



# git Essentials

Versão 1.0

Henry E.L. Cagnini  
Thiago C. Naidon

2023



HENRY E. L. CAGNINI, THIAGO C. NAIDON

# **git Essentials**

**Versão 1.0**

Editora CTISM – UFSM

Santa Maria, RS - 2023

**Presidente da República**

Luiz Inácio Lula da Silva

**Ministro da Educação**

Camilo Santana

**Reitor da Universidade Federal de Santa Maria**

Luciano Schuch

**Diretor geral**

Rafael Adaime Pinto

**Vice diretor**

Fredi Zancan Ferrigolo

**Diretor de ensino**

Deivis Jhones Garlet Bonaldo

**Diagramação**

Henry E. L. Cagnini, Thiago C. Naidon

C131g

Cagnini, Henry E. L.

git Essentials : versão 1.0 / Henry E. L. Cagnini, Thiago C. Naidon – Santa Maria, RS: CTISM/UFSM, 2023.

1. Programa de computador. 2. Ferramenta de versionamento.  
3.Repositório. I.Naidon, Thiago C. II. Título

**ISBN 978-65-00-61487-9**

CDU 004.42

Ficha Catalográfica Elaborada por Rejane Rataeski Moraes da Silva CRB-10/1703



---

# Sumário

---

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Instalação</b>	<b>3</b>
2.1	Instruções para o Windows . . . . .	3
2.2	Instruções para o Linux . . . . .	4
<b>3</b>	<b>Criando repositórios</b>	<b>7</b>
<b>4</b>	<b>Clonando um repositório</b>	<b>9</b>
<b>5</b>	<b>Trabalhando localmente</b>	<b>11</b>
<b>6</b>	<b>Enviando para nuvem</b>	<b>13</b>
<b>7</b>	<b>Resolvendo conflitos</b>	<b>17</b>
7.1	Definindo o método de conciliação de conflitos . . . . .	18
7.2	Finalizando . . . . .	19
7.3	Nenhum conflito detectado . . . . .	21
<b>8</b>	<b>Fazendo fork e pull requests</b>	<b>23</b>
8.1	Passo-a-passo . . . . .	23
8.2	Fluxograma . . . . .	27
<b>9</b>	<b>Conceitos</b>	<b>29</b>
9.1	Repositório . . . . .	29
9.2	Branch . . . . .	30
9.3	Commit . . . . .	30
9.4	Qual a diferença entre repositório remoto e local? . . . . .	31
<b>10</b>	<b>Comandos</b>	<b>33</b>
10.1	Comandos do sistema operacional . . . . .	33
10.2	Comandos do git . . . . .	35
<b>11</b>	<b>Recursos adicionais</b>	<b>39</b>





# CAPÍTULO 1

---

## Introdução

---

Este material organiza o conhecimento sobre o uso de git e Github. O git é uma ferramenta de **versionamento**. Quando estamos trabalhando no desenvolvimento de um trabalho que demora muito tempo para ser concluído (por exemplo, desenvolver um algoritmo, um software, ou até mesmo escrever um relatório de estágio), é desejável que possamos salvar versões **intermediárias** do nosso trabalho, de maneira que possamos recuperá-lo no futuro. Por exemplo, a primeira versão de um software pode possuir uma funcionalidade que foi removida em uma segunda versão; se, todavia, deseja-se reimplementar esta funcionalidade na terceira versão do software, como faríamos para recuperar o código-fonte, se ele já foi excluído? Com o git, isto é possível!

O git foi desenvolvido por [Linus Torvalds](#) em 2005 para desenvolver o Linux, dado que ele precisava, justamente, controlar as diversas versões das funcionalidades implementadas neste sistema operacional.

O git é um programa instalado no seu computador. A maneira mais popular de usar o git é pela linha de comando, através de sua CLI (*command line interface*). É possível também usar o git com programas com interfaces gráficas, e também dentro de ambientes de desenvolvimento integrado (IDEs, na sigla em inglês), como Pycharm, VS Code, etc, porém este material lhe ensinará a mexer no git pela linha de comando.

Colocamos em uma pasta no nosso computador o código-fonte que desejamos versionar (em outras palavras, salvar versões dele). Uma pasta do git é chamada de **repositório**. O git cria um *checkpoint* para cada versão dos arquivos neste repositório. Na terminologia do git, os *checkpoints* são chamados de *commits*.

O git é uma ferramenta totalmente *offline*, desconectada da Internet. Ou seja, por mais que tenhamos diversos commits do nosso código-fonte, eles ficarão salvos no nosso computador, localmente. Se nosso computador estragar, perderemos todo nosso trabalho!

Para isto, foram criados sites para salvar commits na nuvem. O GitHub é, na data de escrita deste material, o site mais popular neste sentido. Existem outros, como GitLab, GitKraken,

BitBucket, etc. Este material, no entanto, usará o GitHub como padrão.

É incorreto e ineficiente usar outros métodos para manter controle de versão de código-fonte, como por exemplo:

- Salvar o código em uma conversa com si mesmo no Whatsapp;
- Enviar um e-mail para si mesmo com o código;
- Usar um programa que não foi feito para este propósito (e.g. Microsoft Word, ou Google Docs);
- Salvar o código-fonte em um arquivo na nuvem (por exemplo no Google Drive).

O git não apenas resolve o problema de salvar código-fonte de uma maneira eficiente, como também apresenta soluções para trabalhar colaborativamente. Ou seja, caso dois ou mais programadores precisem trabalhar no mesmo código, o git tem as ferramentas necessárias para integrar as modificações.

# CAPÍTULO 2

---

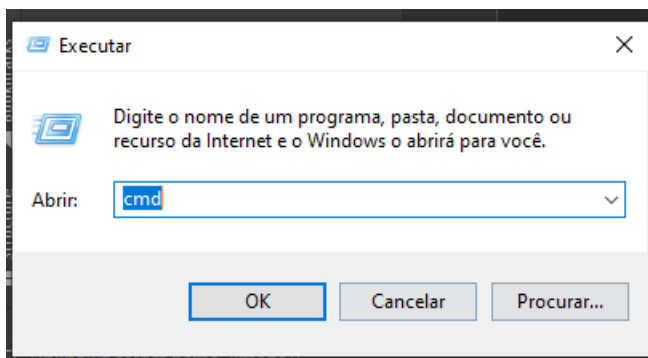
## Instalação

---

Este material cobre a instalação em dois sistemas operacionais, em duas versões específicas: Windows 10 e Linux Ubuntu 20.04 LTS. É provável que as instruções aqui funcionem para outras versões do Windows (e.g. 7, 11), e que as instruções também funcionem para outras distribuições do Linux (e.g. Arch, Debian, etc).

### 2.1 Instruções para o Windows

1. **Baixe o git neste link;**
  - Cuide para baixar a versão adequada (32 ou 64 bits). Provavelmente, seu sistema operacional é 64 bits; se algum problema acontecer na instalação, baixe a versão 32 bits.
2. Siga o passo-a-passo do instalador (apenas clique **Next**, nenhuma opção precisa se modificada);
3. Verifique se o programa foi instalado corretamente. Aperte as teclas **Logotipo do Windows + R** ao mesmo tempo, escreva *cmd* na janela que aparecer, e então **OK**:



4. Se o programa não funcionar, será necessário adicionar o programa ao PATH do sistema manualmente:
  1. Clique no Menu Iniciar;
  2. Digite “Meu Computador” na barra de busca;
  3. No ícone que aparecer, selecione a opção **Propriedades**;
  4. Na lista de opções da direita, selecione a opção **Configurações Avançadas do Sistema**;
  5. Clique no botão **Variáveis do ambiente**;
  6. Na seção *Variáveis do ambiente para o usuário <nome do usuário>*, encontre a variável de nome *PATH*;
  7. Clique duas vezes sobre ela;
  8. Selecione o botão **Novo**;
  9. **Escreva o caminho onde o programa *git* se encontra no seu computador, mais a pasta *bin***;
    - Por padrão, o instalador coloca o git na pasta *C:\Program Files\Git*, portanto você deve digitar o endereço *C:\Program Files\Git\bin* na caixa de texto.
  10. Dê **Ok** em todas as janelas abertas.
5. Feche o prompt de comando e repita o passo 3 novamente.

## 2.2 Instruções para o Linux

1. Abra o Terminal;
2. Digite *sudo apt-get install git*;
3. Insira sua senha;
4. Feche e abra novamente o Terminal, **OU** digite *source ~/.bashrc*;
5. Teste o programa digitando *git --help*. Se uma janela como a abaixo aparecer, estará tudo certo:

```
(base) henry@henry-notebook:~$ git --help
usage: git [-v | --version] [-h | --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
```



# CAPÍTULO 3

---

## Criando repositórios

---

Para criar um novo repositório, acesse o link <https://github.com/new>. A seguinte tela será apresentada:

**Create a new repository**  
A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

---

**Repository template**  
Start your repository with a template repository's contents.

1

---

**Owner \***  / **Repository name \***

Great repository names are short and memorable. Need inspiration? How about [studious-octo-disco](#)?

**Description (optional)**

4

---

5  **Public**  
Anyone on the internet can see this repository. You choose who can commit.

**Private**  
You choose who can see and commit to this repository.

---

**Initialize this repository with:**  
Skip this step if you're importing an existing repository.

6  **Add a README file**  
This is where you can write a long description for your project. [Learn more.](#)

**Add .gitignore**  
Choose which files not to track from a list of templates. [Learn more.](#)

7

**Choose a license**  
A license tells others what they can and can't do with your code. [Learn more.](#)

8

---

Você precisará configurar todos os oito itens. A seguir está a descrição do que cada um deles

faz:

1. **Template:** Se você está copiando a *estrutura* de outro repositório, ou não. Deixe este campo inalterado para este tutorial.
2. **Proprietário:** Se você tiver mais de uma conta no Github (por exemplo, se você fizer parte de uma organização na condição de administrador, como a [CTISM-Prof-Henry](#)), você pode escolher qual conta será a proprietária do repositório. Deixe a sua conta pessoal neste tutorial.
3. **Nome do repositório:** Escolha um nome que faça sentido para o que você estiver armazenando (por exemplo, *codigos\_algoritmos*, ou *codigos\_frontend*)
4. **Descrição:** uma pequena descrição textual para dizer do que se trata o repositório. É opcional, mas sempre é bom preenchê-la.
5. **Público ou privado:** um repositório *público* está acessível para qualquer pessoa na internet; todo mundo pode vê-lo. Já um repositório privado é acessível apenas para você e quem você incluí-lo ( através das configurações do repositório). Prefira sempre fazer repositórios privados.
6. **Adicionar arquivo README:** É um arquivo Markdown com uma descrição mais detalhada sobre as coisas que estão contidas no repositório. Apesar de opcional, é recomendável sempre incluir um arquivo README.
7. **.gitignore:** um arquivo do git usado para ignorar certos arquivos (em outras palavras, evitar que envie-mos estes arquivos para o repositório remoto, e também que o git mantenha um controle sobre as versões destes arquivos). É dependente da linguagem de programação que você está utilizando. Se não houver nada que você queira ignorar no momento, você pode sempre criar o arquivo `.gitignore` posteriormente.
8. **Licença:** como você quer que as pessoas usem seu código-fonte, uma vez que ele vier a público? Pode deixar este campo inalterado para este tutorial.

Após clicar no botão **Create repository**, você poderá clonar o repositório na sua máquina local e começar a mexer nele.





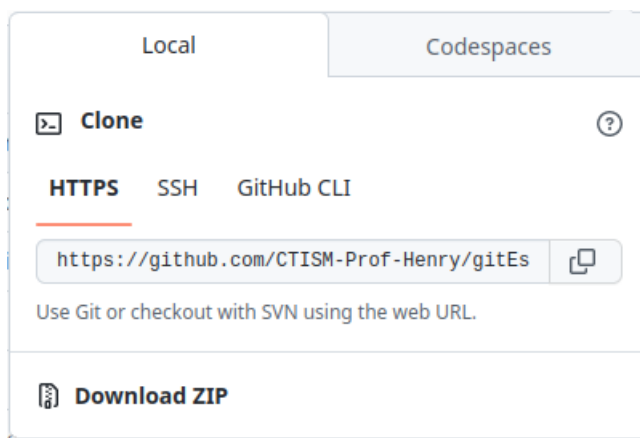
---

## Clonando um repositório

---

Uma vez que você tenha criado um novo repositório, você poderá cloná-lo para a sua máquina local. O ato de **clonar** cria uma cópia do repositório remoto no seu computador. Apesar do GitHub oferecer algumas ferramentas de manipulação de arquivos na sua interface Web (por exemplo, é possível adicionar, remover e modificar arquivos), geralmente, quando estamos trabalhando com código-fonte, queremos executá-lo (e.g. compilá-lo) para ver o resultado.

Na tela inicial do seu repositório recém-criado, clique no botão , e copie a URL do repositório clicando no botão  :



Abra a linha de comando do seu sistema operacional, e, na pasta que você quiser clonar o repositório, digite

```
git clone <URL>
```

Onde <URL> é a URL do seu repositório, copiada anteriormente. Por exemplo, para clonar o repositório deste manual, digite

```
git clone https://github.com/CTISM-Prof-Henry/gitEssentials.git
```

É possível clonar qualquer repositório, inclusive repositórios que não são seus! Tenha em mente apenas que, ao clonar um repositório que não é seu, você não poderá enviar modificações para o repositório remoto.

Após clonar um repositório, é possível trabalhar nele localmente, utilizando git. Apesar do git possuir muitos comandos, veremos na próxima seção o conjunto mínimo de comandos para começar a versionar nosso código-fonte.

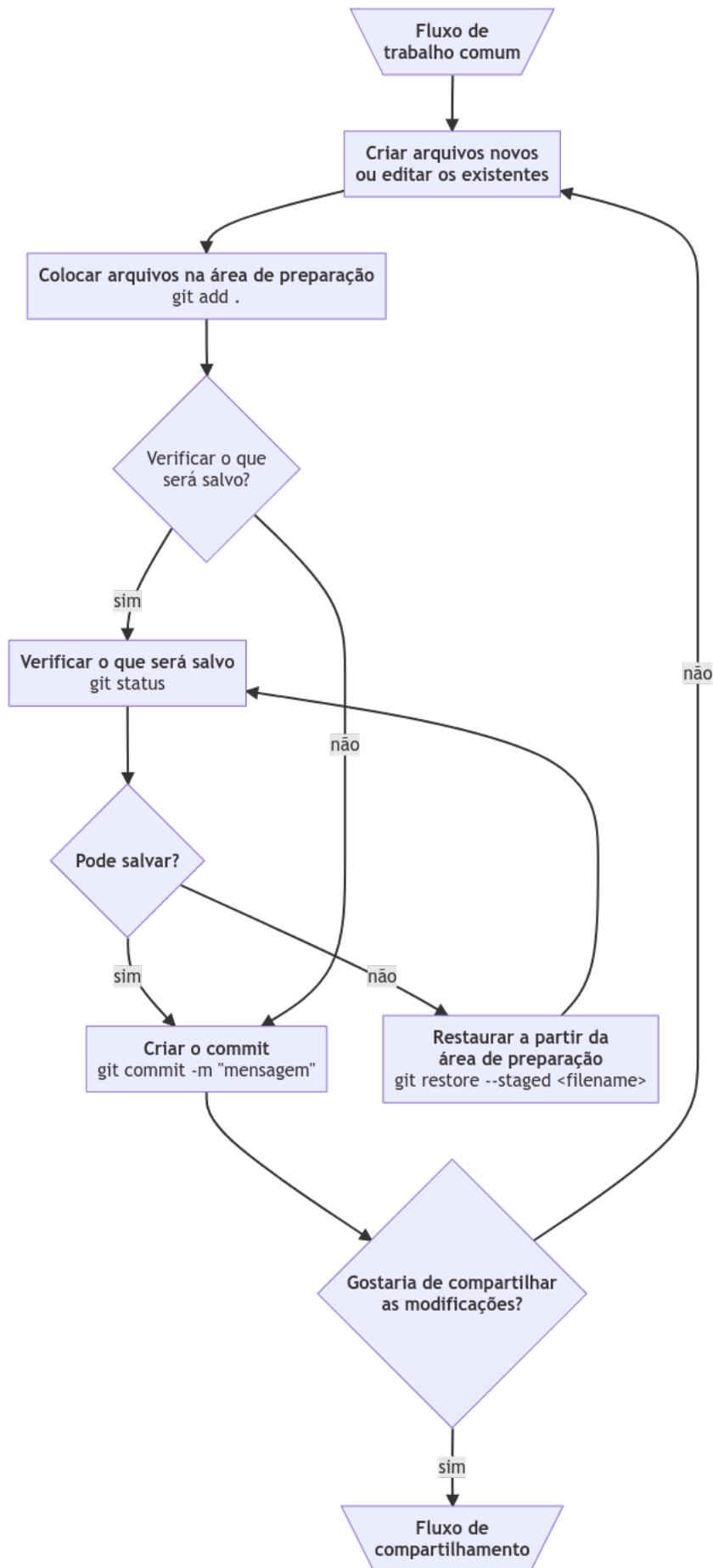
---

### Trabalhando localmente

---

Uma vez clonado um repositório, utilizaremos um conjunto restrito de comandos para trabalhar localmente (ou seja, sem enviar os dados para a nuvem, apenas no nosso computador). Na linha de comando do computador, e dentro da pasta do repositório, utilizaremos os comandos `git status`, `git add`, `git restore`, e `git commit`. Você pode ver uma explicação detalhada para cada um deles na [Seção 10](#). No momento, é mais interessante saber que cada um dos comandos desempenha uma função diferente, e que a **ordem** em que eles são usados importa. Ou seja, utilizar o comando `git commit` (que cria um checkpoint) sem antes preparar os arquivos com `git add` não fará sentido.

Para auxiliar na compreensão da ordem de uso dos comandos, o seguinte fluxograma, adaptado do material de Fabrício Cabral [[CABRAL2022](#)], é mostrado abaixo:



## CAPÍTULO 6

---

### Enviando para nuvem

---

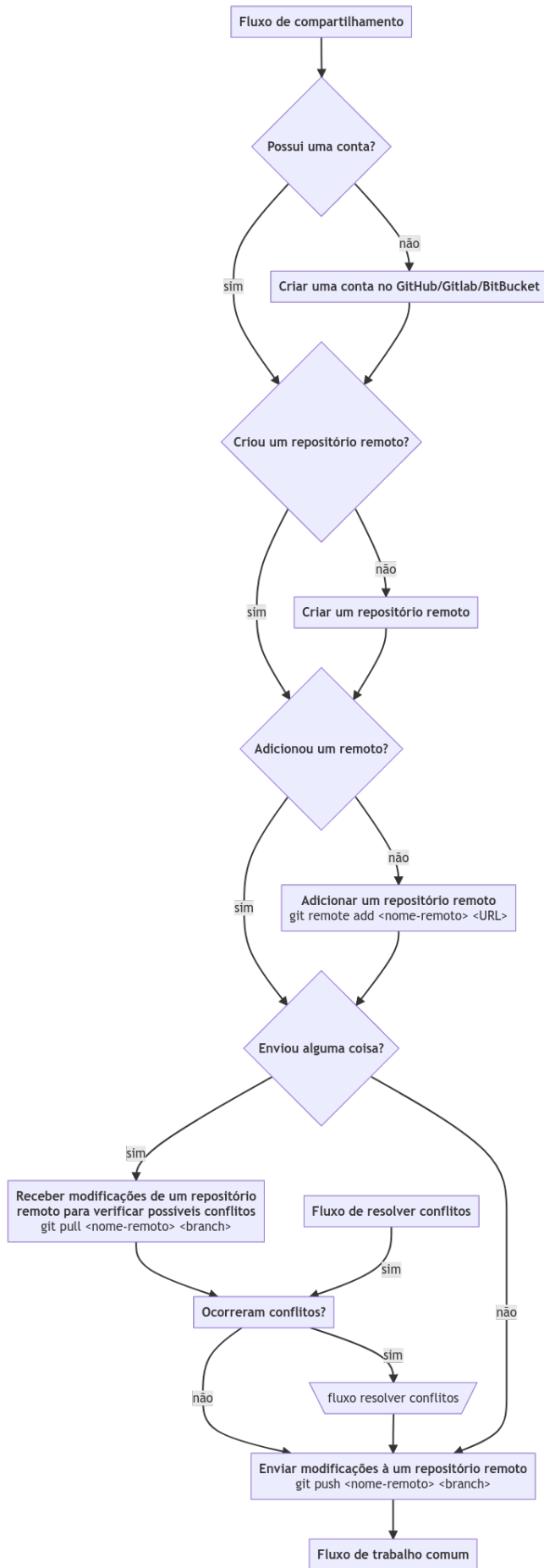
Depois que terminarmos de salvar as modificações localmente, é interessante enviá-las para um repositório remoto, de maneira que exista outra cópia dos nossos commits, em outro computador. Convencionou-se chamar de “nuvem” qualquer serviço que é ofertado na Internet. Portanto, para clareza de explicação, iremos nos referir ao GitHub como um servidor na nuvem, apesar que, estritamente falando, ele é um servidor remoto para armazenamento de repositórios git.

Para que seja possível enviar os commits para o GitHub, é necessário primeiro que o repositório remoto exista. Como este material começa criando um repositório remoto na [Seção 3](#), nos resta apenas enviar o código com `git push`.

O fluxograma abaixo foi adaptado do material de Fabrício Cabral no seu repositório<sup>1</sup>, e mostra como enviamos o código-fonte para o repositório remoto:

---

<sup>1</sup> Disponível em <https://github.com/fabriciofx/gitflowchart>. Acesso em 30/11/2022.



Note que quando **clonamos** um repositório, o nosso repositório remoto já possui uma versão remota. Pode ser que, no processo de enviar o código-fonte para o repositório remoto, apareçam conflitos no código. Isto ocorre porque as versões local e remota estão **desincronizadas**. Em outras palavras, existem modificações em uma das duas versões que não estão presentes na outra versão. Contudo, o git consegue identificar exatamente onde estes conflitos estão, e oferece ferramentas para conciliar as versões diferentes. Um guia de como resolver estes conflitos é apresentado na [Seção 7](#).





---

## Resolvendo conflitos

---

Um conflito ocorre quando duas versões do repositório estão dessincronizadas, e é impossível determinar quais modificações devem ser preservadas, e quais devem ser descartadas. Estas versões não são commits diferentes, mas sim repositórios distintos: por exemplo, um repositório remoto e um repositório local. As modificações introduzidas no repositório remoto podem vir quando clonamos este repositório em outra máquina, que não o nosso computador pessoal, modificamos o código e enviamos ao repositório remoto; ou então quando um colega de trabalho introduz estas modificações a partir do seu computador pessoal.

É possível ver que um conflito ocorreu quando tentamos dar um `git push` e a seguinte tela aparece:

```
(base) henry@henry-notebook:~/Projects/other/gitTest$ git push origin main
To https://github.com/CTISM-Prof-Henry/gitTest.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'https://github.com/CTISM-Prof-Henry/gitTest.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Para auxiliá-lo na compreensão deste fenômeno, é disponibilizado um repositório de código-fonte<sup>1</sup>, usado nas imagens desta Seção. O que está representado na imagem acima diz respeito ao fato de não ser possível enviar o código-fonte para o GitHub pois existem modificações nos arquivos que ainda não colocamos no repositório local. Precisamos então baixá-las, usando o comando `git pull`.

Às vezes, o git consegue conciliar ambas versões do código-fonte (local e remota). Porém, quando isto não é possível, nós mesmos teremos que resolver o conflito que ocorreu nos arquivos. Será necessário o ajuste manual toda vez que, ao executar um comando `git pull`, a tela

---

<sup>1</sup> Disponível em <https://github.com/CTISM-Prof-Henry/gitTest>. Acesso em 01/12/2022.

```
(base) henry@henry-notebook:~/Projects/other/gitTest$ git pull origin main
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 334 bytes | 334.00 KiB/s, done.
From https://github.com/CTISM-Prof-Henry/gitTest
 * branch          main          -> FETCH_HEAD
   a42b5d1..8fa75db main        -> origin/main
hint: You have divergent branches and need to specify how to reconcile them.
hint: You can do so by running one of the following commands sometime before
hint: your next pull:
hint:
hint:   git config pull.rebase false # merge
hint:   git config pull.rebase true  # rebase
hint:   git config pull.ff only     # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set a default
hint: preference for all repositories. You can also pass --rebase, --no-rebase,
hint: or --ff-only on the command line to override the configured default per
hint: invocation.
fatal: Need to specify how to reconcile divergent branches.
```

Ou então a tela

```
(base) henry@henry-notebook:~/Projects/other/gitTest$ git pull origin main
From https://github.com/CTISM-Prof-Henry/gitTest
 * branch          main          -> FETCH_HEAD
Mesclagem automática de README.md
CONFLITO (conteúdo): conflito de mesclagem em README.md
Automatic merge failed; fix conflicts and then commit the result.
```

aparecer. Dependendo de qual tela aparecer, teremos que seguir caminhos diferentes:

- Primeira tela: siga para a [Seção 7.1](#)
- Segunda tela: siga para a [Seção 7.2](#)
- Nenhuma das duas: siga para a [Seção 7.3](#)

## 7.1 Definindo o método de conciliação de conflitos

Na imagem abaixo, o git não sabe como os conflitos devem ser conciliados, e está nos perguntando o que ele deve fazer (texto escrito em amarelo):

```
(base) henry@henry-notebook:~/Projects/other/gitTest$ git pull origin main
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 334 bytes | 334.00 KiB/s, done.
From https://github.com/CTISM-Prof-Henry/gitTest
 * branch          main          -> FETCH_HEAD
   a42b5d1..8fa75db main        -> origin/main
hint: You have divergent branches and need to specify how to reconcile them.
hint: You can do so by running one of the following commands sometime before
hint: your next pull:
hint:
hint:   git config pull.rebase false # merge
hint:   git config pull.rebase true  # rebase
hint:   git config pull.ff only     # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set a default
hint: preference for all repositories. You can also pass --rebase, --no-rebase,
hint: or --ff-only on the command line to override the configured default per
hint: invocation.
fatal: Need to specify how to reconcile divergent branches.
```

Neste material, usaremos a opção **merge**, rodando o seguinte comando:

```
git config pull.rebase false
```

Execute novamente o comando `git pull origin main`:

```
(base) henry@henry-notebook:~/Projects/other/gitTest$ git pull origin main
From https://github.com/CTISM-Prof-Henry/gitTest
 * branch          main          -> FETCH_HEAD
Mesclagem automática de README.md
CONFLITO (conteúdo): conflito de mesclagem em README.md
Automatic merge failed; fix conflicts and then commit the result.
```

Siga para a [Seção 7.2](#) para finalizar a resolução de conflitos.

## 7.2 Finalizando

A mensagem *Automatic merge failed; fix conflicts and then commit the result* quer dizer que teremos que abrir os arquivos que resultaram em conflitos e deixá-los da maneira que eles devem ficar corretamente:

```
(base) henry@henry-notebook:~/Projects/other/gitTest$ git pull origin main
From https://github.com/CTISM-Prof-Henry/gitTest
 * branch          main          -> FETCH_HEAD
Mesclagem automática de README.md
CONFLITO (conteúdo): conflito de mesclagem em README.md
Automatic merge failed; fix conflicts and then commit the result.
```

Saberemos quais arquivos possuem conflitos pela mensagem do git: na imagem acima, o conflito está no arquivo `README.md`.

Abrindo o arquivo `README.md` por um editor de textos (a sugestão dos autores é pelo Sublime Text<sup>1</sup>), vemos que ele se apresenta da seguinte forma:

```
1 # gitTest
2
3 <<<<<<< HEAD
4 Adicionando algo pelo lado direito.
5 =====
6 Adicionando algo pelo lado esquerdo.
7 >>>>>>> 8fa75db4cccadfec3d45a5e8a8ee10c9c6fb1697
8 |
```

- Os caracteres <<<< marcam o começo da região que resultou em um conflito;
- Os caracteres >>>> marcam o fim da região conflitante;
- Os caracteres ===== marcam a divisão do código;
- A seção que começa com HEAD delimita o código-fonte como apresentado no repositório local (do nosso computador);
- A seção marcada pela hash (que varia de acordo com o commit; neste exemplo é a hash `8fa75db4cccadfec3d45a5e8a8ee10c9c6fb1697`) é o código-fonte encontrado no repositório remoto. Na verdade, este hash é o identificador único do commit no repositório remoto.

<sup>1</sup> Disponível em <https://www.sublimetext.com/>. Acesso em 01/12/2022.

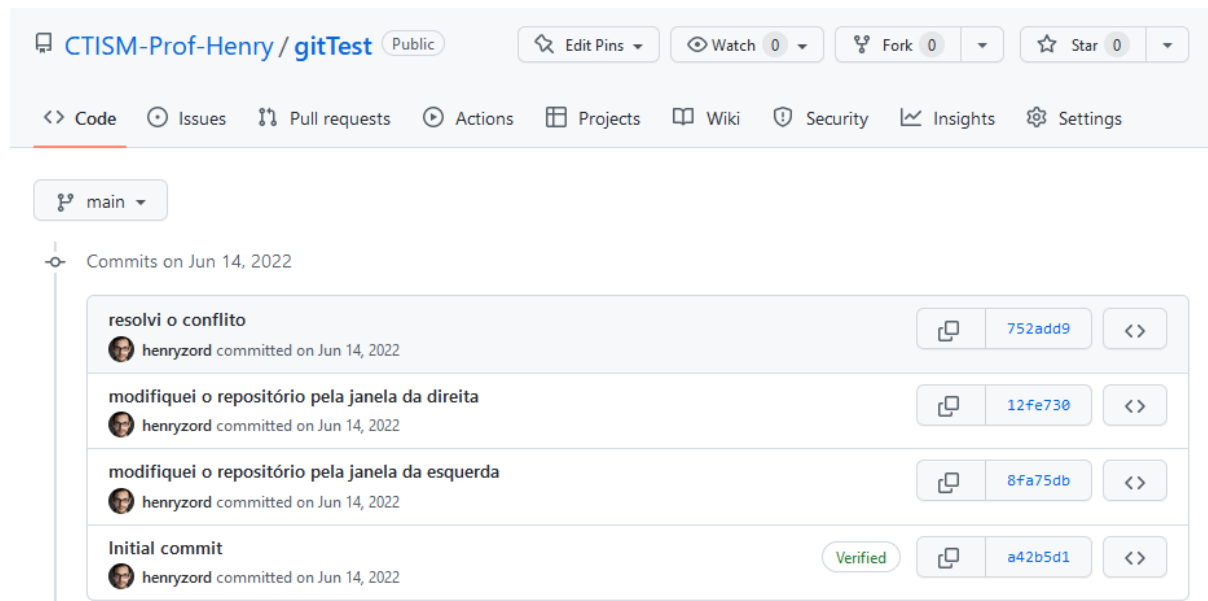
Vamos arrumar o arquivo de maneira que ele concilie as duas modificações, tanto do repositório remoto quanto do repositório local:

```
1 # gitTest
2
3 Adicionando algo pelo lado direito.
4
5 Adicionando algo pelo lado esquerdo.
6
7
```

Após a correção, voltamos à linha de comando para enviar as modificações ao repositório remoto:

```
(base) henry@henry-notebook:~/Projects/other/gitTest$ git add .
(base) henry@henry-notebook:~/Projects/other/gitTest$ git commit -m "resolvi o conflito"
[main 752add9] resolvi o conflito
(base) henry@henry-notebook:~/Projects/other/gitTest$ git push origin main
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 4 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 686 bytes | 686.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/CTISM-Prof-Henry/gitTest.git
 8fa75db..752add9  main -> main
```

Pronto! Com isso, a resolução de conflito que acabamos de fazer estará presente no repositório remoto:



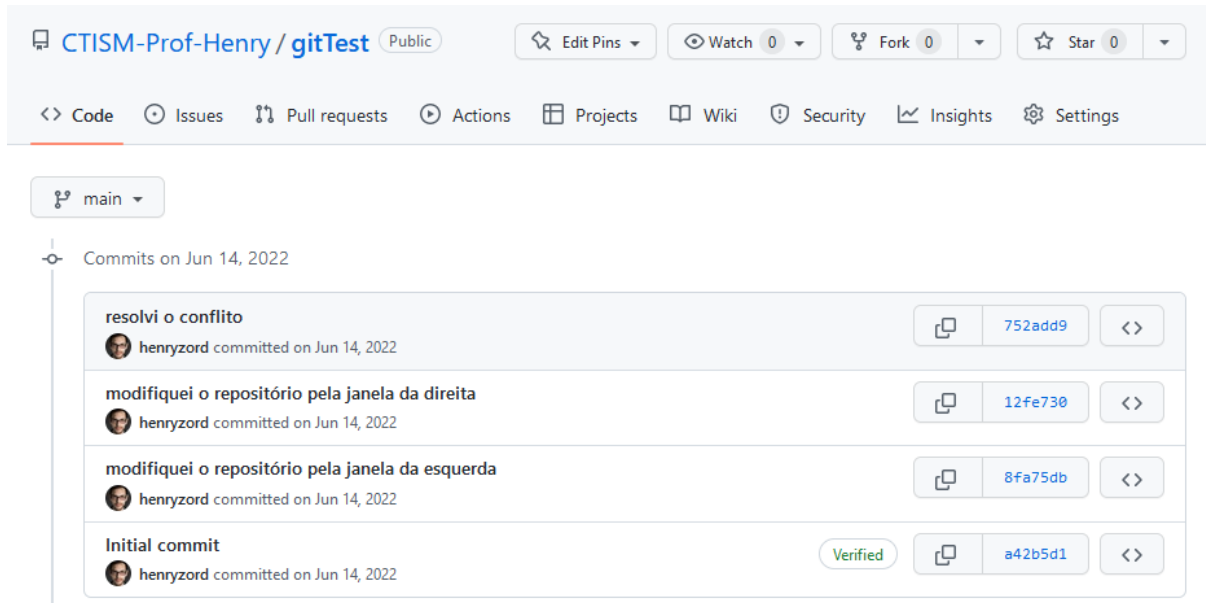
The screenshot shows the GitHub interface for the repository 'CTISM-Prof-Henry / gitTest'. The repository is public and has 0 watches, 0 forks, and 0 stars. The navigation bar includes links for Code, Issues, Pull requests, Actions, Projects, Wiki, Security, Insights, and Settings. The current branch is 'main'. The commit history for June 14, 2022, is displayed as follows:

Commit Message	Author	Commit Hash	Actions
resolvi o conflito	henryzord	752add9	[Copy] [Compare]
modifiquei o repositório pela janela da direita	henryzord	12fe730	[Copy] [Compare]
modifiquei o repositório pela janela da esquerda	henryzord	8fa75db	[Copy] [Compare]
Initial commit	henryzord	a42b5d1	[Verified] [Copy] [Compare]

## 7.3 Nenhum conflito detectado

Se nenhuma das duas telas anteriores se apresentar, quer dizer que com o `git pull`, o `git` conseguiu com sucesso fazer a integração (`merge`) entre os códigos-fonte do repositório remoto e repositório local.

Podemos prosseguir normalmente a partir de então, fazendo um `git push origin main`. Com isso, o código-fonte do repositório local, atualizado às modificações anteriores do repositório remoto, estará no repositório remoto:



The screenshot shows the GitHub interface for the repository 'CTISM-Prof-Henry / gitTest'. The repository is public and has 0 forks and 0 stars. The main branch is selected. The commit history for June 14, 2022, is displayed, showing four commits by user 'henryzord':

Commit Message	Commit Hash	Verification
resolvi o conflito	752add9	
modifiquei o repositório pela janela da direita	12fe730	
modifiquei o repositório pela janela da esquerda	8fa75db	
Initial commit	a42b5d1	Verified



### Fazendo fork e pull requests

---

Uma das principais características do git é servir para o desenvolvimento de código-fonte **co-laborativo**: quando várias pessoas trabalham no mesmo código-fonte, ao mesmo tempo. O git fornece os meios necessários para coordenar o trabalho, de forma que não se perca o foco do trabalho, tentando integrar diferentes versões do mesmo código-fonte.

Uma das modalidades de colaboração com o git é quando fazemos **fork** de um repositório. Um fork é um mecanismo de colaboração em que copiamos o repositório remoto de outra pessoa (fork) para nossa conta do GitHub, fazemos modificações na nossa cópia, e enviamos nossa cópia ao repositório remoto original (pull request), para que as modificações sejam integradas neste. A aceitação das modificações realizadas dependem da aceitação dos mantenedores do repositório remoto original.

#### 8.1 Passo-a-passo


A seguir são descritos os passos necessários para fazer um fork e posteriormente um pull request em um repositório de outro usuário do GitHub.

1. Acessar a página do repositório que deseja-se fazer o fork, no github. Vamos chamá-lo de `https://github.com/CTISM-Prof-Henry/gitEssentials`;
2. Clicar no botão “fork”, no canto superior direito;
3. Confirmar que deseja-se fazer fork na tela que aparecer:

### Create a new fork


A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

**Owner \***      **Repository name \***

 henryzord ▾ / gitEssentials ✓

By default, forks are named the same as their parent repository. You can customize the name to distinguish it further.

**Description (optional)**

 You are creating a fork in your personal account.

Creating repository...

4. Clonar o repositório que foi recém criado (a cópia, não o original) com `git clone`. Por exemplo, se eu, henryzord, fizer um fork do repositório **gitEssentials** (que pertence à conta CTISM-Prof-Henry), a URL do meu repositório copiado será `https://github.com/henryzord/gitEssentials`, e o comando a ser executado é

```
git clone https://github.com/henryzord/gitEssentials
```

5. Criar uma nova **branch local**, e mudar para ela: `git checkout -b <nome da branch>`. Supondo que queiramos criar uma nova branch de nome `top`, o comando a ser executado é

```
git checkout -b top
```

6. Notificar o git de que este repositório relaciona-se com o repositório original: `git remote add upstream <url do repo original>`. No exemplo, ficaria

```
git remote add upstream https://github.com/CTISM-Prof-Henry/  
→gitEssentials
```

7. Fazer as modificações necessárias no código-fonte (editar, deletar ou criar arquivos);
8. Adicionar arquivos com

```
git add .
```

9. Salvar modificações com

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

10. Executar um `git pull` para atualizar o repositório local com as modificações do repositório remoto original

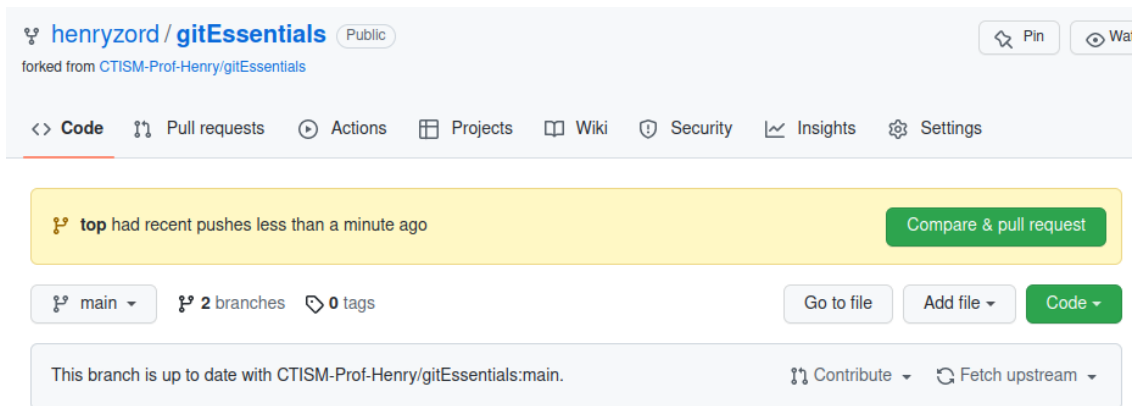


```
git pull origin main
```

11. Enviar modificações para o repositório copiado, em uma **branch remota** que será criada, chamada **top**:

```
git push -u origin top
```

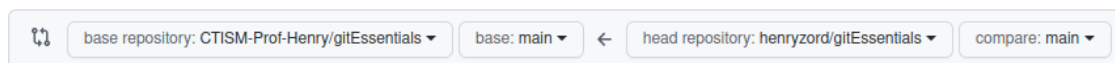
12. Depois que enviarmos as modificações para o nosso repositório copiado, podemos acessá-lo pelo GitHub. Perceberemos uma mensagem na tela inicial, mostrando o quão atualizado nosso código-fonte deste repositório está em relação ao código-fonte do repositório original:



13. Enviaremos as modificações que fizemos no repositório copiado para o repositório original. **Atenção:** iremos fazer isto apenas quando tivermos terminado de fazer **todas** as modificações necessárias no nosso repositório copiado. No nosso repositório copiado, como mostrado na figura acima, vamos clicar no botão **Pull requests**;
14. Clicamos no botão **new pull request**;
15. Abrirá uma tela que irá comparar o nosso código-fonte copiado com o código-fonte original. Como a visualização padrão do GitHub é para a branch **main**, não irá mostrar nada significativo, pois criamos uma nova branch **top**:

### Comparing changes

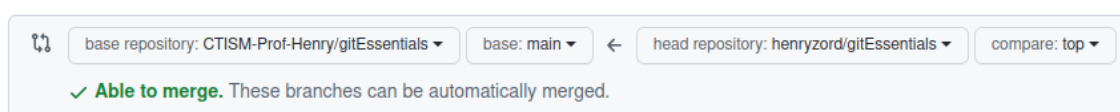
Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).



16. No seletor do repositório copiado, mudamos para a branch **top**:

### Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).

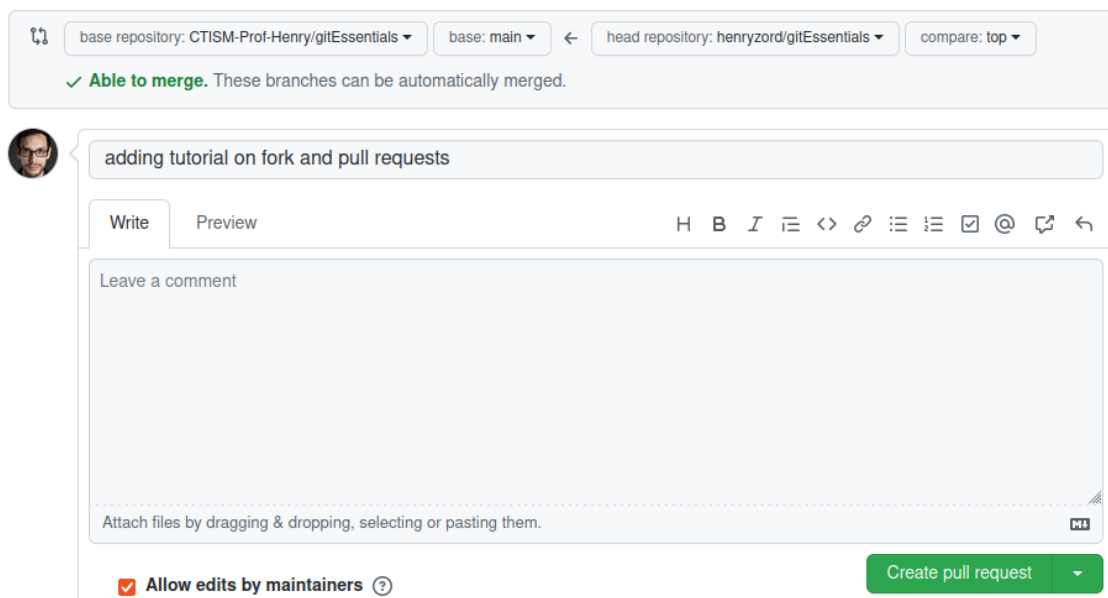


17. Clicamos no botão **create new pull request**;

- Neste passo poderemos escrever uma mensagem para o administrador do repositório original, explicando as modificações feitas no pull request. Depois de escrever a mensagem, clicamos em **create pull request**.

### Open a pull request

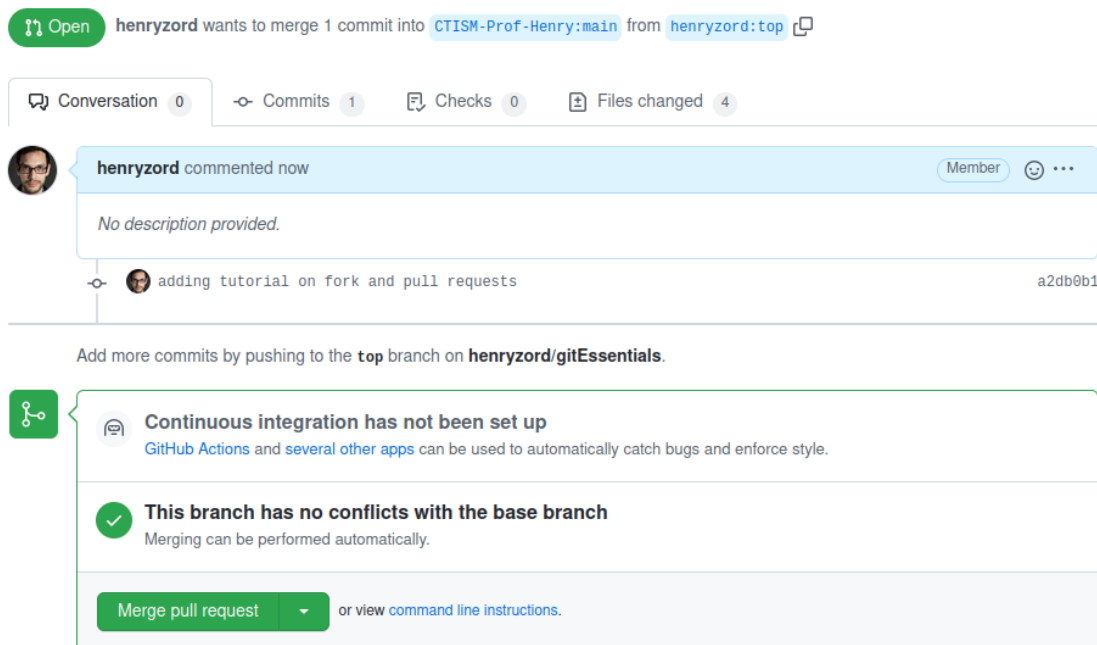
Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).



The screenshot shows the GitHub interface for creating a pull request. At the top, there are dropdown menus for 'base repository: CTISM-Prof-Henry/gitEssentials', 'base: main', 'head repository: henryzord/gitEssentials', and 'compare: top'. A green checkmark indicates 'Able to merge. These branches can be automatically merged.' Below this is a text input field with the title 'adding tutorial on fork and pull requests'. There are 'Write' and 'Preview' tabs, and a rich text editor toolbar with icons for bold, italic, link, and other formatting options. A large text area is labeled 'Leave a comment'. At the bottom, there is a checkbox for 'Allow edits by maintainers' and a green 'Create pull request' button.

- Esta tela é o que o **administrador do repositório original** verá, no GitHub dele:

### adding tutorial on fork and pull requests #1

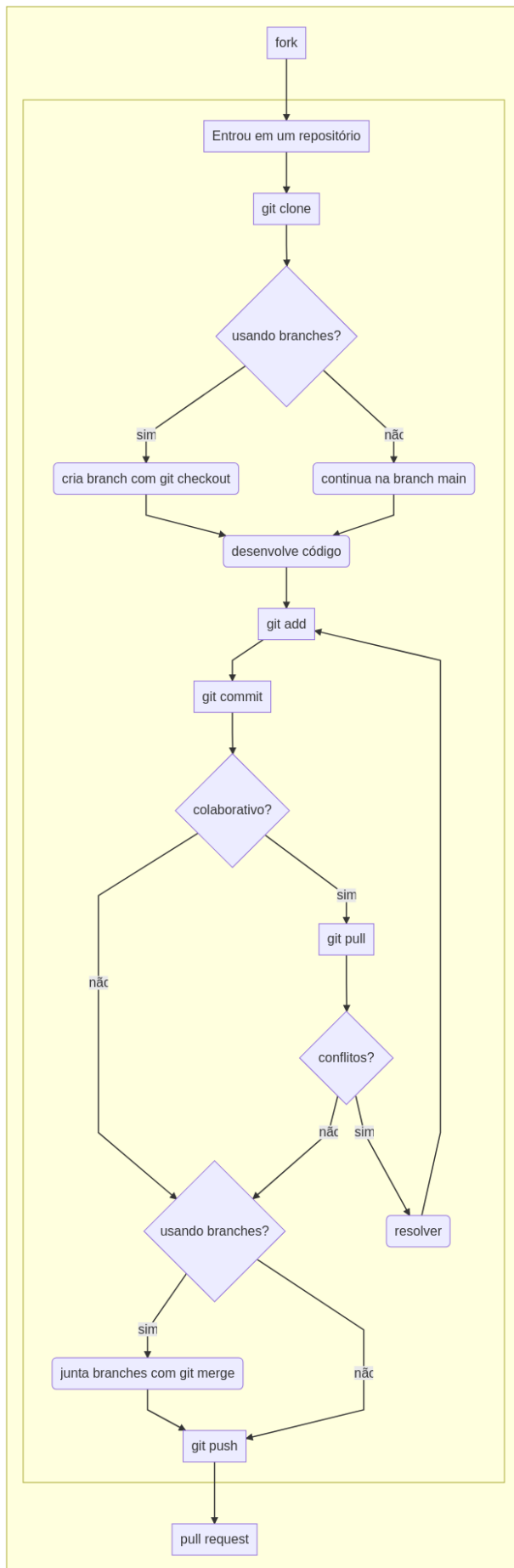


The screenshot shows the GitHub pull request view from the perspective of the repository administrator. At the top, it says 'Open henryzord wants to merge 1 commit into CTISM-Prof-Henry:main from henryzord:top'. Below this are statistics: 'Conversation 0', 'Commits 1', 'Checks 0', and 'Files changed 4'. A comment from 'henryzord' is shown with the text 'No description provided.' and a commit link 'adding tutorial on fork and pull requests' with the hash 'a2db0b1'. Below the comment, there is a message: 'Add more commits by pushing to the top branch on henryzord/gitEssentials.' At the bottom, there is a green box with a warning icon and the text 'Continuous integration has not been set up' and 'GitHub Actions and several other apps can be used to automatically catch bugs and enforce style.' Below this is a green checkmark and the text 'This branch has no conflicts with the base branch' and 'Merging can be performed automatically.' At the bottom of the box is a green 'Merge pull request' button and a link 'or view command line instructions.'

Se ele aceitar nossas modificações, elas serão integradas ao repositório original. Poderemos vê-las na lista de commits do repositório original.

## 8.2 Fluxograma

O fluxograma de trabalho quando estamos trabalhando com forks e pull requests é apresentado na figura abaixo:



Agora que trabalhamos com diversos conceitos do git, é interessante colocá-los em perspectiva, bem como fornecer uma explicação mais detalhada sobre o que cada um deles representa.

### 9.1 Repositório

Também conhecido como *repo* ou *repository*, um repositório é um local para armazenar código-fonte (ou, alternativamente, arquivos não-binários). É possível por exemplo armazenar arquivos de texto como `txt` (não-binário), mas não é possível armazenar arquivos `docx` (binário). A maioria dos arquivos que dizem respeito ao desenvolvimento de um software, algoritmo ou programa são não-binários, e foi com este propósito que os repositórios foram propostos.

Não é recomendável armazenar arquivos binários (e.g. `programa.exe`) em um repositório, pois não poderemos efetivamente manter um controle de versão sobre estes arquivos. Para estes arquivos, é recomendável armazená-los em outros serviços, como Google Drive, One Drive, Dropbox, dentre outros.

Para verificar se um arquivo é binário, tente abri-lo usando um editor de textos, como o bloco de notas: se o texto for não-legível, então o arquivo é binário.

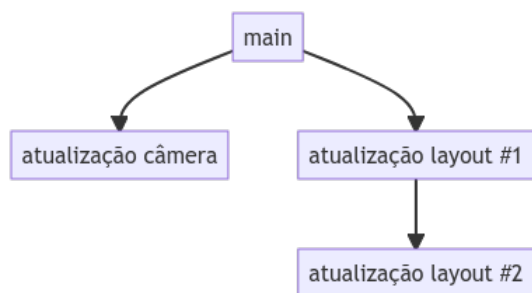
## 9.2 Branch

Um repositório pode conter diversos *branches*, ou “galhos”, na tradução em português (apesar que mesmo programadores brasileiros se referem à estes galhos como *branches*). Branches são uma maneira de manter códigos-fonte similares, porém paralelos, ao mesmo tempo, em um mesmo repositório. Imagine por exemplo uma empresa que desenvolve um aplicativo de tirar fotos e postá-las em uma rede social. Existem diversas equipes de desenvolvedores trabalhando ao mesmo tempo no aplicativo: uma equipe poderia ser responsável por atualizar o software da câmera; enquanto outra muda uma parte do layout. Portanto, usar branches neste cenário é vantajoso, pois as modificações feitas pela equipe do layout não interferirão no código-fonte das outras equipes.

Quando criamos um repositório (seja localmente ou remotamente), é obrigatório criar uma branch principal, onde o código-fonte **de produção** será mantido. A ideia é que nesta branch uma versão pronta do nosso software esteja armazenada, sem bugs e sem tarefas por fazer. Como neste material não temos clientes para os quais vender um software, não tem problema usar a branch main para desenvolver todo o código-fonte de um trabalho. Mas tenha em mente que em uma empresa, esta é a filosofia adotada.

Quando criamos um repositório pelo GitHub, o site cria automaticamente uma branch principal para nós, de nome **main**. É este o mesmo nome que usamos quando usamos os comandos `git push origin main` e `git pull origin main`. Se por outro lado tivermos uma segunda branch em um repositório, de nome **teste**, os comandos seriam `git pull origin teste` e `git push origin teste`.

Você pode criar quantas branches quiser, com os nomes que quiser. Inclusive, você pode ter branches que são branches de outras branches, como mostrado na figura abaixo.



## 9.3 Commit

Um commit é um *checkpoint*, uma versão do código-fonte em uma determinada branch. Ele possui um identificador único, um código de letras e números, chamado de *hash*. A *hash* é única à um commit; nenhum outro commit possuirá a mesma *hash*.

Commits são salvos no repositório (tanto local quanto remoto) para sempre, a menos que sejam explicitamente deletados. Com isso, os arquivos daquele commit estão salvos, mesmo que posteriormente sejam modificados ou deletados. Esta é uma das vantagens de utilizar-se o git, pois podemos ver o histórico de um arquivo, vendo inclusive quem escreveu cada linha de cada arquivo.

Podemos ver a lista de commits de uma determinada branch clicando no botão de commits, na página inicial de um repositório:

A screenshot of a button with a clock icon and the text "6 commits".

Os commits da branch main do repositório que gera este material estão neste link<sup>1</sup>. Para cada um dos commits deste repositório, existem três botões:



O primeiro botão copia a hash do commit. O segundo botão mostrará uma lista das modificações que foram feitas naquele commit. Finalmente, o terceiro link mostra a estrutura do repositório como estava à época deste commit.

## 9.4 Qual a diferença entre repositório remoto e local?

Um **repositório** é uma **pasta** que por sua vez possui outra pasta dentro de si, oculta, chamada `.git`. Dentro da pasta oculta `.git`, existem diversos meta-arquivos que fazem o controle dos arquivos do **repositório**.

Um repositório é um repositório independentemente de onde ele esteja: seja na máquina local (e daí vem o termo **repositório local**), seja em algum site (daí vem o termo **repositório remoto**). Sites que armazenam repositórios são, por exemplo, o [Github](#), [GitKraken](#), [BitBucket](#), [Gitlab](#), dentre outros.

É possível trabalhar com git sem nunca criar uma conta num repositório remoto. Todavia, é recomendável trabalhar com repositórios remotos, para criar cópias do repositório local em outras máquinas. Caso nossa máquina local sofra alguma falha (formatação, falte energia elétrica, etc), nossos dados estarão seguramente armazenados no repositório remoto.

<sup>1</sup> Disponível em <https://github.com/CTISM-Prof-Henry/gitEssentials/commits/main>. Acesso em 01/12/2022.





# CAPÍTULO 10

---

## Comandos

---

Nesta seção veremos uma lista (não exaustiva) dos comandos mais utilizados enquanto usamos git. Para uma lista exaustiva, consulte a documentação oficial [[GIT2022](#)].

### 10.1 Comandos do sistema operacional

Os comandos do sistema operacional são independentes do git, e podem ser utilizados inclusive sem git.

#### 10.1.1 Acessar um novo diretório

Windows, Linux:

```
cd <diretório>
```

Exemplo:

```
cd Downloads
```

## 10.1.2 Listar arquivos em um diretório

Windows:

```
dir <diretório>
```

Linux:

```
ls <diretório>
```

Exemplo:

Windows:

```
dir  
dir Downloads
```

Linux:

```
ls  
ls Downloads
```

## 10.1.3 Limpar tela

Exemplo:

Windows:

```
cls
```

Linux:

```
clear
```

## 10.1.4 Abrir uma janela do navegador de arquivos

Windows:

```
explorer <parâmetro>
```

Linux:

```
nautilus <parâmetro>
```

Exemplo:

Windows:

```
explorer .  
explorer Downloads
```

Linux:

```
nautilus .  
nautilus Downloads
```

## 10.2 Comandos do git

Esta seção apresenta apenas um **resumo** sobre os comandos do git. Cada um destes comandos possui mais parâmetros e funções do que as listadas aqui, porém espera-se que este resumo seja suficiente para programadores iniciantes.

---

**Nota:** é preciso estar dentro de uma pasta que é um repositório git para estes comandos funcionarem.

---

---

**Nota:** Alguns destes comandos dependem do **estado atual** do repositório, que pode ser consultado com um `git status`. Em outras palavras, se você der este comando fora da sequência correta, ele não terá o efeito desejado. Por outro lado, os comandos que **não dependem de estado** são relativamente inofensivos caso foram usados fora da ordem correta.

---

### 10.2.1 git clone

**Copia** um repositório remoto para a máquina local, **se o repositório não existir na máquina local**. Não confundir com a funcionalidade do *git pull*.

Sintaxe:

```
git clone <url do repositório>
```

Exemplo:

```
git clone https://github.com/CTISM-Prof-Henry/gitEssentials
```

### 10.2.2 git status

Mostra o status do repositório na máquina local.

Sintaxe e exemplo:

```
git status
```

### 10.2.3 git add

Adiciona arquivos à lista de modificações-candidatas a serem salvas. Não confundir com a funcionalidade do *git commit*.

Sintaxe:

```
git add <parâmetro>
```

Exemplo:

```
git add . # adiciona todos os arquivos da pasta atual
git add * # adiciona todos os arquivos da pasta atual
git add README.md # adiciona apenas o arquivo README.md
git add README.md main.py estilo.css # adiciona uma lista de arquivos
```

### 10.2.4 git restore

Descarta modificações que foram feitas em um arquivo.

Sintaxe:

```
git restore <parâmetro>
```

Exemplo:

```
git restore README.md # descarta modificações que foram feitas no_
↳README.md
git restore . # descarta modificações que foram feitas nos arquivos_
↳da pasta atual
```

## 10.2.5 git commit

**Salva** as modificações feitas no repositório local, em um checkpoint (também chamado de commit).

**Nota:** só pode ser utilizado após um *git add*.

Sintaxe e exemplo:

```
git commit -m "mensagem explicando o que foi feito neste commit"
```

## 10.2.6 git push

**Uso 1:** Envia modificações da atual branch local para uma branch do repositório remoto, dado que as modificações já foram salvas.

**Nota 1:** só pode ser utilizado após um *git commit*.

**Nota 2:** é uma boa prática ser precedido por um *git pull*.

**Nota 3:** caso você esteja trabalhando em um repositório que é uma cópia de outro repositório (vide [Fazendo fork e pull requests](#)), você deve adicionar a flag `-u` ao comando.

Sintaxe:

```
git push origin <nome da branch remota>
```

Exemplo:

```
git push origin main # envia para a branch remota main
git push origin top  # envia para a branch remota top
git push origin -u top # envia para a branch remota top que
↳ referencia outro repo
```

**Uso 2:** deleta uma branch remota. Veja *git branch* para ver como deletar uma branch local.

Sintaxe:

```
git push origin --delete <nome da branch remota>
```

Exemplo:

```
git push origin --delete top # deleta a branch remota top
```

## 10.2.7 git pull

**Baixa** as modificações da branch de um repositório remoto para a atual branch da máquina local, **se o repositório já existir na máquina local**. Não confundir com a funcionalidade do *git clone*.

Sintaxe:

```
git pull origin <nome da branch remota>
```

Exemplo:

```
git pull origin main # baixa da branch remota main
git pull origin top  # baixa da branch remota top
```

## 10.2.8 git checkout

Muda de uma branch local para outra.

Sintaxe:

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

Exemplo:

```
git checkout top # troca da branch atual para a branch top
git checkout main # troca da branch atual para a branch main
```

## 10.2.9 git branch

**Uso 1:** lista as branches locais.

Sintaxe e exemplo:

```
git branch
```

**Uso 2:** deleta uma branch local. Veja *git push* para deletar uma branch remota.

**Nota:** tenha certeza que você **não está dentro da branch que será deletada**. Veja *git checkout* para ver como trocar de uma branch para outra.

Sintaxe:

```
git branch -d <nome da branch local>
```

Exemplo:

```
git branch -d top # deleta a branch local top
```

# CAPÍTULO 11

---

## Recursos adicionais

---

Esta seção apresenta recursos adicionais que poderão ser utilizados para auxiliar no desenvolvimento de código-fonte com o git e GitHub.

- [StackEdit](#): editor online de texto Markdown (utilizado para escrever documentação no GitHub).
- [Guia Básico de Markdown](#): Manual com listagem de comandos da linguagem Markdown.
- [Biblioteca para grafos em Markdown](#): Extensão do Markdown utilizada para construir grafos, como os que foram utilizados neste guia.





---

## Referências Bibliográficas

---

- [GIT2022] git. Documentação oficial. Disponível em <https://git-scm.com/doc>. Acesso em 01/12/2022.
- [CABRAL2022] Cabral, Fabrício. git flowchart. Disponível em <https://github.com/fabriciofx/gitflowchart>. Acesso em 30/11/2022.