

VERSIONAMENTO

Sumário

O que é Git	4
Serviços de hospedagem Git na Internet	4
Instalando o Git	4
Utilizando o Git	7
Configurando seu usuário local Git	7
Trabalhando com repositórios	7
git init - Inicializando um repositório	7
git clone - Obtendo um repositório remoto	8
Comandos de Snapshotting	8
git status - Verificando se há alterações locais	8
Alterações em repositórios Git	8
git add - Adicionando inclusão e/ou exclusão de arquivos	10
git diff - Exibindo as diferenças entre arquivos	11
git commit - Registrando alterações para o controle de versão	11
git log - Obtendo a listagem de commits no controle de versão	12
git checkout - Visualizando commits anteriores	12
Visualizando estado dos arquivos em um commit	12
Visualizando o estado de um único arquivo	13
git reset - Desfazendo commits	13
Descartado alterações de forma definitiva	14
Desfazendo alterações pendentes de commit	14
Compartilhando atualizações	14
git push - Enviando os commits para o repositório remoto	14
Resolvendo conflitos em arquivos	15
Ajustar conflitos manualmente	16
Resolvendo conflitos revertendo a versão	17
git pull - Obtendo as atualizações do repositório remoto	17
git fetch	17
Verificando se há alterações no repositório remoto	17
Tutorial: Inicializando e publicando um repositório	19

1. Crie uma pasta de projeto	19
2. Inicialize o repositório	19
3. Adicione um arquivo	20
4. Faça um commit	21
5. Crie um repositório no Gitlab	22
6. Envie os commits do repositório local para o repositório no Gitlab	24

Bibliografia	26
---------------------	-----------

O que é Git

Git é um sistema de controle de versão de arquivos (ou sistema de versionamento). Quando um diretório está gerenciado pelo Git, temos o controle sobre os arquivos e diretórios criados, alterados e excluídos nele, podendo saber tudo que foi feito (inclusive por quem e quando), bem como “desfazer” operações. Foi criado pelo “pai” do Linux, **Linus Torvards**, sendo lançado em 2005. É possível usá-lo nos principais sistemas operacionais da atualidade (Windows, Linux e MacOS).

Uma das principais inovações do Git frente aos sistemas de versionamento anteriores, é o fato de que cada diretório gerenciado por ele (chamado de **Repositório**) possui o histórico completo das alterações realizadas e registradas via “commit” (falaremos sobre esse termo adiante). Mesmo assim, é possível conectar um repositório local a um **remoto**, possibilitando que o versionamento também fique em outro computador (inclusive na internet).

O Git, na prática, é um protocolo, um conjunto de comandos executáveis em linha de comando. Porém, existem inúmeras opções de interfaces gráficas como:

- Integrações com o Git em IDEs (Eclipse, IntelliJIdea, NetBeans etc);
- Programas como TortoiseGit, Gitk, SourceTree, GitHub Desktop etc.

Serviços de hospedagem Git na Internet

Como os sistemas de versionamento normalmente são usados para repositórios acessados por um grupo de pessoas, normalmente existe um repositório remoto para centralizar e unificar seu conjunto de arquivos, diretórios e histórico de operações. Esse repositório remoto pode estar numa rede interna da empresa ou na internet.

Os principais serviços de hospedagem de Git na Internet são: **GitHub** (github.com), **GitLab** (gitlab.com) e **Bitbucket** (bitbucket.org). A diferença entre eles está nas limitações quanto a repositórios privados nas modalidades de uso grátis e nas funcionalidades gráficas de gerenciamentos e agile, além da quantidade e tamanho dos repositórios. Apesar de todos terem a mesma funcionalidade básica de armazenar repositórios Git, o foco do material será em **GitLab**. O GitLab também pode ser configurado como serviço numa rede local.

Instalando o Git

Para poder usar o Git, pode ser necessário instalá-lo em seu computador, pois não é um programa nativo em alguns sistemas operacionais.

Para saber se o Git já está instalado em seu SO, abra um terminal e tente executar o comando...

```
git --version
```

Se você ver uma descrição de versão, o Git já está instalado. Caso contrário, receberá a mensagem de comando não encontrado.

Linux: Na maioria das distribuições o Git está disponível como pacote sob o nome **git**. Assim, basta fazer um **apt install git** ou **dnf install git** ou **yum install git**, dependendo da distribuição usada.

MacOS: O git costuma vir instalado nesse SO. Porém, caso não esteja, baixe o instalador em <https://git-scm.com/download/mac> e siga a instruções.

Windows: Baixe o instalador em <https://git-scm.com/download/win> e siga a instruções. Após instalado, use o programa **Git Bash** para executar os comandos git. Trata-se de um terminal com cores customizadas para o Git e que aceita comandos do Linux ao invés de comandos do prompt do Windows. Por exemplo, pode executar **ls**, **cat**, **clear** etc. Outra característica desse terminal é que guarda o histórico de comandos (recuperável usando-se as setas do teclado) mesmo que seja fechado.

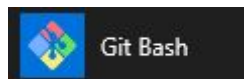


Figura 1: Ícone do Git Bash no Windows

Ao executar o instalador, serão apresentadas diversas telas para realizar configurações. Basta utilizar sempre as opções padrões clicando no botão **Next**.

Se usarmos as configurações padrão, além de podermos utilizar o Git dentro do terminal do Git Bash, também podemos utilizá-lo no **Prompt de Comandos** e no **PowerShell**. Note que, apesar disso, os comandos do Linux só estarão disponíveis no próprio **Git Bash**.

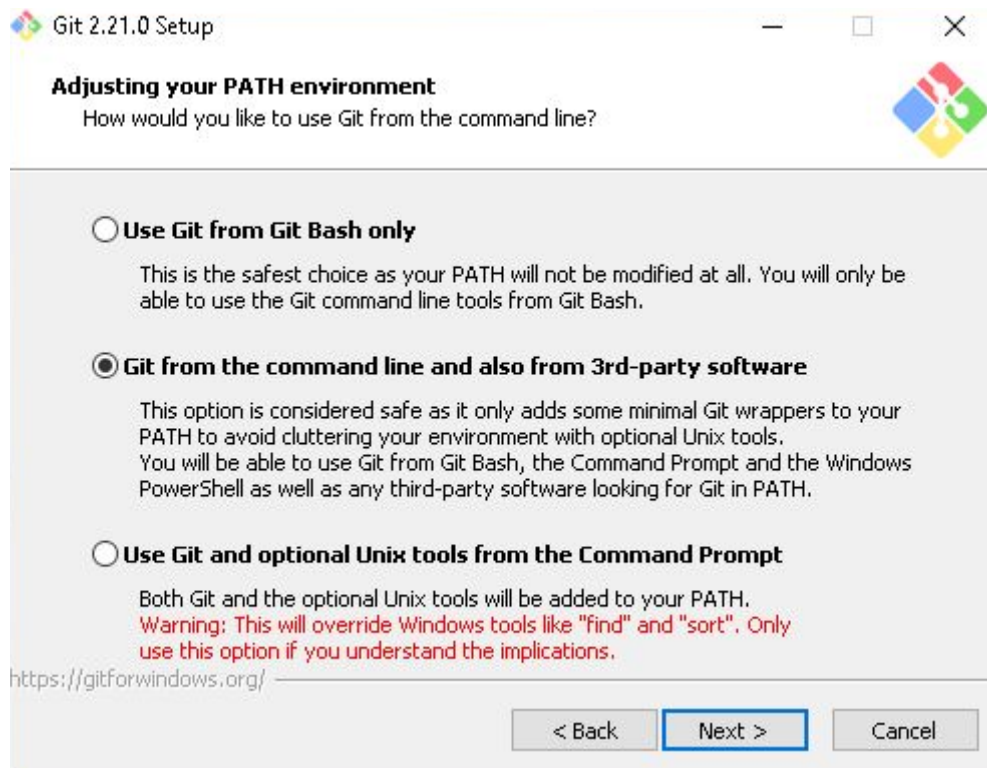


Figura 2: Instalando o Git para uso em diversos terminais no Windows

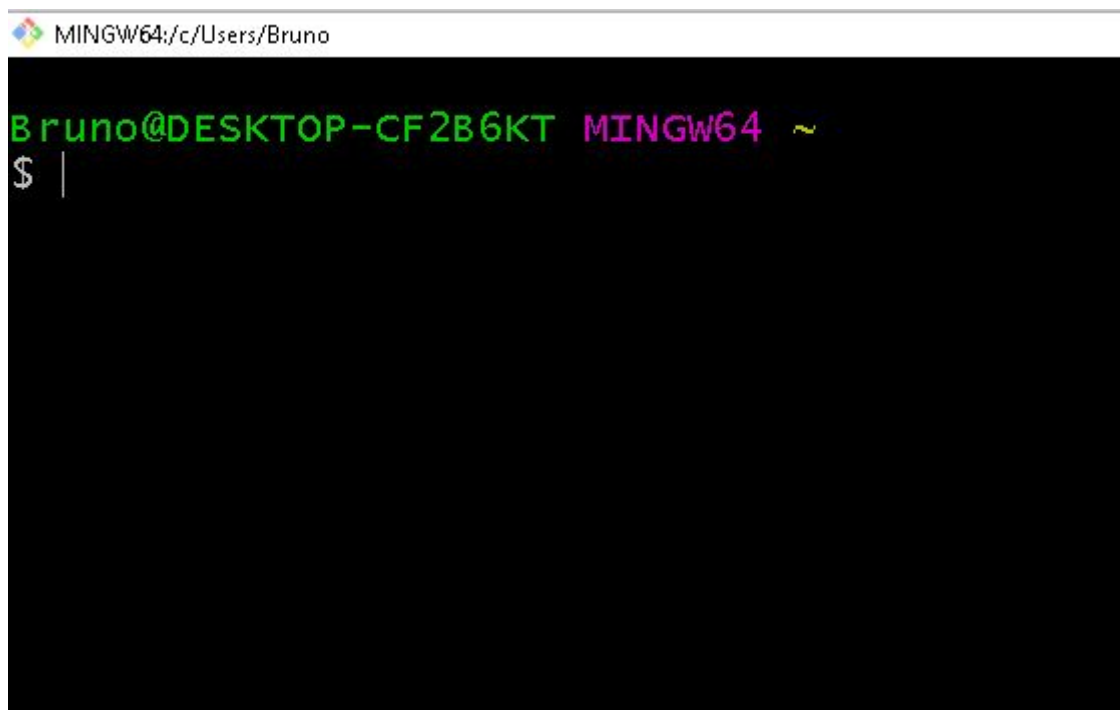


Figura 3: Janela do Git Bash

Utilizando o Git

Configurando seu usuário local Git

Após a instalação, é necessário definir o **nome** e **email** do usuário local Git. Sem essa identificação mínima, não é possível realizar um commit. Para isso, basta executar os comandos...

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

e

```
git config --global user.email "seu_email@seu_provedor.com"
```

Esse comando, quando executado com a flag **--global**, aplica a configuração globalmente no seu computador, o que quer dizer que ele não precisa ser repetido em cada repositório individualmente.

Dica: Nunca use o comando com a flag **--global** em um computador público, visto que isso irá assinar todos os próximos commits feitos em todos os repositórios com as credenciais que você inserir.

Trabalhando com repositórios

git init - Inicializando um repositório

Quando você deseja transformar uma pasta de trabalho em um repositório, basta usar o comando...

```
git init
```

Isso faz com que o Git passe a gerenciar os arquivos e diretórios dentro dessa pasta, assim como todas as alterações que eles sofrerem.

Todo repositório possui em seu diretório raiz uma pasta oculta chamada **.git**. Essa pasta contém todos os metadados sobre aquele repositório. Caso apagarmos essa pasta, perdemos completamente todo o histórico de commits, fazendo que a pasta deixe de ser um repositório.

Podemos nos usar desse conhecimento para desfazer o erro de criar um repositório em uma pasta indesejada.

Um erro muito comum é inicializar um repositório dentro de outro repositório, o que pode fazer com que o Git ignore completamente as alterações feitas no subrepositório. Caso isso ocorra, apague a pasta **.git** do local incorreto.

git clone - Obtendo um repositório remoto

Quando você deseja começar a trabalhar num repositório que já existe, precisa obtê-lo remotamente. Para isso, use o comando...

```
git clone [endereço do repositório]
```

No diretório que deseja que fique o diretório gerenciado pelo git, **não crie o diretório manualmente**. Apenas execute o seguinte comando:

```
git clone  
https://gitlab.com/seu-usuario-gitlab/seu-repositorio-git-lab/
```

Dica: Após o *git clone* esse endereço é o mesmo que seu repositório tem ao acessá-lo pelo navegador.

Serão solicitados seu login e senha do GitLab. **Um diretório com o mesmo nome do repositório será criado** pelo Git. Assim, basta fazer um **cd seu-repositorio-git-lab** para entrar nele.

Lembre-se que ao clonar um repositório, não é necessário realizar o comando **git init**, visto que a pasta criada **já é um repositório**. O comando **git init** só é utilizado quando estamos criando um novo projeto a partir de uma pasta.

Comandos de Snapshoting

A seguir, vamos interagir com o repositório usando os principais comandos do Git. Todos os comandos devem ser executados em um repositório, ou seja, **dentro** da pasta gerenciada pelo Git. Para isso, abra o terminal e utilize os comandos que vimos anteriormente até a pasta desejada.

git status - Verificando se há alterações locais

Para saber se houve alterações em arquivos no repositório local, use o comando...

```
git status
```

Se não houver nenhuma alteração, você deverá ver uma saída como esta:

```
On branch master  
Your branch is up to date with 'origin/master'.  
nothing to commit, working tree clean
```

Alterações em repositórios Git

Quanto ao status de gerenciamento de arquivos, o Git possui 4, que são:

Unstaged: Arquivos que receberam novas alterações. O Git só inclui novas alterações em um envio (**commit**) se explicitamente adicionamos o arquivo a área de **staging** (veja abaixo).

Staged: Arquivos prontos para o envio (**commit**). São todos sobre os quais fizemos o comando **git add**.

Committed: Arquivos cujas alterações foram enviadas (que sofreram **commit**).

Untracked: Arquivos ainda sem monitoramento, ou seja, novos arquivos que foram adicionados ao repositório e nunca fizeram parte de um **commit** e não foram adicionados à área de **staging**

A Figura 4 ilustra a sequência cronológica dos status.

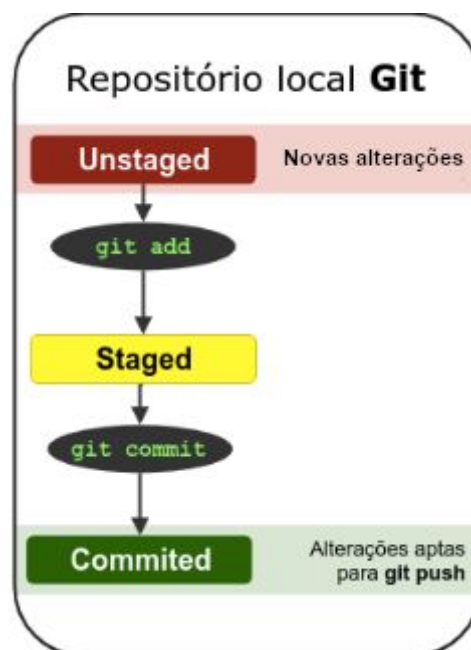


Figura 4: Status de arquivos num repositório Git.

A seguir, um exemplo de saída do **git status** com os 4 tipos de status:

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    new file:   listagem.txt
    modified:   receita.txt
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be
committed)
  (use "git checkout -- <file>..." to discard changes in
working directory)
    deleted:    poema.txt
Untracked files:
  (use "git add <file>..." to include in what will be
committed)
    declaracao.txt
```

git add - Adicionando inclusão e/ou exclusão de arquivos

No Git, criação e a exclusão de arquivos são operações que precisam ser informadas explicitamente antes de serem enviadas ao controle de versão. Usa-se, então, o comando **git add**. Exemplos:

Você acaba de criar o arquivo **poema.txt**, no diretório raiz do repositório:

```
git add poema.txt
```

Você acaba de criar o arquivo **japao.txt**, no diretório **países** do repositório:

```
git add países/japao.txt
```

Você acaba de excluir o arquivo **receita.txt**, no diretório raiz do repositório:

```
git add receita.txt
```

Você acaba de criar e/ou excluir vários arquivos do repositório, inclusive em vários subdiretórios, e quer adicionar tudo de uma só vez:

```
git add .
```

Atenção: O **git add** ainda **não** registra as alterações no controle de versão do repositório! Uma vez feito isso, a(s) alteração(ões) informadas estão prontas para o registro (**commit**). Esse sim, faz com que o controle de versão adicione as informações permanentemente ao repositório.

git diff - Exibindo as diferenças entre arquivos

As diferenças entre arquivos ainda não enviadas via **commit** podem ser vistas usando-se o comando...

```
git diff
```

Após a execução desse comando, a saída é como a seguir:

```
diff --git a/listagem.txt b/listagem.txt
index cf2f7b0..9430a63 100644
--- a/listagem.txt
+++ b/listagem.txt
@@ -1,2 @@
-frutas
+legumes
+massas
```

```
diff --git a/receita.txt b/receita.txt
deleted file mode 100644
index f702bcd..0000000
--- a/receita.txt
@@ -1,3 +0,0 @@
-ingredientes
-instruções
-cuidados
```

Atenção: Arquivos recém criados e sobre os quais não foi executado o comando **git add (untracked)** não aparecem na saída do comando **git diff**.

git commit - Registrando alterações para o controle de versão

O Git não registra alterações por arquivo e sim por “pacotes” de envio de alterações (**commits**). Num commit podem estar registradas criação, alteração ou exclusão de 1 ou mais arquivos. Todo **commit** deve possuir uma mensagem informada pelo usuário Git.

Por exemplo, vamos supor que você já fez todos os **git add** que precisava. Basta executar o comando...

```
git commit -m "Motivo das alterações que estou enviando"
```

Com isso, se você, por exemplo, criou 1 arquivo, alterou 4 e excluiu 2, seu **commit** contemplará 7 arquivos afetados.

A mensagem que resume o motivo daquele commit é obrigatória e com soft limit de **72 caracteres** (boa prática).

Atenção: o conceito de **commits** não impede que possamos reverter alterações em um único arquivo, mesmo que suas alterações tenham ido junto às de outros.

Dúvida comum: É possível enviar a alteração de apenas 1 arquivo num **commit**? Sim. Basta fazer o **git add** somente sobre o arquivo desejado antes do **git commit**.

git log - Obtendo a listagem de commits no controle de versão

Para ver quais commits estão registrados no controle de versão, use o comando

```
git log
```

Ele vai gerar uma saída como o exemplo a seguir:

```
commit 65ebf9015f9c8f03735fd5be014b835368521755 (HEAD ->
master)
Author: José Ruela <jruela@gmail.com>
Date:   Mon Feb 15 08:20:12 2019 -0300
    correções ortográficas em poema.txt
commit c28447fc0f9aa7dae89f5738afe0d5081270f024
Author: José Ruela <jruela@gmail.com>
Date:   Mon Feb 15 07:43:15 2019 -0300
    adiciona nova estrofe em poema.txt
```

Notou um código alfanumérico após cada palavra “commit”? Esse é o **código identificador** de cada **commit**. Sabe aquele “código de rastreamento” de encomendas físicas? É como se fosse isso. O Git associa cada identificador desse a um conjunto de alterações.

git checkout - Visualizando commits anteriores

Podemos pedir para que um arquivo, ou até o repositório inteiro, “volte no tempo” e fique como era até determinado commit. Essa reversão é temporária no caso desse comando, de forma que é simples revertermos para o estado atual dos arquivos.

Visualizando estado dos arquivos em um commit

Vamos supor que deseja que todo o repositório local fique como estava até determinado commit. Encontre o identificador do commit através do comando **git log** e execute...

```
git checkout 63fe1916beddc6ad3a986e4bcf74a9e2ee9f501d
```

Esse comando faz o repositório inteiro “voltar no tempo” e ficar exatamente como ficou após o commit informado. Isso não quer dizer que descartamos todos os commits posteriores a esse! O repositório fica em um estado chamado **detached head**, que indica que o ponteiro interno do git não está mais apontando para o último commit dessa branch, mas sim para um commit intermediário.

Para reverter os arquivos para o estado mais atual, ou seja, de acordo ao último commit da branch, basta executar o comando...

```
git checkout master
```

Visualizando o estado de um único arquivo

Vamos supor que deseja que um o arquivo **ingredientes.txt** volte a ficar como era até um determinado **commit**. Você faz o **git log** e localiza o **identificador** do **commit** desejado. Então, basta fazer:

```
git checkout 63fe1916beddc6ad3a986e4bcf74a9e2ee9f501d  
ingredientes.txt
```

Caso você queira que o mesmo arquivo fique exatamente como está atualmente no repositório remoto, faça 2 comandos:

```
git fetch
```

depois

```
git checkout origin/master ingredientes.txt
```

git reset - Desfazendo commits

Você já teve aquele pensamento “se eu pudesse voltar no tempo e fazer tudo diferente”? Bom, no mundo real não podemos voltar no tempo, mas o Git nos permite, como que, fazer isso em relação a nossos arquivos. O comando reset permite desfazer commits já realizados em um repositório. Note que, diferente do **git checkout**, os commits são efetivamente eliminados da branch.

Para desfazer todos os commits posteriores a um commit específico da branch, podemos usar o comando **git log** para localizar o identificador do commit e executar...

```
git reset 4e35b01632c66c0b655eb79a061f8e9724d47058
```

Se observamos o histórico de commits com o comando **git log**, vemos que todos os commits posteriores ao commit identificado não existem mais. Porém, todo

nosso trabalho permanece presente em forma de alterações pendentes a um commit. Assim, podemos adicionar os arquivos e criar novamente os commits na ordem que desejarmos.

Descartando alterações de forma definitiva

Caso o objetivo seja, além de desfazer os commits, descartar todo o trabalho realizado neles, podemos utilizar o comando...

```
git reset 4e35b01632c66c0b655eb79a061f8e9724d47058 --hard
```

A flag **hard** **descarta de forma irrecuperável** todas as alterações feitas nos commits posteriores ao commit identificado. **Muito cuidado ao usá-la!**

Desfazendo alterações pendentes de commit

Algumas vezes acontece de começarmos um sessão de trabalho e, depois de alguns minutos, decidirmos que é melhor descartar tudo que fizemos e começar do zero. Podemos conseguir isso facilmente usando o comando...

```
git reset --hard
```

Quando executamos o **git reset** e não informamos um identificador de commit, ele automaticamente aponta para o último commit realizado na branch atual. Dessa forma, o **git reset --hard** descarta todas as alterações que não entraram em um commit, deixando o diretório de trabalho limpo (sem alterações para commit). Isso permite que possamos fazer comandos como **git pull** e **git checkout** sem problemas.

Compartilhando atualizações

git push - Enviando os commits para o repositório remoto

Mesmo que você tenha criado seu repositório git a partir de um clone de um remoto, registrar os commits não os envia para ele. É necessário informar, explicitamente, que você deseja enviar todos os seus commits locais para o repositório remoto. Para isso, use o comando:

```
git push
```

Esse comando irá solicitar seus login e senha do GitLab. Ao entrar com eles corretamente e se o push funcionar sem problemas, você deve ver uma saída como a seguir.

```
Counting objects: 5, done.  
Delta compression using up to 4 threads.  
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (5/5), 449 bytes | 224.00 KiB/s, done.  
Total 5 (delta 0), reused 0 (delta 0)  
To https://gitlab.com/jruela/java-queiro  
c28447f..65ebf90  master -> master
```

Pode ocorrer que, ao tentar o **push**, você seja informado que ele foi rejeitado (**rejected**). Isso ocorre quando alguém fez um push para o mesmo repositório remoto após seu último **pull**. Nesse caso, você terá uma saída como esta:

```
! [rejected]          master -> master (fetch first)  
error: failed to push some refs to  
'https://gitlab.com/jruela/java-queiro'  
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 resolver isso, basta fazer um **git pull**. Se aparecer uma tela meio estranha (editor de textos vim), digite **:x** e confirme com o **enter**. Esse é o comando para sair do editor de texto e salvar o arquivo. O texto exibido é apenas um texto informativo. Feito isso, execute o **git push** novamente.

Resolvendo conflitos em arquivos

Pode ocorrer que, ao tentar um **git pull**, você se depare com um problema de conflito (**conflict**). Isso ocorre quando outra pessoa mexeu em um mesmo arquivo que você e o enviou ao repositório remoto após seu último **pull**. Nesse caso a saída será como esta:

```
Unpacking objects: 100% (3/3), done.
From https://gitlab.com/jruela/java-queiro
65ebf90..29a5fca master -> origin/master
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the
result.
```

Caso isso ocorra, você tem 2 saídas: Abrir o(s) arquivo(s) com conflito e ajustar manualmente seu conteúdo ou fazer o(s) arquivo(s) com conflito “voltarem” a uma versão anterior.

Ajustar conflitos manualmente

Caso um ou mais arquivo seus tenham sofrido com conflito, ao abrir um deles, terá coisas um tanto estranhas em seu conteúdo. Vejamos um exemplo de como fica um arquivo desses a seguir.

```
# java-queiro
<<<<<< HEAD
se eu não te amasse tanto assim...
=====
se eu num te amasse um tanto assim...
>>>>>> 4e35b01632c66c0b655eb79a061f8e9724d47058
```

Confuso, não? E isso é um exemplo de um arquivo bem pequeno com uma linha de conflito. Se optou por ajustar manualmente, verifique como o conteúdo deve ficar e certifique-se de tirar TODAS as linhas com <<<<<< e >>>>>>, pois são marcações de conflito do Git.

Após fazer os ajustes necessários, execute o comandos...

```
git add .
git commit
```

Provavelmente você verá a tela do editor vim novamente (exemplo na Figura 7). Basta usar o **:x** para sair dela. É apenas uma mensagem informativa.


```
Merge branch 'master' of https://gitlab.com/jyoshiriro/java-queiro

# Conflicts:
#   README.md
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#   .git/MERGE_HEAD
# and try again.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
```

Figura 7: Tela do editor após um “git commit -a”

Resolvendo conflitos revertendo a versão

Caso quera simplesmente reverter a versão do arquivo para a de algum commit ou para a versão do repositório remoto, siga as instruções do tópico “**git checkout**” e depois execute os comandos...

```
git add .
git commit
```

Provavelmente você verá a tela do editor vim novamente (exemplo na Figura 6). Basta usar o **:x** para sair dela. A mensagem que exibida é uma mensagem padrão para o novo commit que está sendo realizado.

git pull - Obtendo as atualizações do repositório remoto

Uma vez que seu repositório já está clonado, sempre que quiser obter as alterações do repositório remoto em seu repositório local, use o comando...

```
git pull
```

Com ele, todas as alterações que eventualmente não estejam registradas em seu repositório local passarão a existir. Esse comando também sincroniza uma série de configurações do repositório remoto com o local.

git fetch

Verificando se há alterações no repositório remoto

Caso você deseje saber se existem alterações no repositório remoto que você ainda não recuperou via **git pull**, basta executar estes 2 comandos...

```
git fetch
```

depois

```
git diff origin/master
```

A saída é no mesmo estilo que a que vimos para o **git diff**.

O **git fetch** faz download dos metadados da branch remota, mas não os baixa os commits para o repositório local. Ou seja você consegue buscar as diferenças em relação a branch atual, mas não altera nada nesse branch.

Aqui temos que entender a diferença entre **git fetch** e **git pull**. O **git pull** equivale a fazer o **git fetch** (download dos metadados) e o comando **git merge** (incorpora os commits com a branch local).

Tutorial: Inicializando e publicando um repositório

Neste tutorial, vamos criar uma nova pasta em nosso sistema de arquivos, inicializar essa pasta como um repositório, trabalhar com arquivos e fazer commit das alterações que realizadas.

Por fim, vamos enviar essas alterações para um repositório remoto no Gitlab.

1. Crie uma pasta de projeto

Vamos criar uma nova pasta na área de trabalho. Para isso, abra o Git Bash. A pasta inicial é a pasta home do seu usuário, ou seja, **/c/Users/SeuUsuario**. (podemos consultar o caminho usando comando **pwd**).

Dica: A pasta home do usuário é representada pelo símbolo **~** em sistemas Unix.

Para acessar o Desktop, usamos o comando...

```
cd Desktop
```

Note que utilizamos a letra **D** maiúscula, pois os sistemas Unix são **case-sensitive**.

O terminal exibirá o seguinte caminho:

```
jruela@DESKTOP-CF2B6KT MINGW64 ~/Desktop  
$
```

Agora vamos criar uma nova pasta do projeto usando o comando...

```
mkdir meu-projeto
```

e entrar na pasta...

```
cd meu-projeto
```

Seu terminal irá exibir um caminho similar a esse:

```
jruela@DESKTOP-CF2B6KT MINGW64 ~/Desktop/meu-projeto  
$
```

2. Inicialize o repositório

Para que o Git passe a gerenciar o controle de versão dos arquivos dessa pasta, basta usar o comando...

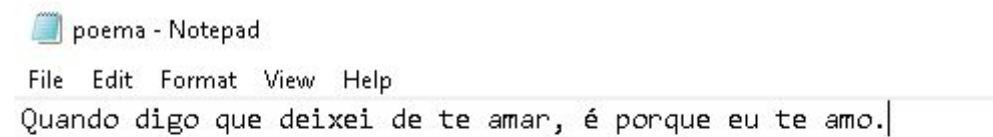
```
git init
```

O resultado do comando será...

```
Initialized empty Git repository in  
C:/Users/jruela/Desktop/meu-projeto/.git/
```

3. Adicione um arquivo

Agora, vamos ver como o Git trabalha com alterações feitas no diretório de trabalho. Para isso, abra o bloco de notas, adicione uma frase no arquivo e salve dentro da pasta que criamos...



Para consultar as alterações nosso repositório, use o comando...

```
git status
```

O resultado deve aparecer da seguinte forma:

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

Isso indica que o novo arquivo não foi adicionado a **área de staging**, ou seja, ele não será incluso em um novo commit.

Vamos adicionar esse arquivo a área de staging usando o comando

```
git add poema.txt
```

Se executarmos o comando git status novamente, veremos o seguinte relatório:

```
On branch master

No commits yet

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

    new file:   poema.txt
```

4. Faça um commit

Com o novo arquivo e suas alterações adicionadas na área de staging, basta executar o seguinte comando para gerar um commit:

```
git commit -m "Adicionar a primeira frase da poesia"
```

O resultado desse comando será o seguinte:

```
[master (root-commit) f4cba5d] Adicionar a primeira frase da poesia
1 file changed, 1 insertion(+)
create mode 100644 poema.txt
```

Isso indica que o commit foi finalizado corretamente. Para consultarmos todo o histórico de commits, usamos o comando...

```
git log
```

O resultado do comando será o seguinte:

```
commit f4cba5df97b7bf91840c50a985cbcd990275f134 (HEAD ->
master)
Author: Jose Ruela <jruela@gmail.com>
Date:   Tue Mar 12 09:40:02 2019 -0700

    Adicionar a primeira frase da poesia
```

Note que cada commit mostra informações de autor e data, além de possuir um hash que o identifica de forma única.

É possível consultar as alterações registradas em cada commit usando o comando **git show**. No exemplo, o comando seria executado da seguinte forma:

```
git show f4cba5df97b7bf91840c50a985cbcd990275f134
```

E seu resultado seria:

```
commit f4cba5df97b7bf91840c50a985cbcd990275f134 (HEAD ->
master)

Author: Jose Ruela <jruela@gmail.com>

Date:   Tue Mar 12 09:40:02 2019 -0700

    Adicionar a primeira frase da poesia

diff --git a/poema.txt b/poema.txt
new file mode 100644
index 0000000..134ac70
--- /dev/null
+++ b/poema.txt
@@ -0,0 +1 @@
+Quando digo que deixei de te amar, é porque eu te amo
```

5. Crie um repositório no Gitlab

Vamos criar nosso primeiro repositório no GitLab (gitlab.com). É necessário criar uma conta de forma gratuita e confirmá-la em seu email. Feito isso, basta clicar no ícone **[+]** e escolher a opção **New Project**, como mostra a Figura 1.

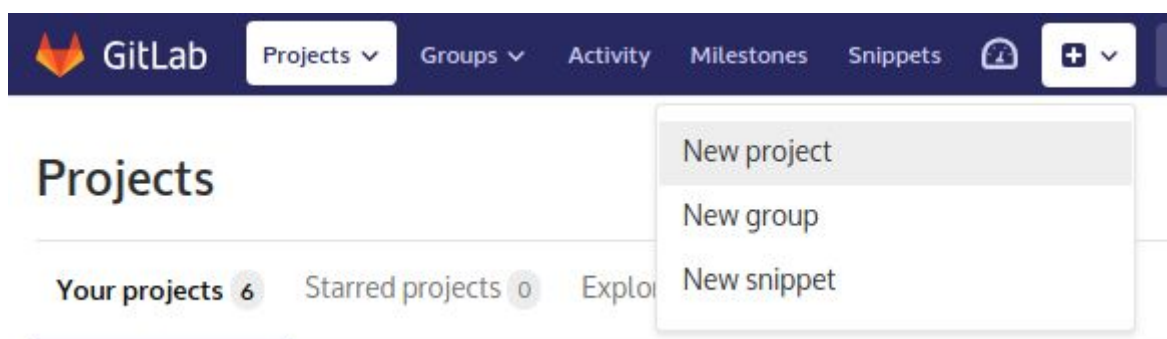
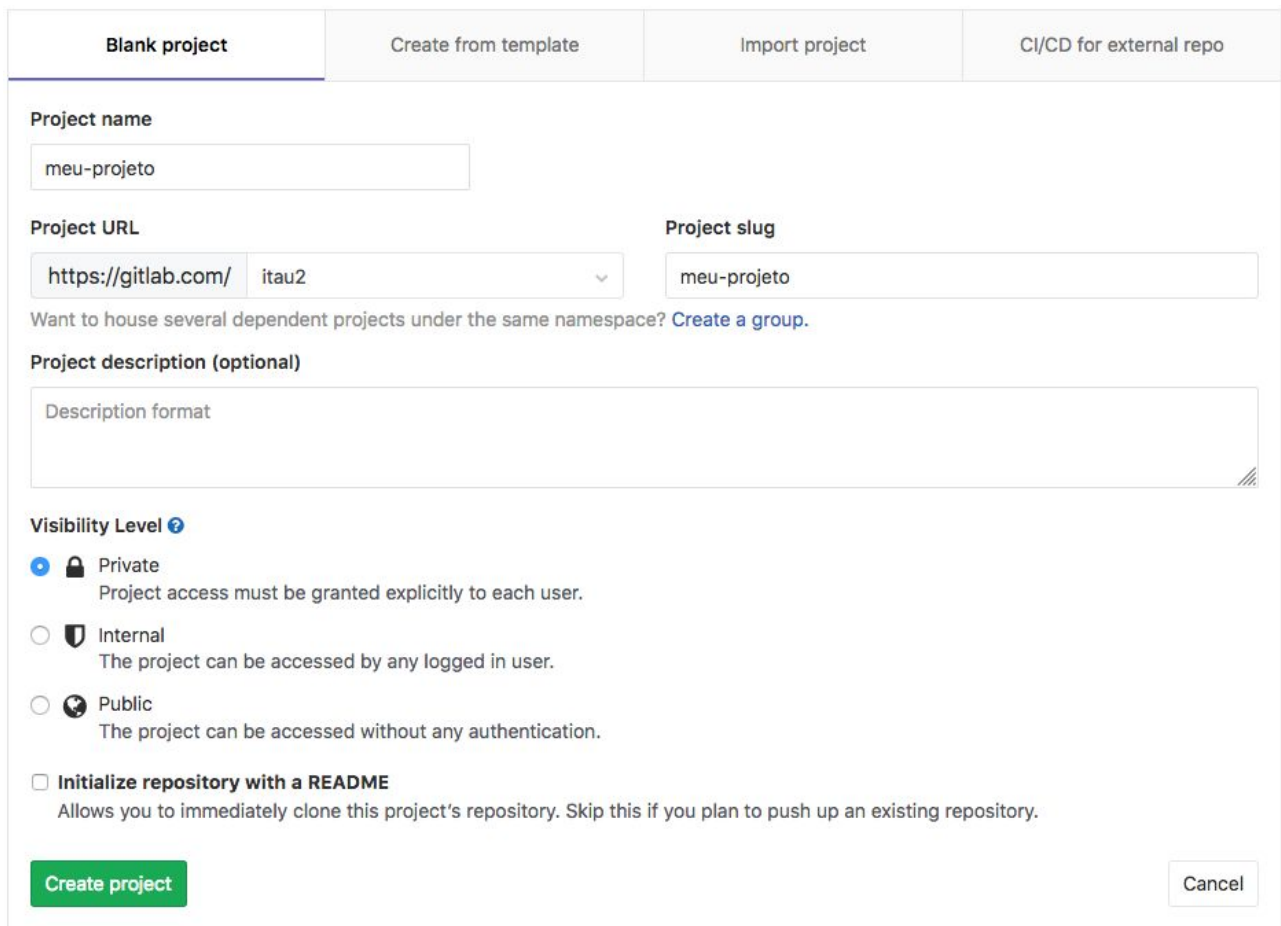


Figura 1: Solicitando a criação de um repositório Git no GitLab

Você será encaminhado ao formulário de criação de repositório. Nele, informe o nome do repositório (**Project name**), sua URI (**Project slug**) e, opcionalmente, sua descrição (**Project description**).

Defina ainda a visibilidade do projeto (**Visibility Level**), conforme as orientações (**Private** - Somente pessoas autorizadas terão acesso; **Internal** - Qualquer usuário autenticado poderá ver o projeto e **Public** - Qualquer pessoa poderá ver o projeto).

É possível ver um exemplo desse formulário na Figura 2.



Blank project Create from template Import project CI/CD for external repo

Project name
meu-projeto

Project URL Project slug
https://gitlab.com/ itau2 meu-projeto

Want to house several dependent projects under the same namespace? [Create a group.](#)

Project description (optional)
Description format

Visibility Level ?

☒ Private
Project access must be granted explicitly to each user.

☐ Internal
The project can be accessed by any logged in user.

☐ Public
The project can be accessed without any authentication.

☐ Initialize repository with a README
Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

Create project Cancel

Figura 2: Formulário de criação de um repositório Git no GitLab

Atenção: Não marque a opção **Initialize repository with a README**. Essa opção inicia o repositório já com um commit criado, de forma que vai impedir que possamos subir os commits que fizemos no repositório local.

Após clicar em **Create project**, já é possível ver o repositório, como mostra a Figura 3.

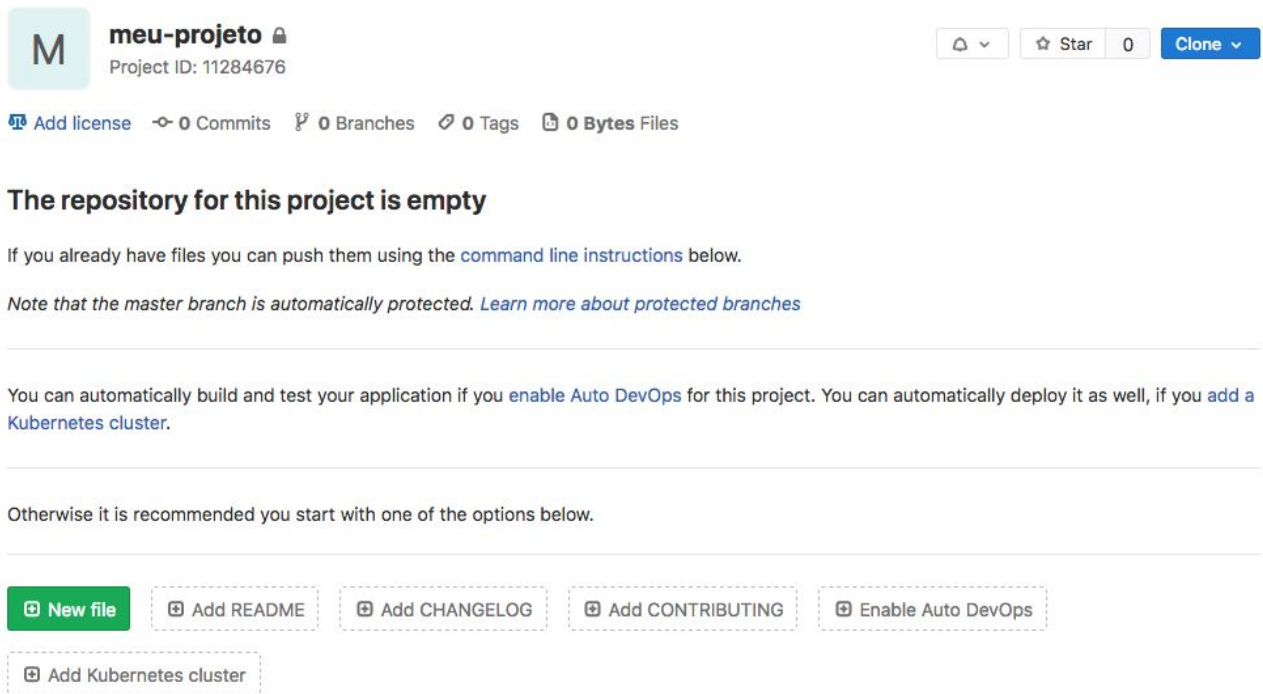


Figura 3: Repositório Git recém criado no GitLab

6. Envie os commits do repositório local para o repositório no Gitlab

Para enviar commits para o repositório no Gitlab, é necessário ter sua URL. Podemos copiar a URL através do barra de endereços do browser (desde que esteja na página inicial do repositório) ou através do botão **Clone**, conforme mostra a figura x.

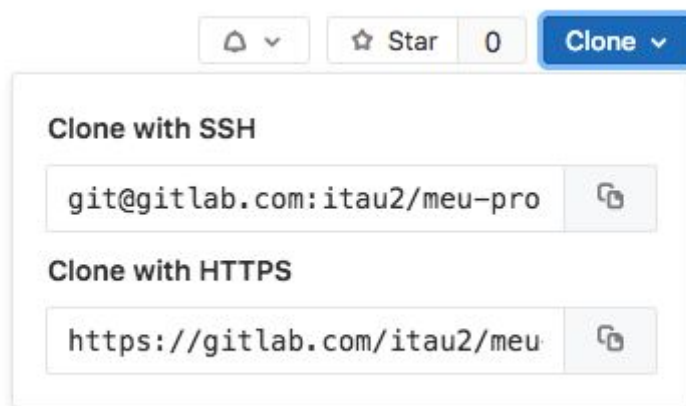


Figura x: Obtendo a URL de um repositório no Gitlab

Vamos usar o endereço HTTPS para fins deste tutorial. Com a URL copiada, basta usar o seguinte comando para enviar o commit...


```
git push https://gitlab.com/itau2/meu-projeto.git master
```

Coloque o nome de usuário e senha do Gitlab para que o comando seja executado corretamente. Após o envio dos dados, você poderá ver os commits publicados no Gitlab.


master

meu-projeto / +

History

Find file

Web IDE



Adicionar a primeira frase da poesia
Jose Ruela authored 2 hours ago

f4cba5df


Add README

Add CHANGELOG

Add CONTRIBUTING

Add Kubernetes cluster

Set up CI/CD

Name	Last commit	Last update
 poema.txt	Adicionar a primeira frase da poesia	2 hours ago

Note que fazer um push para um repositório vazio no Gitlab, estamos efetivamente inicializando ele com os metadados do repositório local. Dessa forma, o repositório remoto passa a atuar como um clone do local e é essa relação que permite enviar ou baixar commits entre eles. Não é possível enviar ou baixar commits de repositórios não relacionados, ou seja, que não tenham os mesmos metadados de inicialização.

Bibliografia

AQUILES, Alexandre; FERREIRA, Rodrigo. Controlando versões com Git e GitHub. São Paulo: Casa do Código, 2014.

Git. Git - Reference. Disponível em <<https://git-scm.com/book/pt-br/v2>>. Acesso em: 15 de fevereiro de 2018.