



Universidade Federal de Santa Maria
Centro de Tecnologia
Programa de Educação Tutorial - Sistemas de Informação

TUTORIAL



Iago da Cunha Corrêa
Cássio Castro Araujo
Alexandre Moreira Medina

Santa Maria, 2016
Sumário

- 1. Git**
 - 1.1. O que é controle de versões?**
 - 1.2. Uma breve história do Git**
- 2. Características do Git**
- 3. Instalação**
 - 3.1. Instalando Git no Linux**
 - 3.2. Instalando Git no Windows**
- 4. Primeiros passos**
 - 4.1. Configurações**
 - 4.2. Ajuda no Git**
- 5. Comandos do Git**
 - 5.1. Obtendo um Repositório**
 - 5.2. Gravando alterações**
 - 5.3. Histórico**
 - 5.4. Desfazendo**
 - 5.5. Repositórios Remoto**
 - 5.6. Criando Tags**
 - 5.7. Ramificações**
- 6. Conclusão**
- 7. Referências**

1. Git

1.1. O que é controle de versões?

O controle de versões pode ser descrito como uma sistemática para gerenciar as diferentes partes, versões e modificações no desenvolvimento de um documento qualquer, de forma eficiente, organizada e prática. Essa sistemática permite, mediante a necessidade do usuário, recuperar uma determinada versão deste documento. De forma tranquila e prática, pode-se dizer que é como registrar todas as modificações efetuadas nos arquivos durante o seu desenvolvimento.

Este tipo de sistema é comumente utilizado em projetos onde vários integrantes contribuem paralelamente com o seu desenvolvimento. Em projetos de softwares, o controle de versões pode ser utilizado para controlar o histórico de desenvolvimento dos códigos-fontes e também da documentação do mesmo.

Ao empregar o controle de versões, você estará garantindo segurança e praticidade para colaboração em seu projeto pois, ao modificar algum arquivo ou executar instruções comprometedoras no código-fonte de um software, você terá garantia de que poderá realizar uma reversão dos arquivos, e da mesma forma verificar as alterações que os demais integrantes do projeto fizeram.

1.2. Uma breve história do Git

O *Git* é um software para controle de versões criado por Linus Torvalds em 2005. A ideia de seu desenvolvimento surgiu quando Torvalds e os desenvolvedores do kernel Linux optaram por não utilizar mais o software proprietário BitKeeper, após Larry Macvoy (detentor dos direitos autorais do BitKeeper) remover o acesso gratuito ao software.

Torvalds tinha a intenção de desfrutar de um sistema distribuído que fosse rápido, fluído e que funcionasse de maneira similar ao BitKeeper. Sem muitas opções, Torvalds decidiu desenvolver o próprio software controlador de versões e assim nasceu o *Git*.

Em suma, o *Git* foi desenvolvido e projetado por Torvalds para auxiliar no desenvolvimento do kernel do Linux, porém, com a ênfase em velocidade e praticidade, aliadas à licença *GNU*, cujos termos declaram o software como livre, o *Git* acabou ganhando vários admiradores e usuários ao redor do mundo.

Atualmente o *Git* é mantido por Junio Hamano, um dos principais colaboradores do projeto em 2005 e, responsável por entregar a versão 1.0 do software em 21 de dezembro de 2005.

2. Características do Git

Nesse tópico será explicado como o Git trabalha, e vamos buscar entender também as principais características que o diferem dos outros tipos de controladores de versão.

2.1. Snapshots

A forma como o Git manuseia as informações se difere em muito dos outros controladores de versão. Ao invés de armazenar uma lista contendo as mudanças

efetuadas nos arquivos, como a maioria dos controladores, o Git registra “momentos”, os quais são chamados de snapshots, contendo uma espécie foto dos arquivos.

Cada commit que o usuário faz no seu diretório Git, é como se fosse feita uma captura de todos os arquivos presentes no diretório e criada uma referência para essas capturas. Visando manter a eficiência, caso um arquivo não tenha sido modificado, a sua snapshot não é atualizada, apenas a sua referência é atualizada.

2.2. Operações locais

Quase todas operações do Git são feitas de forma local. Desta forma, não existem, ao utilizar o Git, problemas que normalmente acontecem quando se usa um controlador de versão que depende muito de uma rede como, por exemplo, problemas no tráfego de arquivos ou latências de rede.

Como o Git opera em sua maioria localmente, quase que todas operações são efetuadas de forma instantânea.

2.3. Integridade

A integridade das informações de cada projeto são asseguradas por um checksum (chave de verificação de integridade). Dessa forma, o Git garante um item fundamental de sua filosofia, que trata da integridade dos seus arquivos. Com o checksum, o Git assegura que o usuário não irá perder informação em movimentação ou corromper arquivos sem que ele mesmo perceba.

Para realizar checksum, o Git usa uma implementação de hash SHA-1, que é basicamente uma string de 40 caracteres hexadecimais, calculada a partir de um arquivo ou estrutura do diretório Git.

2.4. Estados de operação

Quando se está atuando com um diretório Git, o mesmo obriga os arquivos a estarem em um dos 3 estados de operação: consolidado (committed), modificado (modified) ou preparado (staged).

Os dados se encontram committed quando já estão assegurados na base de dados do diretório Git. Se encontram modified, caso os arquivos tenham sido modificados e, no momento, se encontram diferentes do que está committed no diretório Git. Agora, o arquivo se encontra staged quando o usuário já terminou todas suas mudanças no arquivo e, lhe enviou pra área de staged para que as mudanças efetuadas sejam registradas como committed.

Para manter a consistência desses estados, o Git divide seu diretório em três seções que permanecem fundamentais para o projeto como um todo: o repositório Git, o diretório de trabalho e a área de preparação.

O repositório Git é onde se encontram todos os arquivos em sua forma committed, ou seja, todos os arquivos que servem como base para analisar o estado em que se encontram os arquivos no diretório do trabalho.

Como você já deve ter percebido, no diretório de trabalho situam-se todos os arquivos que são modificáveis. Estes arquivos são rotineiramente comparados com os arquivos correspondentes presentes no repositório Git para serem classificados como modified, se necessário.

Já a área de preparação serve para preparar os arquivos que foram modificados para serem enviados ao repositório e, assim, serem atualizados no repositório Git.

A Figura 1 demonstra de forma mais clara o funcionamento dos estados de operação do Git.

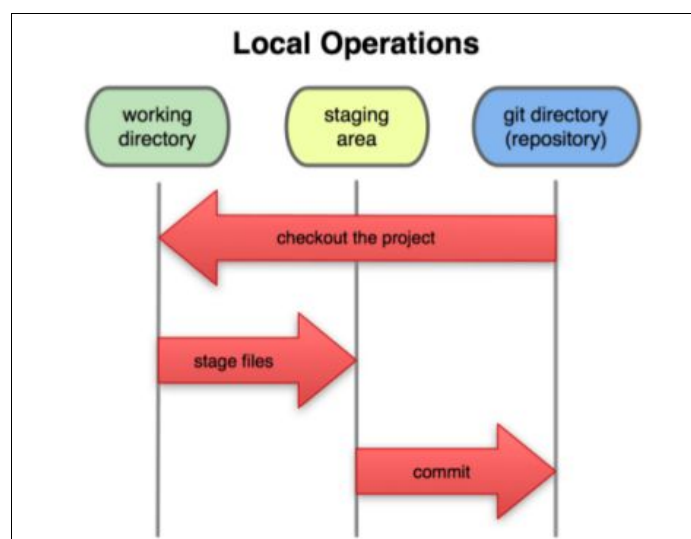


Figura 1. Demonstração dos estações operação do Git.

O fluxo de trabalho no Git funciona basicamente da seguinte maneira: os arquivos são modificados no diretório de trabalho, o usuário seleciona os arquivos que pretende consolidar no repositório Git e lhes envia para a área de preparação, e, por fim, executa um commit, o qual envia os arquivos presentes na área de preparação para o repositório Git.

3. Instalação

Veremos agora como instalar o Git nos sistemas operacionais Linux e Windows.

3.1. Instalando o Git no Linux

Para instalar o Git no Linux, é necessário executar um comando de instalação no terminal do seu SO.

O comando varia de distribuição para distribuição e vamos listar aqui os comandos que devem ser utilizados nas distribuições mais utilizadas.

Para Debian, Ubuntu e derivadas:

```
apt-get install git
```

Para Fedora:

```
yum install git (até Fedora 21)
```

```
dnf install git (Fedora 22 em diante)
```

Para openSuse:

```
zypper install git
```

Para Arch Linux:

```
pacman -S git
```

Pode ser necessário realizar um chaveamento para modo *root* antes de executar o comando de instalação. Caso não saiba como fazer, procure na documentação da distribuição que está utilizando.

3.2. Instalando o Git no Windows

Para instalar o Git no Windows, primeiramente você deve acessar o link <https://git-for-windows.github.io/> no seu navegador de preferência, mostrado na Figura 2. Assim, você deve estar no site da imagem e, então, precisa clicar em download.

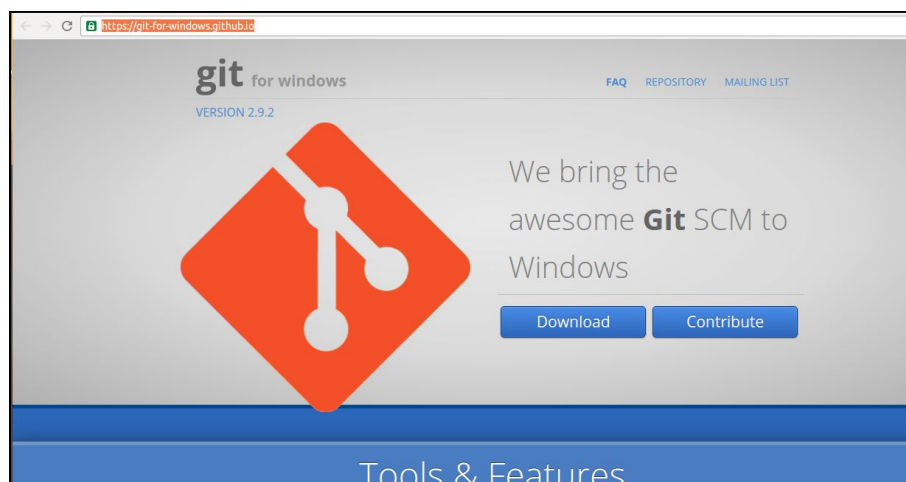


Figura 2. Site de instalação do Git para o windows.

Após clicar no botão de download, você será encaminhado para um site dentro do controlador de versões online *GitHub* onde você deverá deslocar-se para a parte inferior do site e baixar a versão estável mais atual no formato *.exe* (Figura 3).

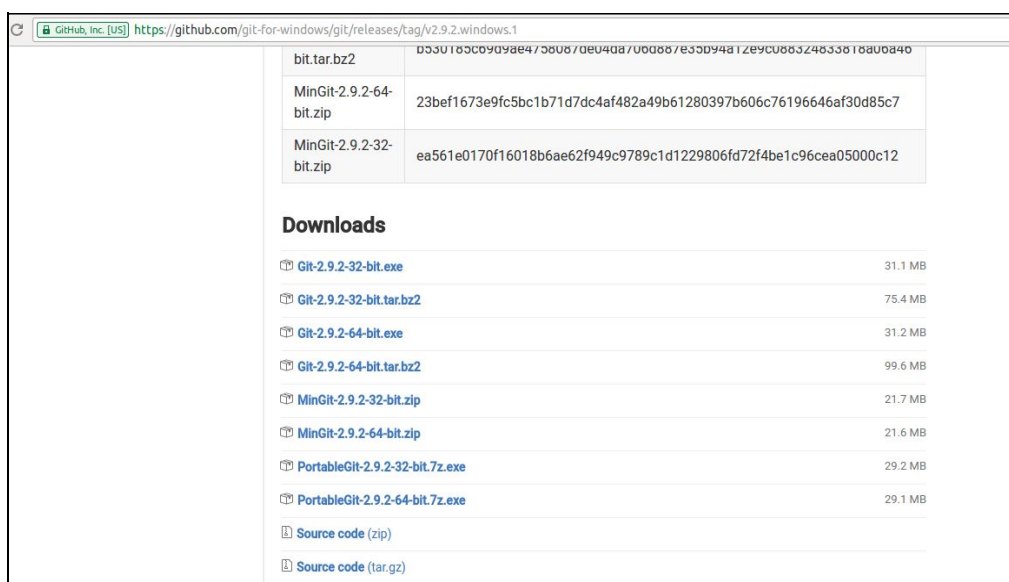


Figura 3. Baixar o .exe mais atual.

Terminando de baixar o instalador, é necessário apenas prosseguir com a instalação normal. O pacote conta com uma interface gráfica para usuários menos experientes, mas possui também uma interface básica que emula um shell.

4. Primeiros passos

Agora que temos a ferramenta Git devidamente instalada, podemos executar as configurações iniciais para início do trabalho.

4.1. Configurações

Uma vez que o Git esteja instalado no seu sistema, é necessário uma configuração prévia antes de começar a utilizar o software produtivamente.

O git config é a ferramenta do Git usada para configurações. Através dela você pode alterar configurações pré-definidas no momento em que desejar e toda configuração que você fizer será mantida através das atualizações.

As configurações podem ser mantidas em três arquivos e que são:

- `/etc/config`: Nesse arquivo, estão as configurações pertinentes a todos os usuários do sistema.
- `~/.gitconfig`: Configuração específica de cada usuário.
- `.git/config`: Configuração de cada diretório git.

As três configurações podem existir, porém apenas uma estará em vigor. O arquivo de configuração mais próximo do diretório `.git` sempre sobrescreve a configuração do git mais genérica, ou seja, caso exista um arquivo de configuração presente no diretório `.git`, tal arquivo terá prioridade sobre o arquivo de configuração presente em `/etc/gitconfig`.

Agora podemos identificar seu nome de usuário e e-mail para o git. Essas informações são importantes, pois cada commit efetuado por um usuário utiliza esses dados. Os comandos são os seguintes:

```
git config --global user.name "fulano"
git config --global user.email fulano@email.com
```

A diretiva `--global` é responsável por registrar as informações no `~/.gitconfig`. Se você desejar que essas informações sejam diferentes para projetos específicos, é necessário apenas digitar o mesmo comando sem a diretiva `--global` no diretório em que foi criado um projeto `.git`.

Para selecionar o editor de texto padrão do Git, como o Nano, por exemplo, é necessário entrar com o seguinte comando:

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

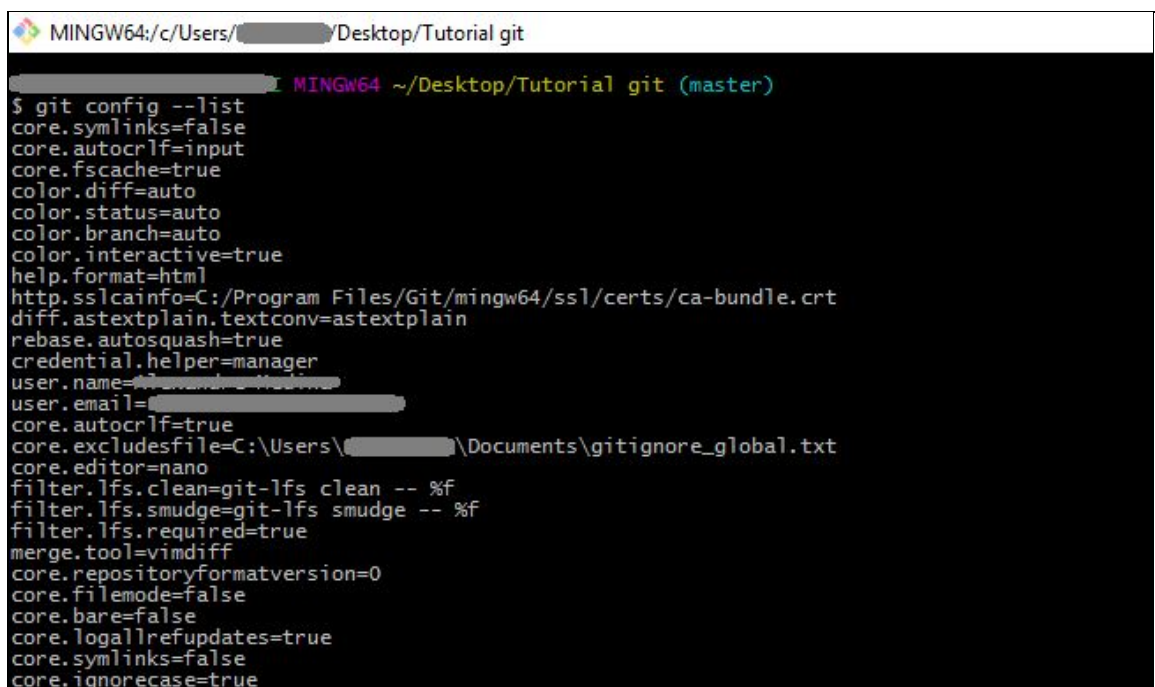
Para definir a seleção de ferramenta de Diff padrão, o comando é o seguinte:

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

Ferramentas de *Diff* são usadas para a solução de conflitos de merge dentro de um projeto.

Como exibe a Figura 4, é possível também listar todas as suas configurações com o comando:

```
git config --list
```



```

MINGW64: c:/Users/[redacted]/Desktop/Tutorial git
MINGW64 ~/Desktop/Tutorial git (master)
$ git config --list
core.symlinks=false
core.autocrlf=input
core.fscache=true
color.diff=auto
color.status=auto
color.branch=auto
color.interactive=true
help.format=html
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
diff.astextplain.textconv=astextplain
rebase.autosquash=true
credential.helper=manager
user.name=[redacted]
user.email=[redacted]
core.autocrlf=true
core.excludesfile=C:\Users\[redacted]\Documents\gitignore_global.txt
core.editor=nano
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.required=true
merge.tool=vimdiff
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
core.symlinks=false
core.ignorecase=true

```

Figura 4. Exibindo configurações do git.

4.3. Ajuda no Git

Caso você precise de ajuda com alguma ferramenta do Git, você pode usar as manpage (apenas no linux) através do comando:

```
man git
```

5. Comandos do Git

No presente capítulo, vamos aprender como trabalhar com o Git em si. Vamos aprender, em sequência, como criar um repositório Git, realizar commits para gravar alterações em arquivos, visualizar históricos de alterações, desfazer alterações recentes, criar Tags e ainda, trabalhar remotamente.

5.1. Obtendo um repositório

Conforme mostrado na Figura 5, para criar começar a realizar os registros do seu projeto, basta se deslocar via terminal até o diretório onde os arquivos do projeto estão localizados e entrar com o seguinte comando:

```
git init
```

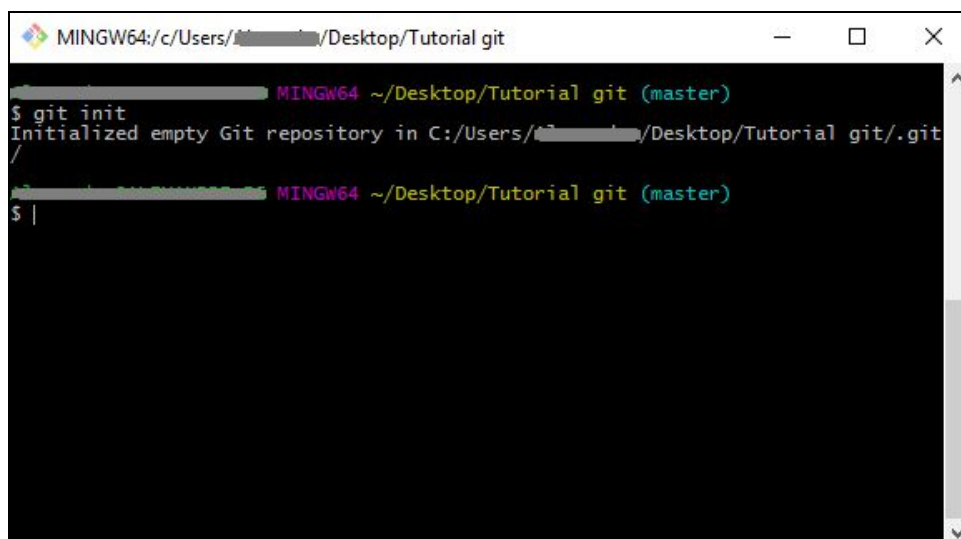


Figura 5. Inicializando repositório.

5.2. Gravando alterações

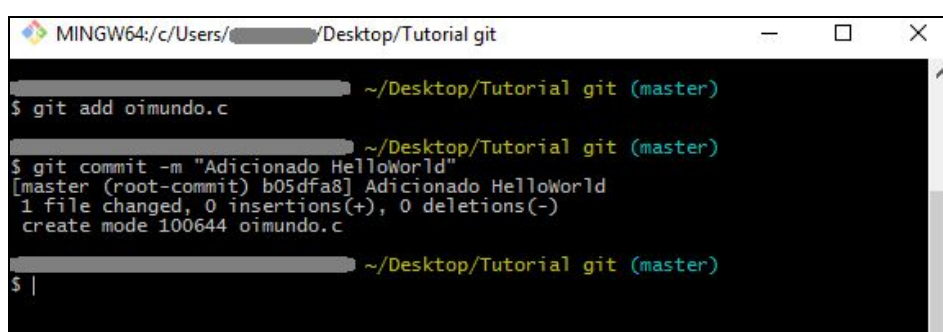
O comando da seção anterior irá criar um subdiretório .git contendo o esqueleto necessário para o funcionamento do repositório git. Nessa etapa ainda não existe nenhum arquivo registrado no repositório. Os arquivos são gravados no diretório .git a partir de commits efetuados pelo gerente do diretório quando for desejado.

Antes de efetuar um commit, é preciso que sejam especificados pelo usuário os arquivos que serão salvos ou consolidados (arquivos presentes em um commit), usando o comando:

```
git add oimundo.c
```

O comando acima especifica que o arquivo `example.c` irá ser usado em um commit posterior. Com o arquivo `oimundo.c` na lista de arquivos que serão consolidados e, sendo apenas ele necessário na mesma lista, só é preciso efetuar um commit como demonstra a Figura 6 com o comando:

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

A screenshot of a terminal window titled "MINGW64:/c/Users/[redacted]/Desktop/Tutorial git". The terminal shows the following commands and output:

```
$ git add oimundo.c
$ git commit -m "Adicionado HelloWorld"
[master (root-commit) b05dfa8] Adicionado HelloWorld
1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 oimundo.c
$ |
```

Figura 6. Adicionando e commitando um arquivo.

A diretiva `-m` é necessária para associar mensagem sobre o commit e usa a próxima entrada entre aspas como a mensagem do commit.

Agora você possui um diretório `.git` com um commit de um arquivo que você deseja monitorar. Com isso, é recomendado que, a cada mudança significativa no arquivo, todo o processo de commit seja feito para que as mudanças no arquivo sejam registradas no diretório `.git`.

É importante lembrar que os arquivos presentes no diretório podem estar em dois estados distintos: monitorado ou não monitorado. Os arquivos monitorados são os arquivos presentes no último commit, podendo estar inalterados, modificados ou selecionados para um outro commit. Os arquivos não monitorados são os que não foram consolidados e também não estão presentes em nenhuma seleção pré commit.

Conforme mostra a Figura 7, para visualizar o status dos seus arquivos, é usado o comando:

```
git status
```

```

MINGW64:/c/Users/[redacted]/Desktop/Tutorial git
MINGW64 ~/Desktop/Tutorial git (master)
$ git commit -m "Adicionado Hello World"
[master (root-commit) 4ea151a] Adicionado Hello World
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 oimundo.c

MINGW64 ~/Desktop/Tutorial git (master)
$ git add oimundo.c

MINGW64 ~/Desktop/Tutorial git (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   oimundo.c

MINGW64 ~/Desktop/Tutorial git (master)
$ git commit -m "Adicionado \\n ao HelloWorld"
[master d2d6180] Adicionado \\n ao HelloWorld
1 file changed, 6 insertions(+)

```

Figura 7. Utilizando git status.

O comando status tem como saída o estado em que se encontra cada arquivo modificado. Você pode efetuar o comando para um arquivo específico da seguinte maneira:

```
git status example.c
```

É possível observar as mudanças feitas em arquivos monitorados modificados através do comando:

```
git diff
```

Para remover arquivos presentes no seu repositório .git, é preciso removê-lo não só do diretório em que ele se encontra, mas também é removê-lo da área de seleção e então efetuar o commit.

A seguir, um exemplo:

```
git rm "arquivo destino"
```

5.3. Histórico

Para verificar o histórico dos commits efetuados em determinado repositório, existe o comando:

```
git log
```

Este comando, apresentado na Figura 8, mostra os commits efetuados de cima para baixo sendo o superior o mais recente. Um conjunto de informações do commit também são apresentadas como nome do autor, e-mail, data e a mensagem do commit.

Existem também algumas diretivas úteis que facilitam na hora de averiguar o histórico dos commits. A diretiva -p mostra, além do histórico básico do commit, as modificações realizadas nos arquivos.

A diretiva `--stat` mostra um conjunto de estatísticas úteis de cada commit, como por exemplo a quantidade de arquivos modificados e a quantidade de linhas modificadas.

```

MINGW64: c:/Users/.../Desktop/Tutorial git

$ git log
commit d2d61806f89c21e172c4a4d5414885e068188f4f
Author: ...
Date:   Fri Dec 9 15:06:36 2016 -0200

    Adicionado \n ao HelloWorld

commit 4ea151a63cc204bc4fc85e9b867630881062b68b
Author: ...
Date:   Fri Dec 9 15:05:36 2016 -0200

    Adicionado Hello World

$ git log -p
commit d2d61806f89c21e172c4a4d5414885e068188f4f
Author: ...
Date:   Fri Dec 9 15:06:36 2016 -0200

    Adicionado \n ao HelloWorld

diff --git a/oimundo.c b/oimundo.c
index e69de29..817d720 100644
--- a/oimundo.c
+++ b/oimundo.c
@@ -0,0 +1,6 @@
+#include <stdio.h>
+
+int main () {
+    printf("Olá mundo!\n");
+    return 0;
+}
\ No newline at end of file

commit 4ea151a63cc204bc4fc85e9b867630881062b68b
Author: ...
Date:   Fri Dec 9 15:05:36 2016 -0200

    Adicionado Hello World

diff --git a/oimundo.c b/oimundo.c
new file mode 100644
index 0000000..e69de29

```

Figura 8. Utilizando `git log`.

5.4. Desfazendo

Pode ser necessário desfazer alguma modificação que gerou um erro no projeto. Para poder refazer o último commit efetuado, é utilizada a diretiva `--amend` do comando `git commit` como no exemplo a seguir:

```
git commit --amend
```

Como resultado, esse comando irá abrir o mesmo editor de mensagens de commit usado pelo sistema e, caso não tenha sido efetuada nenhuma mudança nos arquivos desde o último commit, o git irá apenas reescrever a mensagem do commit.

Caso o gerente do arquivo tenha esquecido de efetuar alguma modificação antes de realizar o commit, ele pode realizar a modificação do arquivo, entrar com o comando do git `add` recebendo como parâmetro o arquivo modificado e, refazer o commit com a diretiva `--amend`.

5.5. Repositórios Remoto

Um repositório remoto do Git normalmente é hospedado da internet para que fique de fácil acesso para várias pessoas envolvidas em um projeto.

Quando você deseja compartilhar um diretório Git com outras pessoas, é necessário que você aprenda a gerenciar um repositório remoto do Git. Alguns dos conhecimentos necessários para conseguir gerir um repositório remoto são: adicionar um repositório remoto, remover repositórios remotos inválidos, gerenciar ramificações (explicado na seção 5.7).

Nesta seção iremos tratar sobre respeito de repositórios remotos e como trabalhar com outras pessoas em um projeto utilizando o Git.

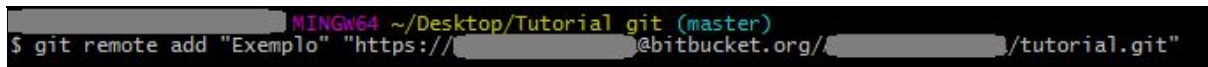
Vamos começar exibindo os remotos do seu repositório. Para isso entre com o comando:

```
git remote
```

É possível ainda, utilizar a diretiva “-v” para mostrar a url completa do seu remoto.

Para adicionar remotos, usa-se o comando “git remote add” como demonstrado na Figura 9 com a seguinte estrutura:

```
git remote add “apelido/nome curto” “url”
```

A screenshot of a terminal window with a black background. The prompt is '\$' and the command being entered is 'git remote add "Exemplo" "https://[redacted]@bitbucket.org/[redacted]/tutorial.git"'. The terminal title bar shows 'MINGW64 ~/Desktop/Tutorial git (master)'.

```
$ git remote add "Exemplo" "https://[redacted]@bitbucket.org/[redacted]/tutorial.git"
```

Figura 9. Adicionando apelido ao repositório.

Concedendo um apelido para um remoto, quando for necessário, é possível trazer ao repositório Git apenas os arquivos daquele remoto, usando o comando “fetch” seguido apenas do apelido do remoto.

```
git fetch “apelido”
```

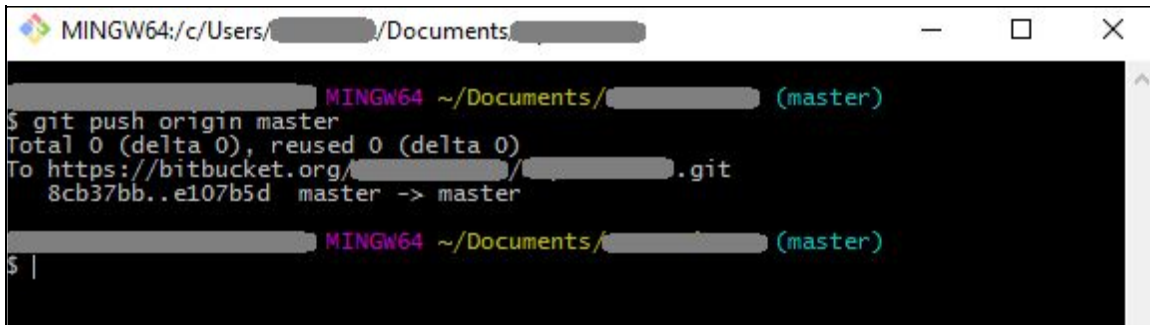
É importante lembrar que o comando fetch apenas traz ao repositório os arquivos solicitados, o merge com os arquivos atuais deve ser feito de forma manual, quando o mesmo estiver pronto.

Caso seja de interesse do usuário, pode-se usar o comando “pull” para trazer os arquivos, pois o mesmo já efetua merges automaticamente.

```
git pull “apelido”
```

Quando o seu trabalho está pronto e já lhe é de interesse enviar os arquivos modificados ao remoto do projeto, usa-se o comando “push” para efetuar a tarefa. É possível enviar o arquivo para o remoto de origem ou para um remoto em específico como mostra a Figura 10.

```
git push origin master
```

A screenshot of a Windows terminal window titled 'MINGW64: c:/Users/[redacted]/Documents/[redacted]'. The prompt is 'MINGW64 ~/Documents/[redacted] (master)'. The user enters '\$ git push origin master'. The output shows 'Total 0 (delta 0), reused 0 (delta 0)' and 'To https://bitbucket.org/[redacted]/[redacted].git', followed by a commit hash '8cb37bb..e107b5d' and 'master -> master'. The prompt returns to 'MINGW64 ~/Documents/[redacted] (master)' with a cursor on a new line.

```
MINGW64: c:/Users/[redacted]/Documents/[redacted]
MINGW64 ~/Documents/[redacted] (master)
$ git push origin master
Total 0 (delta 0), reused 0 (delta 0)
To https://bitbucket.org/[redacted]/[redacted].git
 8cb37bb..e107b5d  master -> master

MINGW64 ~/Documents/[redacted] (master)
$ |
```

Figura 10. Enviando arquivos para o remoto de origem.

Vale lembrar que, para poder escrever arquivos no repositório remoto, você necessita ter permissão de escrita.

Quando quiser ter informações sobre um remoto, use:

```
git remote show "nome remoto"
```

Para renomear um remoto, use:

```
git remote rename "nome antigo" "nome novo"
```

Para remover um remoto, use:

```
git remote rm "nome remoto"
```

5.6. Criando Tags

O Git também oferece a possibilidade de criar tags em pontos específicos na história do código como pontos importantes. Geralmente as pessoas utilizam essa funcionalidade para marcar pontos de update (v1.0, v1.1, etc).

Git possui dois tipos principais de tags: leve e anotada. Uma tag do tipo leve é semelhante a uma ramificação (será melhor detalhado na seção 5.7) que não muda, isto é, é um ponteiro para um commit específico. Diferente disso, tags anotadas são armazenadas como objetos inteiros no BD do Git. Estas possuem uma chave de verificação e outros dados importantes para identificação, tais quais, nome, criador da tag, email, etc.

Para criar uma tag anotada é simples, como pode ser visto na Figura 11, basta adicionar o "-a" ao executar o comando tag:

```
git tag -a v1.0
```

Para verificar o que contém na tag basta utilizar o seguinte comando:

```
git show v1.0
```

```

MINGW32: D:/Personal/Git Demo/local clone
$ git tag
$ git tag -a v1.0 -m 'First Verion Tag'
$ git show v1.0
tag v1.0
Tagger: Ashutosh Meher <ashuu.online@gmail.com>
Date:   Fri Sep 4 14:09:06 2015 +0530

First Verion Tag

commit cf277a93e852516815d1c034fecd326f7984a088
Author: Ashutosh Meher <ashuu.online@gmail.com>
Date:   Fri Sep 4 14:08:22 2015 +0530

    First Update

diff --git a/myproject/Doc2.txt b/myproject/Doc2.txt
index bb3303e..e431fd7 100644
--- a/myproject/Doc2.txt
+++ b/myproject/Doc2.txt
@@ -1,8 +1,4 @@
 doc 2
 doc 22
 doc 222
- doc 2222
- doc 22222
- doc 222222
- doc 2222222
- doc 22222222
\ No newline at end of file
+ doc 2222
\ No newline at end of file
<END>
<END>
$

```

Figura 11. Criando Tag Anotada

5.7. Ramificações

O GIT não armazena seus dados de versionamento como uma série de conjuntos de mudanças ou diferenças, ao invés disso, ele utiliza o conceito de snapshots. Ao realizar uma consolidação (commit) o GIT armazena um objeto que contém um ponteiro para um snapshot do conteúdo, autor e metadados e 0 ou, se existirem, um número referente à ponteiros para os commits que são pais desse novo commit.

Uma ramificação, no Git, é simplesmente um ponteiro móvel para um desses commits. Master é o nome padrão utilizado no Git. Quando fizemos commits a ramificação principal (branch master) que aponta para o último commit feito pelo usuário avança automaticamente.

Em suma, se você criar uma nova ramificação, o Git cria um novo ponteiro para que você possa se mover. Por exemplo, criando uma ramificação chamado teste:

```
git branch teste
```

Esse comando criará um novo ponteiro para o mesmo commit que você está no momento.

Para mudar para uma ramificação já existente, usamos ao invés de “branch” a palavra “checkout” como demonstra a Figura 12.

```
git checkout teste
```

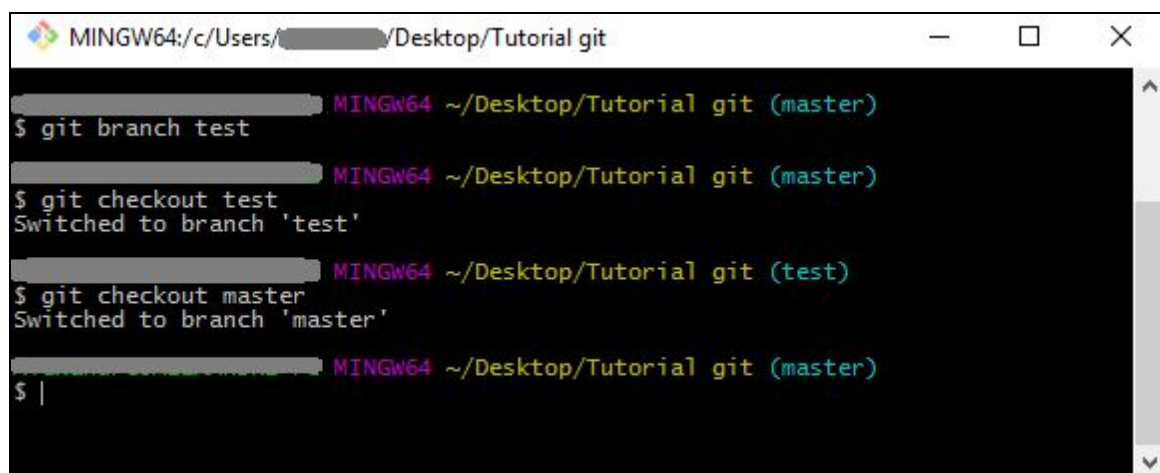

O comando acima faz com que o HEAD aponte para a ramificação teste.

Ao efetuarmos uma nova alteração, ou seja, um novo commit, o ponteiro HEAD acompanha a ramificação teste, porém o master segue no commit anterior, a menos que seja efetuado o comando:

```
git checkout master
```

Com o comando acima, o HEAD aponta novamente para o master e reverte os arquivos em seu diretório de trabalho para o estado em que estavam no snapshot para onde o master apontava. Em suma, ele volta ao trabalho antes de criar a ramificação teste, mas sem apagar o que foi feito na ramificação teste, para que seja possível criar uma “ramificação” no código.

Se criarmos um novo commit, o head acompanha o master apontando para esse novo commit.

A screenshot of a Windows terminal window titled 'MINGW64: c:/Users/[redacted]/Desktop/Tutorial git'. The terminal shows a sequence of Git commands and their outputs. The prompt is '\$' and the current directory is '~/Desktop/Tutorial git'. The commands and outputs are: 1. '\$ git branch test' followed by 'MINGW64 ~/Desktop/Tutorial git (master)'; 2. '\$ git checkout test' followed by 'Switched to branch 'test''; 3. '\$ git checkout master' followed by 'Switched to branch 'master''; 4. The prompt '\$ |' is shown at the end. The terminal has a dark background with light-colored text.

```
MINGW64: c:/Users/[redacted]/Desktop/Tutorial git
$ git branch test
MINGW64 ~/Desktop/Tutorial git (master)
$ git checkout test
Switched to branch 'test'
MINGW64 ~/Desktop/Tutorial git (test)
$ git checkout master
Switched to branch 'master'
MINGW64 ~/Desktop/Tutorial git (master)
$ |
```

Figura 12. Criando nova ramificação e alternando entre elas.

6. Conclusão

Através do presente guia, pode-se afirmar que o Git é uma ferramenta versátil, poderosa e que se utilizada de maneira correta, auxilia muito em projetos individuais. Podemos voltar a versões anteriores do código, caso ocorra algum bug com a atualização do mesmo, é possível, também, comentar o andamento do projeto, facilitando a visualização do progresso de cada commit. No caso de serem envolvidas equipes de trabalhos, além do que ocorre em projetos individuais, pode ocorrer de vários funcionários do projeto poderem trabalhar quase que de maneira simultânea sem prejudicar o andamento do projeto

7. Referências

Scott Chacon and Ben Straub, **Git book** - <https://git-scm.com/book/pt-br/v1/> - Acessado em Setembro de 2016

Roger Dudler, **Git - guia prático** -

http://rogerdudler.github.io/git-guide/index.pt_BR.html - Acessado em Setembro de 2016

Treehouse, **Why You Should Switch from Subversion to Git** -

<http://thinkvitamin.com/code/why-you-should-switch-from-subversion-to-git/> - Acessado em Outubro de 2016

André Felipe Dias, **Vantagens e Desvantagens do Controle de Versão Distribuído** -

<https://blog.pronus.io/posts/vantagens-e-desvantagens-do-controle-de-versao-distribuido/> - Acessado em Outubro de 2016

Diego Eis, **Iniciando no Git** - <http://tableless.com.br/iniciando-no-git-parte-1/> - Acessado em Outubro de 2016

Atlassian, **Become a Git Guru** - <https://www.atlassian.com/git/tutorials/> - Acessado em Outubro de 2016