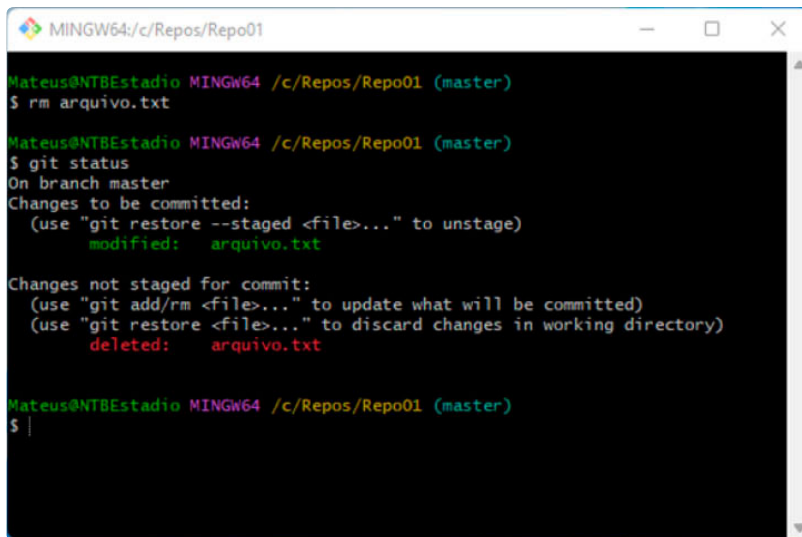
O Vim (Vi IMproved) é um antigo e poderoso editor de texto, usado até hoje por milhões de pessoas. Você pode configurar o Git para usar seu editor preferido, mas, se você não fizer isso, é com ele que você terá que trabalhar. Depois de digitar sua mensagem de confirmação, você pode pressionar Esc para sair do modo de edição. Em seguida, pode digitar o comando :w para gravar as alterações e o comando :q para sair. Você também pode digitar o comando em pares como :wq.

**Atenção!**

Quando você sair do Vim, o Git executará automaticamente o commit do arquivo para o repositório.

Se fosse necessário excluir um arquivo, como deveria ser feito?

Pois bem, a remoção de um arquivo do repositório de um projeto no Git é muito tranquila. Para isso, você necessitará utilizar o comando **rm** que, ao especificar o arquivo, irá excluí-lo do diretório de trabalho e, dessa forma, não será mais possível acessá-lo. A imagem a seguir mostra a instrução para a execução da deleção do arquivo, bem como a confirmação pelo comando `git status`.



```
MINGW64/c/Repos/Repo01
Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ rm arquivo.txt

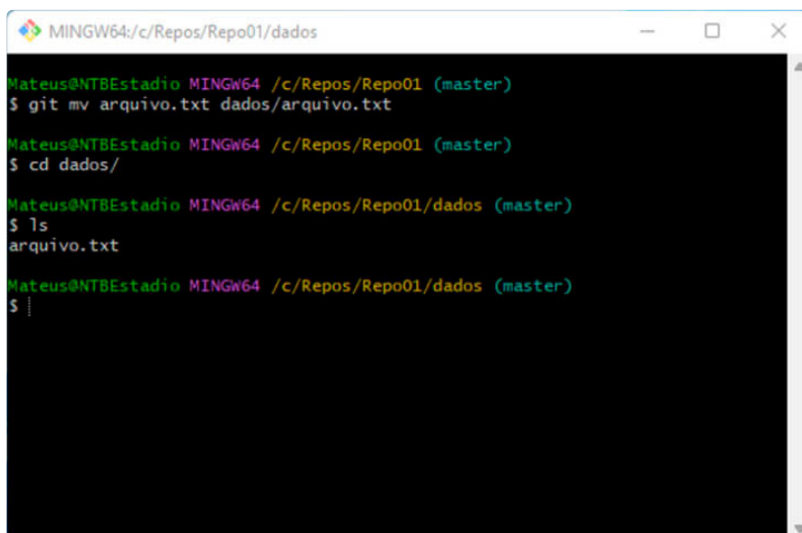
Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   arquivo.txt

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    arquivo.txt

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ |
```

Modificando o arquivo no repositório Git.

Uma curiosidade: ao contrário de muitos outros sistemas de versionamento de código-fonte, o Git não rastreia explicitamente a movimentação de arquivos. Se você renomear um arquivo, nenhum metadado relacionado à alteração desse arquivo será armazenado no Git. A princípio, pode parecer confuso que o Git tenha apenas um comando para renomear um arquivo ou até mesmo para movê-lo. Na imagem seguinte, ao executar o comando `git mv arquivo.txt dados/arquivo.txt`, o arquivo será movido para o diretório dados.



```
MINGW64/c/Repos/Repo01/dados
Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ git mv arquivo.txt dados/arquivo.txt

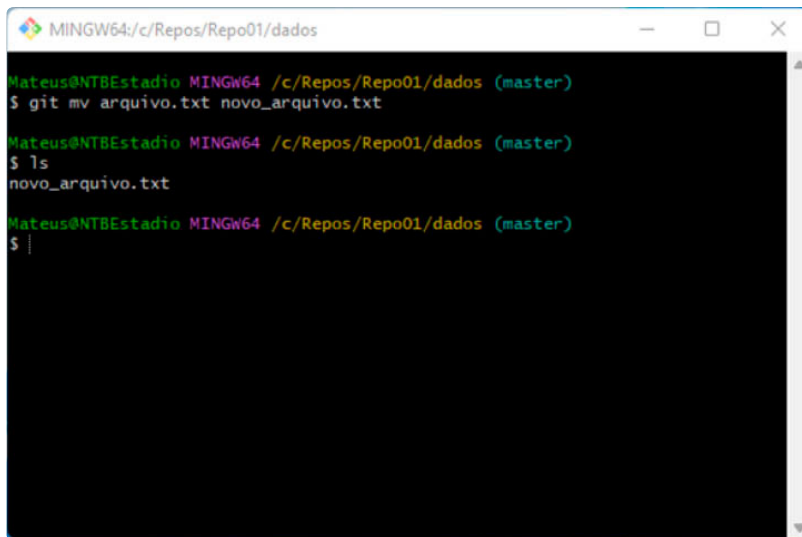
Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ cd dados/

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01/dados (master)
$ ls
arquivo.txt

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01/dados (master)
$ |
```

Movendo arquivo para outro diretório.

Vamos usar o comando `mv` em sua outra possibilidade, portanto, renomeando o arquivo. Logo, você pode verificar que, executando o comando `git mv arquivo.txt novo_arquivo.txt`, ele passará de `arquivo.txt` para `novo_arquivo.txt`.



```
MINGW64/c/Repos/Repo01/dados
Mateus@NTBEstadio MINGW64 /c/Repos/Repo01/dados (master)
$ git mv arquivo.txt novo_arquivo.txt

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01/dados (master)
$ ls
novo_arquivo.txt

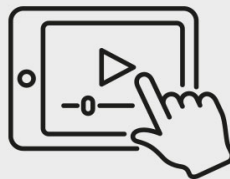
Mateus@NTBEstadio MINGW64 /c/Repos/Repo01/dados (master)
$
```

Renomeando arquivo no repositório Git.



## Veja agora o uso do GitHub

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



## Clonando um repositório Git existente

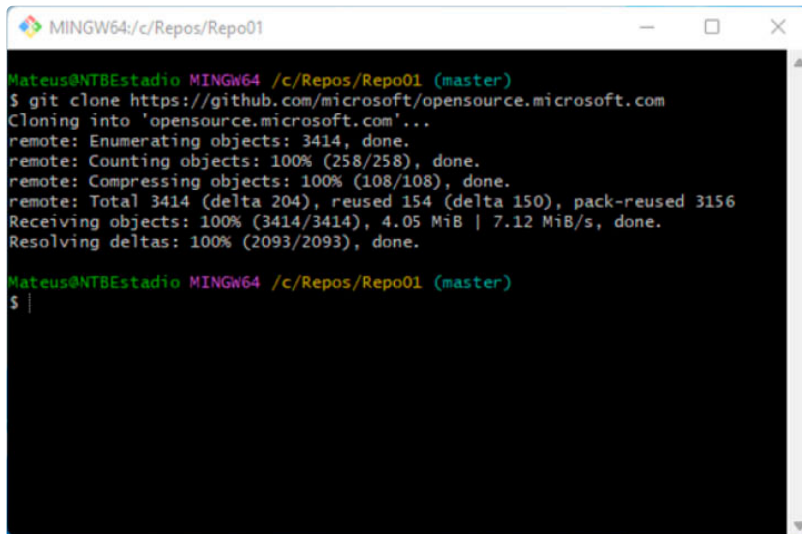
Se você deseja obter uma cópia de um repositório Git existente – por exemplo, um projeto para o qual gostaria de contribuir –, o comando necessário é `git clone`. Se você estiver familiarizado com outros sistemas de controle de versionamento, como o Subversion, notará que o comando a ser usado é “clone”, e não “checkout”. Essa é uma distinção importante, porque, ao invés de obter apenas uma cópia de trabalho, o Git recebe uma cópia completa de quase todos os dados que o servidor possui. Cada versão de cada arquivo é baixada por padrão quando você executa o `git clone`.

Na verdade, se o disco do seu servidor for corrompido, é possível utilizar qualquer um dos clones em qualquer cliente para definir o servidor de contingência e dessa maneira retornar ao estado em que estava quando foi clonado. Você clona um repositório com o comando `git clone [url]`.



Por exemplo, se você quiser clonar o projeto Open Source da Microsoft disponível no Git, digite:

<https://github.com/microsoft/opensource.microsoft.com>, conforme mostrado a seguir:

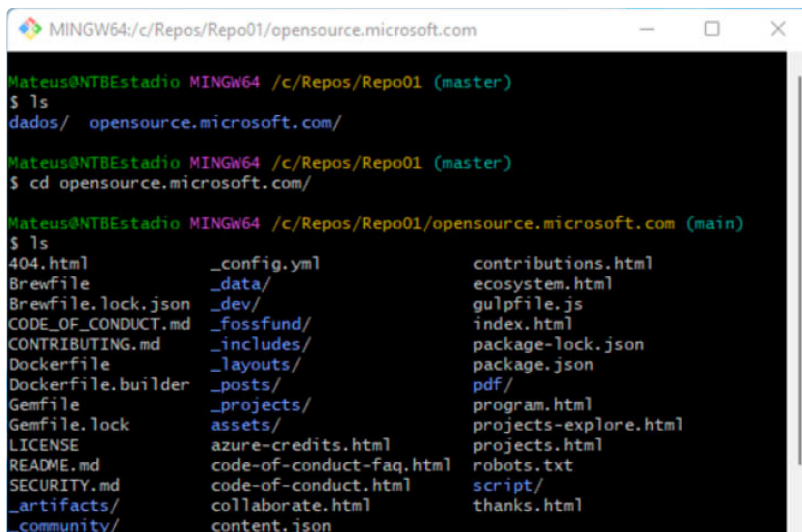


```
MINGW64/c/Repos/Repo01
Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ git clone https://github.com/microsoft/opensource.microsoft.com
Cloning into 'opensource.microsoft.com'...
remote: Enumerating objects: 3414, done.
remote: Counting objects: 100% (258/258), done.
remote: Compressing objects: 100% (108/108), done.
remote: Total 3414 (delta 204), reused 154 (delta 150), pack-reused 3156
Receiving objects: 100% (3414/3414), 4.05 MiB | 7.12 MiB/s, done.
Resolving deltas: 100% (2093/2093), done.

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$
```

Clonando projeto no Git.

Será criado um diretório chamado “opensource.microsoft.com”, onde é inicializado um diretório .git dentro dele, o qual baixará todos os dados para esse repositório e verificará se há uma cópia de trabalho da versão mais recente disponível. Você pode acessar o diretório `opensource.microsoft.com` e verificar que os arquivos do projeto estão presentes e prontos para serem trabalhados.



```
MINGW64/c/Repos/Repo01/opensource.microsoft.com
Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ ls
dados/  opensource.microsoft.com/

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01 (master)
$ cd opensource.microsoft.com/

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01/opensource.microsoft.com (main)
$ ls
404.html      _config.yml      contributions.html
Brewfile      _data/           ecosystem.html
Brewfile.lock.json  _dev/           gulpfile.js
CODE_OF_CONDUCT.md  _fossfund/      index.html
CONTRIBUTING.md   _includes/      package-lock.json
Dockerfile       _layouts/       package.json
Dockerfile.builder  _posts/        pdf/
Gemfile         _projects/      program.html
Gemfile.lock     assets/         projects-explore.html
LICENSE         azure-credits.html  projects.html
README.md        code-of-conduct-faq.html  robots.txt
SECURITY.md      code-of-conduct.html  script/
_artifacts/     collaborate.html    thanks.html
_community/     content.json
```

Estrutura e arquivos clonados.

Bem, com isso, finalizamos os principais comandos que você utilizará inicialmente em um projeto usando o Git. No próximo módulo, aprenderemos a criar ramificações, a tratar de conflitos de atualizações e muito mais.



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

### Questão 1

Analise as assertivas.

- I. Para iniciar um projeto no sistema de controle de versão Git, é necessário informar o diretório que será usado e digitar a

instrução `git init` para que sejam criadas as subpastas e os arquivos necessários para o versionamento do projeto.

Porque

II. O Git clonará a estrutura existente de outro projeto sem incorporar os arquivos do projeto gerador.

A

I é verdadeira, II é verdadeira e a II justifica a I.

B

I é verdadeira, II é verdadeira, porém a II não justifica a I.

C

I e II são falsas.

D

I é falsa e II é verdadeira.

E

I é verdadeira e II é falsa.

**Parabéns! A alternativa E está correta.**

O sistema de versionamento de código-fonte Git possibilita que seja criado um projeto novo sem obrigatoriamente clonar a estrutura existente de um projeto.

## Questão 2

Analise as afirmativas:

I. Adição de arquivos no sistema de controle de código-fonte Git pode ser feita digitando, por exemplo, `Git add arquivo.txt`.

Porque

II. Na modificação dos arquivos no repositório criado para um projeto Git, deve ser usada unicamente a instrução: `echo "Adicionando outra linha ao arquivo" >> arquivo.txt`.

**A** I é verdadeira, II é verdadeira e a II justifica a I.

**B** I é verdadeira, II é verdadeira, porém a II não justifica a I.

**C** I e II são falsas.

**D** I é verdadeira e II é falsa.

**E** I é falsa e II é verdadeira.

Parabéns! A alternativa D está correta.

Podem ser adicionados quantos arquivos forem necessários a um repositório Git, bastando para isso utilizar a instrução: `Git add`. No entanto, quando procedemos com a modificação dos arquivos, além da instrução `echo`, é necessário comitar o código-fonte e adicionar uma mensagem comentando a mudança realizada.



#### 4 - Operações de Merge, Diff, Branching e Merging.

Ao final deste módulo, você será capaz de aplicar o Git para controle de versões de software.

# Manipulação de arquivos no Git

Uma vez que foi criado ou clonado um repositório Git, operações como adição, modificação e exclusão de arquivos são tarefas comuns no desenvolvimento de códigos-fonte pelos desenvolvedores. Nesse contexto, torna-se interessante não proceder as operações com os arquivos diretamente no repositório principal (main ou master), e sim em uma ramificação que será sua própria estação de trabalho e na qual, oportunamente, você comitará os arquivos do projeto para o ramo principal e o sistema de controle de versionamento avaliará se acontecerá conflito ou não para ser resolvido pelas versões submetidas dos arquivos por outros desenvolvedores.

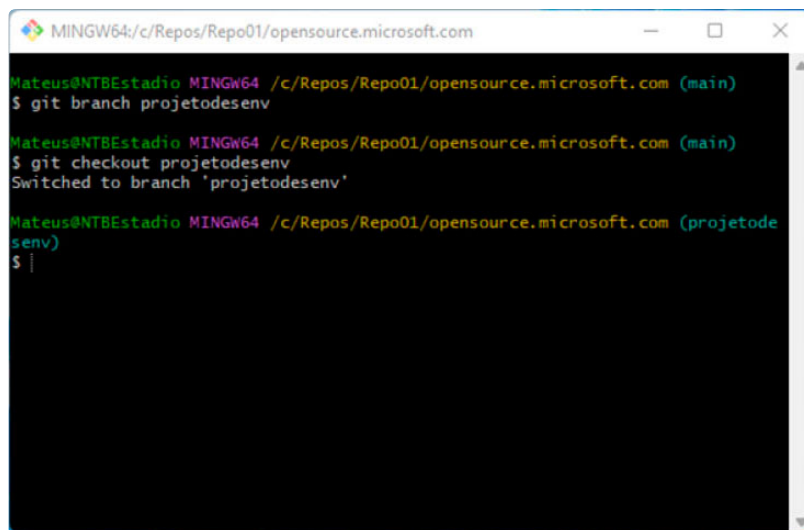
Pois bem, esse é o assunto deste módulo. Vamos lá?

## Criando novo branch

Anteriormente, você teve uma visão geral sobre comandos básicos do Git e como usá-los. Agora, vamos ver o que acontece quando você pede ao Git para criar um novo branch. Para criar um novo branch, tudo que você precisa fazer é chamar o **git branch** seguido do nome do branch que você gostaria que fosse chamado. Mas o que seria mesmo um branch?

Imagine uma árvore grande e frondosa com muitos galhos. Pois bem, é nessa analogia que o branch no Git se sustenta. Preliminarmente, criamos um repositório no módulo passado chamado de Repo01 que possui sua própria estrutura de arquivos. Quando criamos um novo branch, portanto, uma ramificação do branch principal (main), estamos originando uma cópia do repositório correspondente nesse material ao Repo01.

Na imagem a seguir, podemos criar um novo branch digitando: `git branch projetodesenv`. E na sequência, para alterar do branch que estamos para o que acabamos de criar, é preciso utilizar o comando: `git checkout projetodesenv`.



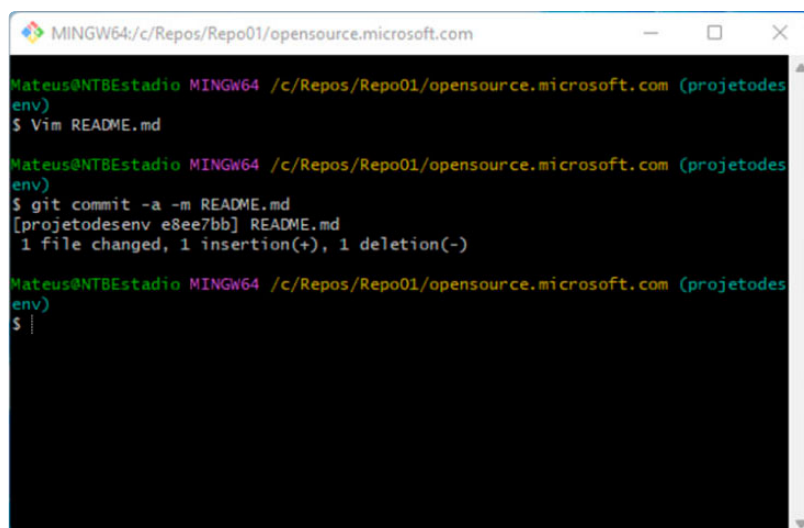
```
MINGW64:/c/Repos/Repo01/opensource.microsoft.com
Mateus@NTBEstadio MINGW64 /c/Repos/Repo01/opensource.microsoft.com (main)
$ git branch projetodesenv

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01/opensource.microsoft.com (main)
$ git checkout projetodesenv
Switched to branch 'projetodesenv'

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01/opensource.microsoft.com (projetodesenv)
$ |
```

Criando uma ramificação.

Na ramificação projetodesenv, vamos alterar um arquivo usando o editor de textos Vim. Digite: Vim README.md, modifique o arquivo e em seguida salve. Agora, execute a instrução de confirmação no repositório dessa alteração feita do arquivo com o comando: git commit -a -m README.md.



```
MINGW64:/c/Repos/Repo01/opensource.microsoft.com
Mateus@NTBEstadio MINGW64 /c/Repos/Repo01/opensource.microsoft.com (projetodesenv)
$ Vim README.md

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01/opensource.microsoft.com (projetodesenv)
$ git commit -a -m README.md
[projetodesenv e8ee7bb] README.md
1 file changed, 1 insertion(+), 1 deletion(-)

Mateus@NTBEstadio MINGW64 /c/Repos/Repo01/opensource.microsoft.com (projetodesenv)
$ |
```

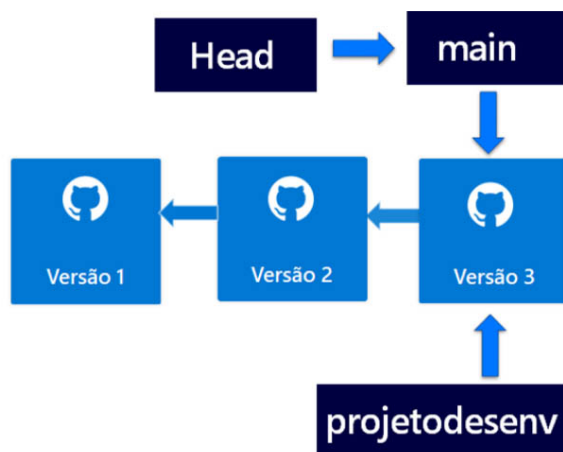
Modificando arquivo no branch projetodesenv.

Usando o comando git checkout main, você muda do branch projetodesenv para o principal (main), conforme mostrado na imagem a seguir. Observe ainda que é mostrada uma mensagem de que a ramificação projetodesenv está atualizada com o branch main.

```
MINGW64:/c/Repos/Repo01/opensource.microsoft.com
Mateus@NTBEstadio MINGW64 /c/Repos/Repo01/opensource.microsoft.com (projeto-desenv)
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
Mateus@NTBEstadio MINGW64 /c/Repos/Repo01/opensource.microsoft.com (main)
$
```

Mudando para o branch main.

Talvez você esteja se perguntando: como o Git sabe em qual branch você está atualmente trabalhando, não é mesmo? Pois bem, ele mantém um ponteiro especial chamado HEAD que informa que branch você está usando. A imagem abaixo ilustra esse mecanismo de referência no qual o HEAD direciona para a branch main, logo, para o ramo principal.



Ponteiro HEAD.

**Importante:** quando você alterna entre ramificações no Git, os arquivos em seu diretório de trabalho serão modificados. Se você mudar para uma ramificação mais antiga, seu diretório de trabalho será revertido para se parecer com a última vez que você realizou os commits nesse ramo.

Com certeza, essa dica lhe ajudará a evitar situações não desejadas. Mas, afinal, quais situações são recomendadas para criação de um novo branch? Basicamente, são três:

01

Quando houver a necessidade de reescrita de um algum código-fonte e você achar mais prudente trabalhar no desenvolvimento

desse código de maneira mais isolada sem impactar o restante da aplicação.

02

Quando houver uma perspectiva de elaboração de novos recursos e funcionalidades para sua aplicação e você desejar trabalhar nesse código de forma separada, evitando quaisquer transtornos com a versão utilizada pela aplicação.

03

Quando houver correção de problemas identificados em código-fonte.

**Observação:** você pode renomear o branch principal do seu repositório para o nome que preferir. Para isso, digite o comando: `git branch -m main meubranhprincipal`.

## Operação Merge

Para explicarmos a operação **Merge**, abordaremos uma situação-problema do mundo real que contém um exemplo simples de ramificação e fusão com um fluxo de trabalho. O roteiro do nosso cenário consiste das seguintes atividades para o desenvolvimento de uma nova funcionalidade para um site:

1. A criação de uma ramificação para uma nova história em que você está trabalhando, portanto, a criação de um novo branch.
2. Adicionar ou modificar um código-fonte nesse ramo e comitá-lo na sequência.

Tudo estava indo bem, até que você recebe uma ligação informando que há um problema crítico que precisa ser corrigido no site, portanto, é esperado que você siga as seguintes tarefas:

1. Mude para a ramificação de produção.

2. Crie uma nova ramificação para adicionar a correção do problema reportado.
3. Depois de testado, mescle a ramificação de correção e envie para produção.
4. Volte para seu ramo original e continue trabalhando.

A operação **merge** pode ser feita seguindo uma das abordagens. Vamos conferi-las!

#### Straight merge

Essa estratégia consiste em obter o histórico de um branch e mesclá-lo com o histórico de outro. Portanto, essa operação de merge deve ser usada quando se realiza um pull, e para implementar essa possibilidade, basta criar um novo arquivo e realizar um add e commit no repositório Git.

Quando usamos o comando `git pull`, o seu repositório local será atualizado com a versão mais recente de um repositório remoto. Ao realizar esse commit no branch, pode-se notar que ele ainda não existe no branch main. Para isso, basta realizar um merge entre ambos. Você precisará alterar para o branch que receberá as alterações realizadas.

Por fim, basta usar o `git merge` com o nome do branch que será mesclado com o branch atual (main) e a operação será feita.

#### Squashed commits

Ao usarmos essa abordagem de operação de merge, obteremos o histórico de um branch e iremos comprimi-lo (squash) no próximo commit para ser colocado no topo de outro branch. Você deve usar essa estratégia quando os branches desenvolvidos adicionarão apenas funcionalidades ao código-fonte já criado. Portanto, implementa-se as funcionalidades, por exemplo, para corrigir algum erro dentro de um branch para então proceder com um commit squashed para mesclar as alterações.

#### Cherry-picking

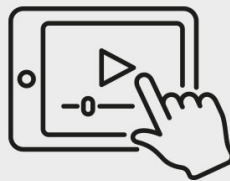


Essa operação merge deve ser feita quando é necessário efetivar o merge especificamente para apenas um commit entre branches, logo, não almejamos fazer um commit para todas as alterações. Para proceder, utilize o seguinte comando: `git cherry-pick <commit específico>`. Você deve usar tal operação merge quando estiver trabalhando em projeto de médio e grande porte em que necessita, por exemplo, de correções de problemas encontrados e que estes sejam corrigidos de forma mais específica por determinado commit.



## Veja agora branch e merge e saiba como e para que aplicar no desenvolvimento

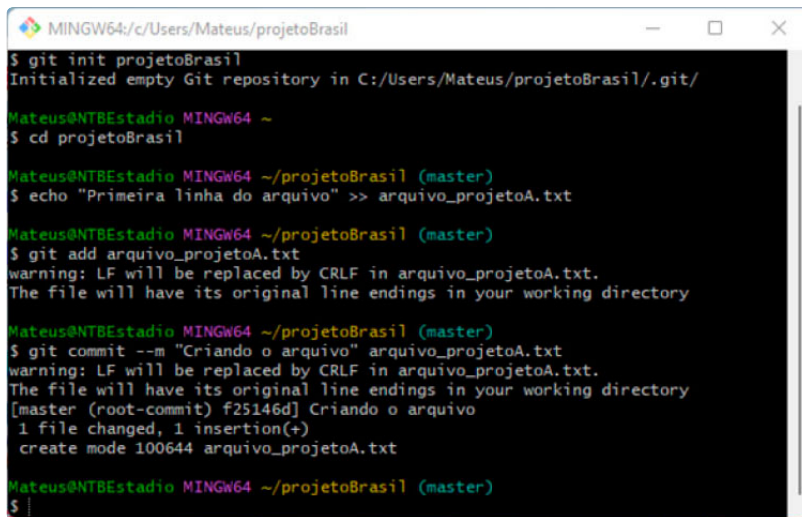
Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



## Operação DIFF

Para verificar o que foi modificado em seu branch, você pode usar os comandos **git status** e **git diff**. Se o comando `git status` não foi suficiente para que seja possível identificar exatamente o que foi modificado em sua ramificação, você pode utilizar o comando `git diff`. Com esse comando, identificamos as linhas que exatamente foram adicionadas ou removidas de um arquivo, facilitando, dessa maneira, a comparação por parte do desenvolvedor.

Vamos analisar de forma prática como executar essa verificação. Inicie criando um novo repositório, crie um arquivo, adicione ao repositório e commit o arquivo no repositório, conforme exibido na imagem.



```
MINGW64/c/Users/Mateus/projetoBrasil
$ git init projetoBrasil
Initialized empty Git repository in C:/Users/Mateus/projetoBrasil/.git/

Mateus@NTBEstadio MINGW64 ~
$ cd projetoBrasil

Mateus@NTBEstadio MINGW64 ~/projetoBrasil (master)
$ echo "Primeira linha do arquivo" >> arquivo_projetoA.txt

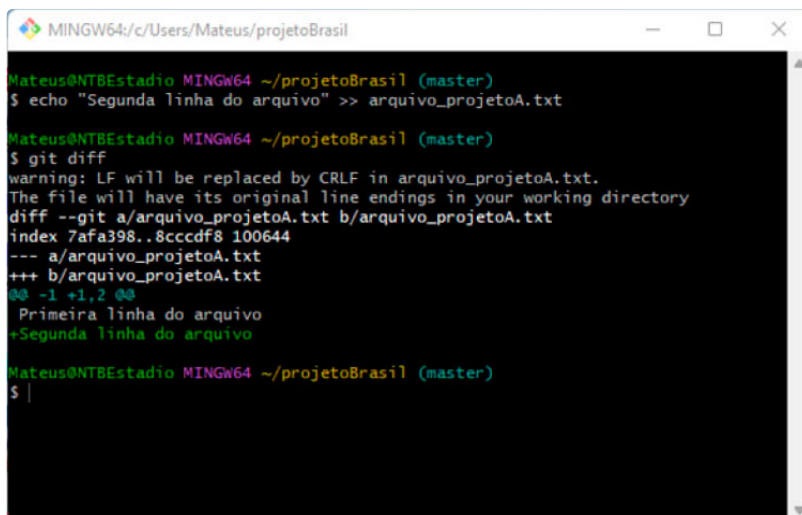
Mateus@NTBEstadio MINGW64 ~/projetoBrasil (master)
$ git add arquivo_projetoA.txt
warning: LF will be replaced by CRLF in arquivo_projetoA.txt.
The file will have its original line endings in your working directory

Mateus@NTBEstadio MINGW64 ~/projetoBrasil (master)
$ git commit --m "Criando o arquivo" arquivo_projetoA.txt
warning: LF will be replaced by CRLF in arquivo_projetoA.txt.
The file will have its original line endings in your working directory
[master (root-commit) f25146d] Criando o arquivo
1 file changed, 1 insertion(+)
create mode 100644 arquivo_projetoA.txt

Mateus@NTBEstadio MINGW64 ~/projetoBrasil (master)
$
```

Criando repositório projetoBrasil.

Agora, vamos adicionar mais uma linha no arquivo e, na sequência, executar o comando Git diff sem adicionar qualquer parâmetro para verificar o seu retorno. Assim, identificaremos as alterações.



```
MINGW64/c/Users/Mateus/projetoBrasil

Mateus@NTBEstadio MINGW64 ~/projetoBrasil (master)
$ echo "Segunda linha do arquivo" >> arquivo_projetoA.txt

Mateus@NTBEstadio MINGW64 ~/projetoBrasil (master)
$ git diff
warning: LF will be replaced by CRLF in arquivo_projetoA.txt.
The file will have its original line endings in your working directory
diff --git a/arquivo_projetoA.txt b/arquivo_projetoA.txt
index 7afa398..8cccdf8 100644
--- a/arquivo_projetoA.txt
+++ b/arquivo_projetoA.txt
@@ -1,2 @@
 Primeira linha do arquivo
+Segunda linha do arquivo

Mateus@NTBEstadio MINGW64 ~/projetoBrasil (master)
$
```

Comparando as modificações no arquivo.

Observe que o comando git diff, sem parâmetros, permite visualizar as alterações ainda não selecionadas para o commit. A linha diff --git a/arquivo\_projetoA.txt b/arquivo\_projetoA.txt mostra as fontes de entrada da comparação.

Na linha --- a/arquivo\_projetoA.txt temos marcações atribuídas a essa fonte de dados que não representa modificação. No entanto, +++ b/arquivo\_projetoA.txt, o símbolo +++ caracteriza que o arquivo sofreu alterações. O resultado restante da comparação é uma lista de "fragmentos" de comparação. Uma comparação exibe apenas as seções do arquivo que tem alterações. No exemplo apresentado, temos apenas um fragmento, pois estamos trabalhando com um cenário simples, conforme mostrado na imagem apresentada a seguir. Nesta imagem, temos -1 +1, significando que a linha 1 teve alterações.

```
@@ -1 +1,2 @@
Primeira linha do arquivo
+Segunda linha do arquivo

Mateus@NTBEstadio MINGW64 ~/projetoBrasil (master)
$ |
```

Lista de “fragmentos” de comparação.

---

Lembre-se de que o arquivo só apresentará a modificação feita após você executar o comando git commit.

Neste módulo, você pôde entender mais sobre a criação de ramificações, fusão de arquivos e as possibilidades de tratamento de conflitos, e, por último, aprendeu a identificar as alterações feitas em um arquivo.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

### Questão 1

Analise as afirmativas:

I. O comando git merge não altera nenhum commit existente, ele apenas cria um novo commit de fusão, portanto, mesclagem entre dois branches por exemplo.

Porque

II. Se não houver nenhum conflito nos arquivos que você modificou no seu branch com o ramo principal, isso significa que o commit não foi executado com sucesso.

A

I é verdadeira, II é verdadeira e a II justifica a I.

B

I é verdadeira, II é verdadeira, porém a II não justifica a I.

C

I e II são falsas.

D

I é falsa e II é verdadeira.

E

I é verdadeira e II é falsa.

Parabéns! A alternativa E está correta.

A operação merge tem como objetivo ser executada e fundir alterações que foram feitas em arquivos no sistema de controle de versionamento Git entre dois ou mais ramos por meio da instrução commit. Eventualmente, conflitos dessas alterações podem ocorrer

e, dessa forma, o desenvolvedor precisará analisar cada um para proceder da melhor forma possível.

## Questão 2

Analise as afirmativas:

I. Para comparar as modificações feitas em determinado arquivo em um repositório git, deve-se usar o comando: git diff.

Porque

II. O comando git cherry pick deve ser usado quando se pretender especificar um commit para determinado ramo a ser executado.

**A** I é verdadeira, II é verdadeira e a II justifica a I.

**B** I é verdadeira e II é verdadeira, porém a II não justifica a I.

**C** I e II são falsas.

**D** I é falso e II é verdadeira.

**E** I é verdadeira e II é falsa.

**Parabéns! A alternativa B está correta.**

Ambas as assertivas estão corretas, porém, ao usar o comando git diff, apenas identificamos o que foi modificado, e não há relação direta na especificação de um commit a ser usado para a instrução git cherry pick.

## Considerações finais

Neste conteúdo, você compreendeu os conceitos, a estrutura e os comandos do sistema de controle de versionamento Git. Pôde perceber a importância de sua adoção para gerenciamento de código-fonte por parte do time de desenvolvedores e de que maneira ele pode ser utilizado para que as atividades desempenhadas pelos programadores sejam harmônicas, eficientes e produtivas.

Para encerrar, ouça um pouco mais sobre como controlar o código-fonte com o GitHub.

Para ouvir o *áudio*, acesse a versão online deste conteúdo.



## Explore +

**Confira as indicações que separamos para você!**

Pesquise a documentação oficial no site do Git ([git-scm](https://git-scm.com)).

Pesquise o controle de código usando VS Code ([code.visualstudio](https://code.visualstudio.com)).

# Referências

ABILDSKOV, J. **Practical Git**. 1. ed. New York: Apress, 2020.

GEISSHIRT, K.; ZATTIN, E.; OLSSON, A. **Git Version Control Cookbook – Second Edition**. 1. ed. New York: Apress, 2018.

LIBERTY, J. **Git for Programmers**. 1. ed. Birmingham, UK: Packt, 2021.



## Material para download

Clique no botão abaixo para fazer o download do conteúdo completo em formato PDF.



Download material

O que você achou do conteúdo?



Relatar problema