



Escola piloto de computação

Apostila git para iniciantes



**Mossoró - RN
2020**

Sumário

Introdução	4
Teoria	5
O que é um controle de versão?	5
O que é o git?	7
Dinâmica de trabalho do git	8
Controle de diretórios :	8
Working directory	8
Stage area	9
Git directory	9
Controle de branch:	9
Git Bash:	11
Primeiros passos com o Git	12
Configurando o ambiente:	12
Criando um projeto:	12
git init	12
Gerenciando arquivos no working directory:	13
git status	13
git add	14
git diff	15
git commit	16
git show	17
Controle de versão e marcadores:	17
git log	17
git tag	18
git branch	19
git checkout	20
git merge	20
git reset	22
Repositórios Remotos	23
Configurando o repositório remoto no github e gitlab:	23
Gerenciamento do repositório remoto :	24
git clone	24
git push	25
git remote	26

git pull	26
git fetch	27
Bibliografia	29

Introdução

O git é um ferramenta de controle de versão desenvolvida pelo **Linus Torvalds** criador do linux, ela foi desenvolvida quando Linus estava criando o kernel do linux e necessitava de um controle de versão para gerenciar o seu projeto, como solução foi desenvolvido um controle de versão distribuído outrora chamado de git. Atualmente o git é um dos sistema de controle de versão mais utilizados no mundo, ele tem uma dinâmica de trabalho única que faz com que se diferencie dos demais controles de versão do mercado, sendo que por ser distribuído agrega os mais diversos repositórios web como o github e gitlab.

Teoria

O que é um controle de versão?

Quando se desenvolve um projeto de magnitude grande, média ou até mesmo pequena, surge em meio ao desenvolvimento a necessidade de se ter um controle de versão do projeto. Sobrescrever os arquivos de um projeto para cada nova versão é algo ineficiente e pouco seguro, e de mesma forma fazer manualmente o controle de vários arquivos de diferentes versões não é algo trivial, pois dependendo do projeto pode ficar algo complexo de gerenciar e manter. Desta forma um controle de versão é uma ferramenta que soluciona o gerenciamento das versões do projeto, ela organiza as versões de um projeto de forma que seja possível recuperar uma versão antiga, saber o que foi alterado em cada versão, entre outras coisas.

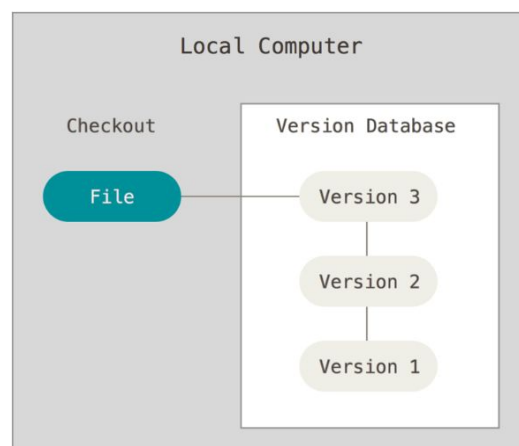


Figura 1 - Controle de versão local.
Fonte: Pro Git, Scott Chacon, Ben Straub

Os sistemas de controle de versões (VCSs) são basicamente bancos de dados que armazenam os backups das versões de um projeto em uma máquina local. OS VCSs foram os primeiros sistemas de controle de versões criados, sua arquitetura é extremamente simples e trivial, podendo ser considerado um banco de dados com arquitetura delta. Naturalmente quando os projetos começaram a ficar maiores e mais complexos, demandando uma equipe de desenvolvimento e várias máquinas locais, os programadores começaram a utilizar sistemas em rede para distribuir o projeto, a dinâmica de trabalho mudou e necessitava de um novo paradigma de controle de versão. O problema essencial em utilizar um simples sistema como um banco de dados em rede é que pode existir incompatibilidade de trabalhos, espera ocupada por acesso a recursos e comprometimento da integridade dos dados de um projeto. Desta forma um novo sistema de controle de versão denominado

sistemas centralizados de controle de versão (CVCSs) surgiu para gerenciar um projeto em rede.

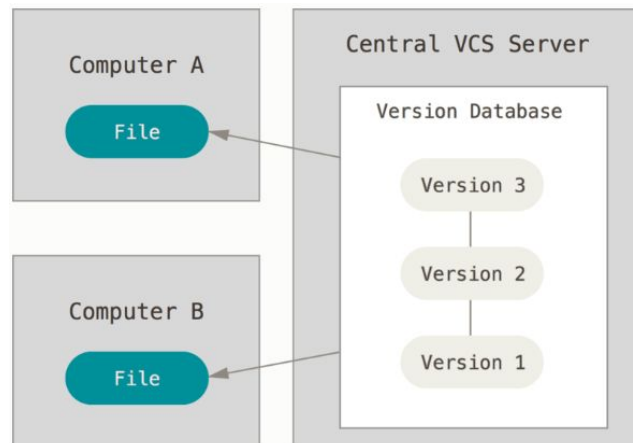


Figura 2 - Controle de versão centralizado.
Fonte: Pro Git, Scott Chacon, Ben Straub

Os CVCSs são sistemas baseados no VCS, a existência de uma rede central em sua arquitetura é a principal diferença entre um VCS comum e o CVCS. A rede central é responsável por armazenar as versões do projeto e se comunica com várias máquinas locais, quando uma máquina local deseja acessar o projeto, a rede distribui a versão mais atual do projeto, dando também aos gerentes de projeto uma visão do que todos os que trabalham no projeto fizeram nas sucessivas versões. Por mais que o CVCS solucione a dinâmica de trabalho em rede, ainda existem bastantes problemas estruturais, por exemplo um problema bastante óbvio é que se o servidor ficar inativo ninguém pode trabalhar no projeto. A dinâmica de trabalho com CVCs exige que os programadores tenham acesso a internet e que o servidor esteja ativo, em uma eventual parada para manutenção todo o sistema fica inativo e o projeto inacessível, pensando nesses problemas surge o Sistemas de Controle de Versão Distribuído (DVCSs) no qual os dados do servidor é copiados para a máquina local do programador.

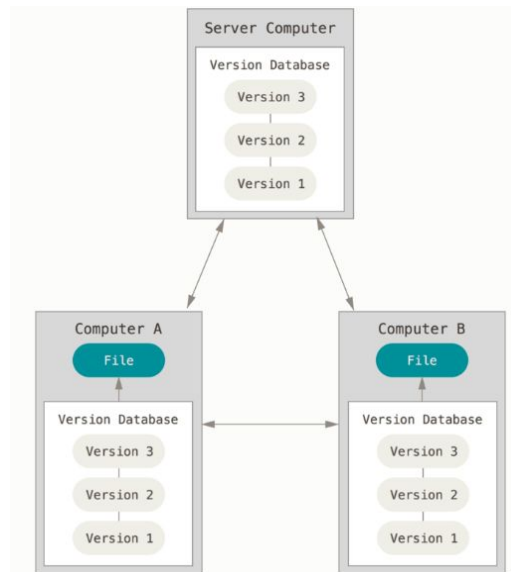


Figura 3 - Controle de versão distribuído.
Fonte: Pro Git, Scott Chacon, Ben Straub

Os DVCSs funcionam então como se fossem uma mistura de um VCS e um CVSS. No DVCS existe um servidor que armazena o repositório central do projeto, nesse repositório está armazenado os arquivos e diferentes versões do projeto, quando uma máquina local acessa o servidor ele copia os arquivos existentes no repositório do servidor fazendo assim uma cópia para a máquina local, isso significa que mesmo que servidor fique inativo ou tenha seus dados corrompidos o repositório local situado na máquina do usuário tem garantia de estar inalterável. O projeto local também pode ser utilizado para reparo dos dados do repositório caso haja um eventual problema no servidor.

O que é o git?

Como já citado o Git é um sistema de controle de versão distribuído (DVCSs) desenvolvido por **Linus Benedict Torvalds** (criador do linux) quando estava desenvolvendo o kernel do linux. O git é atualmente uma das ferramentas de controle de versão mais usada em todo o mundo, ela possui integração com vários sistemas de repositórios web remotos como github e gitlab.

Essencialmente a principal diferencial do git é a forma como ele trabalha com os arquivos do seu projeto, diferentemente de outras ferramentas de controle de versão que armazenam várias versões de um mesmo arquivo, o git armazena um instantâneo do arquivo, ou seja, em vez do git armazenar as alterações de um arquivo para outro como é o caso dos sistemas baseados em delta, o git armazena uma imagem atual do arquivo denominado de instantâneo.

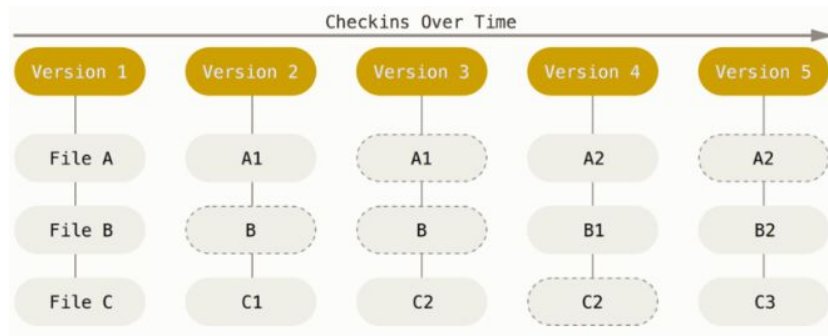


Figura 4 -Armazenando dados como capturas instantâneas do projeto ao longo do tempo.

Fonte: Pro Git, Scott Chacon, Ben Straub

Outra diferença essencial entre o git e outros SVCs é que quase todas as operações do git acontecem de forma local, ou seja, a presença do servidor não é obrigatória para que o git funcione. Vale citar também que a forma como o git ver as versões de um projeto é um tanto única, ele ver algo como isso 24b9da6552252987aa493b52f8696cd6d3b00373 que nada mais é que uma soma de verificação em um mecanismo chamado hash SHA-1, o funcionamento deste entretanto não é o foco deste estudo, contudo ele será visto quando você for recuperar versões ou exibir o arquivo de log a partir do comando git log.

Por fim a última diferença é a dinâmica de trabalho do git, seus arquivos são gerenciados por 3 áreas de trabalho denominadas de working directory, stage área e git directory, mais a frente iremos abordar cada um de forma detalhada.

Dinâmica de trabalho do git

Controle de diretórios :

O git tem uma dinâmica de funcionamento baseada em diretórios de trabalhos e em branches de trabalho. O ambiente de trabalho do Git é dividido em três como já mencionado, quando um arquivo é adicionado em um projeto este basicamente pode está em três tipos de estágios: alterado, preparado e salvo, estes estágios são referentes a qual camada de trabalho os arquivos estão situados, essas camadas são as seguintes : working directory, stage área e git directory.

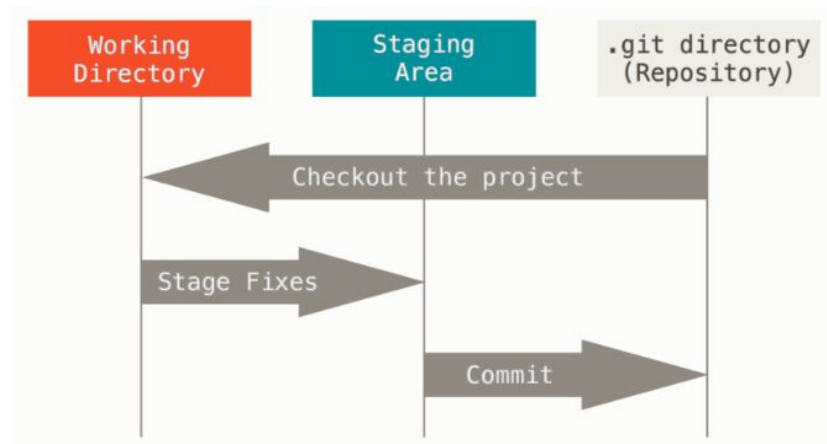


Figura 5 -Camadas de trabalho do git.
Fonte: Pro Git, Scott Chacon, Ben Straub

1. Working directory

Esta é a camada mais abaixo na hierarquia de camadas, ela é basicamente o local real na máquina onde os seus arquivos se encontram, por exemplo suponha que você esteja desenvolvendo um projeto que tem atualmente 2 arquivos, se estes não estiverem salvos em qualquer outra camada estes se mantêm na camada working directory (diretório de trabalho).

Quando criado um projeto git com o comando "git init" o git faz uma soma de verificação inicial gerando uma hash SHA-1 quando o arquivo é alterado o git sabe que ele foi alterado porque esse código hash SHA-1 é alterado, logo, mesmo que os arquivos não estejam salvos ou tenham suas alterações notificadas, o git sabe que eles foram alterados e que não estão salvos em suas camadas superiores.

2. Stage area

Esta é a segunda camada do git e nela estão os arquivos que estão no estado preparado, basicamente quando você quiser salvar vários arquivos de uma só vez você precisa os preparar nesta camada adicionando com o comando "git add". Quando tiver com todos os arquivos, e deseja realizar o seu commit, basta fazê-lo que o "git commit", este irá pegar todos os arquivos da área preparada e irá colocá-los na camada do git directory criando assim uma versão de projeto.

3. Git directory

Esta é a área do banco de dados local do git, basicamente se seus arquivos se encontram nesta área significa que eles estão salvos no sistema de arquivo do git, sendo gerada uma chave que pode futuramente ser utilizada para recuperação da versão posteriormente (checkout).

Controle de branch:

O git atua sobre um princípio de branches, significa que tudo o que for feito e guardado será sobre uma branch. Para facilitar um pouco as coisas entenda a branch como um ramo de desenvolvimento do projeto, ou seja, todo projeto está sobre uma branch denominada (master) ela é a chamada branch principal.

Na branch principal é possível criar várias branches distintas, essas branches possuem suas próprias áreas de espera e seus próprios git directory, ou seja, salvar um arquivo em uma branch não interfere em outra branch, sendo assim branches não se comunicam a não ser que você tente fazer um merge que é uma ação de tentar unir duas branches distintas. Isso ocorre porque o propósito fundamental de uma branch é distribuir tarefas e testar coisas novas como ferramentas por exemplo, o fato é que uma hora ou outra esse ramo vai voltar a fonte que é a master a partir de um merge.

Ambiente de trabalho

O ambiente de trabalho que será utilizado é o próprio git bash disponível no site do git (<https://gitforwindows.org/>). Vale lembrar que para os usuários de linux o git já vem pré instalado nativamente então não existe a necessidade de instalar, caso o mesmo não venha basta utilizar o comando `sudo apt-get install git-all`.

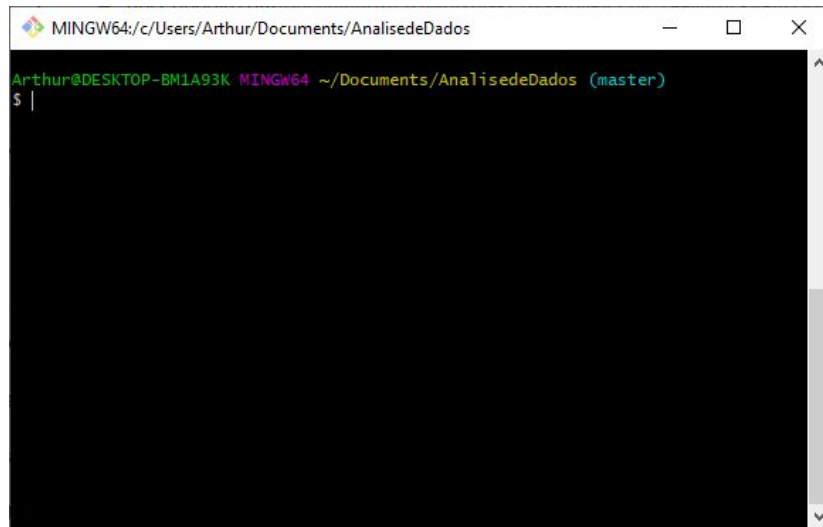


Figura 4 - terminal do git bash.

Fonte: próprio do autor.

A interface gráfica proposta a ser utilizada é a própria interface gráfica nativa do git, ela pode ser acessada através do comando `"gitk"` digitado no terminal do git bash. Vale lembrar que para ela ser acessada na pasta corrente deve haver um projeto do git, mais adiante será mostrado como se criar um projeto via terminal git bash.

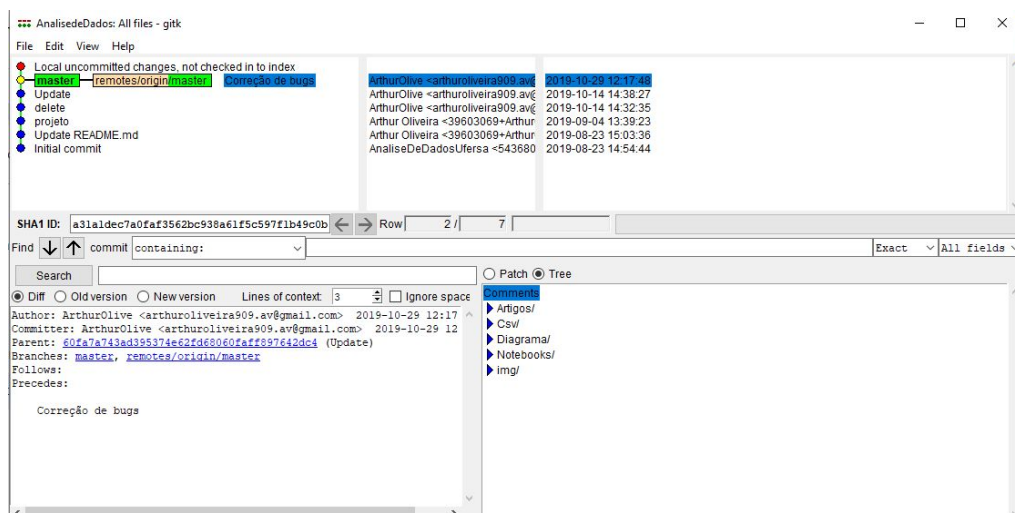


Figura 5 - Interface gráfica nativa do git

Git Bash:

O terminal do git Bash é baseado no terminal do linux, assim os comandos do git bash são idênticos aos do linux. Se voce não tiver familiariedade com os comandos simples como (ls e cd) e como o foco do curso não é ensinar comandos no prompt de comando do linux, é recomendado que seja visto um pouco sobre, abaixo segue um link de o blog do devmedia (<https://www.devmedia.com.br/comandos-importantes-linux/23893>).

Enfim começando pelo git bash, quando selecionado um projeto iniciado temos em tela as seguintes marcações:

1. Nome do usuário.
2. Nome da máquina.
3. Sistema operacional.
4. Diretório.
5. Branch.

O nome do usuário e o nome da máquina está na seção marcada pela cor verde como é possível ver na "**Figura 4**", essa seção contém a assinatura da máquina que o projeto se encontra, ou seja, no meu caso o meu projeto está na máquina cujo usuário logado é "Arthur" e o nome da máquina é "DESKTOP-BM1A93K". Esse nome em verde é utilizado para identificar por exemplo ações sobre o projeto, como um commit por exemplo.

O trecho marcado de rosa diz respeito ao sistema operacional que se encontra o projeto, no meu caso está no windows x64 denominado por MINGW64. Em amarelo está descrito o diretório atual do projeto separado por '/' que é o separador padrão do windows, este diretório em amarelo é o caminho absoluto do local do projeto git, seguido pelo nome em azul que representa a branch atual do projeto, que neste caso da figura 4 é a branch padrão denominada de master.

Todos os comandos utilizados no sobre um projeto git vem precedidos do nome git, ou seja, para identificar se um comando é referente ao prompt ou ao git basta observar se este inicializa com o nome git precedido de argumento.

Primeiros passos com o Git

Configurando o ambiente:

Assim que o git bash é instalado, é necessário realizar uma configuração inicial de alguns metadados que podem interferir na utilização do projeto, como nome do usuário e email. Para configurar basta abrir o git bash em qualquer diretório e digitar o comando `"git config --global user.name 'Seu_Nome'"` e `"git config --global user.email 'Seu_Email'"`, após realizar a alteração no nome do usuário, para verificar se realmente está tudo ok basta digitar o comando `"git config --global --list"`.

Criando um projeto:

- **git init**

Para criar um projeto novo no git com o git bash se utiliza o comando `"git init"`, este por sua vez cria uma estrutura inicial do git na pasta atual onde o git bash está localizado. inicialmente ele fará uma soma de verificação e saberá todos os arquivos presentes no diretório onde o projeto foi inicializado, e criará uma branch denominada master, que como já citado é branch padrão do git.

A terminal window with a black background and green text. The prompt is 'Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git'. The command '\$ git init' is entered, followed by the output 'Initialized empty Git repository in C:/Users/Arthur/Documents/git_projetos/exemplo_git/.git/'. The prompt then changes to 'Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)' and a new '\$ |' prompt is shown.

```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git
$ git init
Initialized empty Git repository in C:/Users/Arthur/Documents/git_projetos/exemplo_git/.git/
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ |
```

Figura 6 - Criando um projeto no git bash.

Fonte: próprio do autor.

Após inicializar o projeto note que se houver algum arquivo no presente diretório, este estará no seu diretório de trabalho (working directory) e deverá estar no estado "alterado". Para exemplo didático dentro desse diretório existe um arquivo chamado `"meu_primeiro_arquivo_git.txt"` nele contém o famoso "hello world". Abaixo segue todos os arquivos do meu diretório do projeto.

```

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ ls -la
total 9
drwxr-xr-x 1 Arthur 197121  0 mar  2 12:12 ./
drwxr-xr-x 1 Arthur 197121  0 mar  2 12:07 ../
drwxr-xr-x 1 Arthur 197121  0 mar  2 12:07 .git/
-rw-r--r-- 1 Arthur 197121 11 mar  2 12:12 meu_primeiro_arquivo_git.txt

```

Figura 7 - Explorando os arquivos existentes no diretório.

Fonte: próprio do autor.

Note que existe uma pasta chamada `.git` que é criada por padrão pelo o git, contudo só se é possível observar a pasta por um gerenciador de arquivo como o git bash, você não irá encontrar a mesma em seus arquivos, isso ocorre porque essa pasta está protegida e oculta, e só deve ser alterada por comandos via git bash. Sendo assim o real conteúdo do projeto em questão é o seguinte:

```

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ ls
meu_primeiro_arquivo_git.txt

```

Figura 8 - Mostrando os arquivos visíveis do diretório.

Fonte: próprio do autor.

Gerenciando arquivos no working directory:

- **git status**

Após inicializar um novo projeto, os arquivos já pré-existentes estarão no modo preparado como citado na teoria dos diretórios de trabalho, isso ocorre pois os mesmos estão situados em um espaço local da máquina de trabalho podendo ser verificado com o comando "git status" que verifica todos os arquivos presentes na pasta local e seus respectivos estados.

Os arquivos retornados coloridos por vermelhos representam os arquivos que estão situados no working directory, por vez esses arquivos podem ter status de acordo com a soma de verificação, os arquivos com coloração verde estão presentes na staging area, ou seja, estão no estado preparados esperando por um commit. Vale citar que os arquivos não alterados e comitados não aparecem quando se é utilizado o git status isso porque estes estão salvos no git directory e não representam algo significativo no trabalho de controle de versão, já que eles não foram alterados não há a necessidade de salvar novamente. o arquivo "meu_primeiro_arquivo_git.txt" está no work directory, e portanto está com status "alterado".

```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git status
On branch master

No commits yet

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

    meu_primeiro_arquivo_git.txt

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

Figura 9 - Exibindo os status do git.

Fonte: próprio do autor.

- **git add**

Note que o próprio texto retornado já nos dar um boa dica do que deve ser feito para salvar o arquivo, ele sugere que seja feito o comando "git add" que serve para salvar o arquivo na staging area, utilizando o comando git add, o git basicamente move o instantâneo do arquivo para uma outra camada, porém isso não significa que o mesmo não pode ser alterado em qualquer instante, em termos mais simples significa que ele foi salvo em um pacote aguardando um commit, que irá criar uma versão de projeto. Executando o comando "git add [nome_arquivo]" temos o seguinte:

```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git add meu_primeiro_arquivo_git.txt

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git status
On branch master

No commits yet

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

    new file:   meu_primeiro_arquivo_git.txt
```

Figura 10 - Adicionando um arquivo na staging area.

Fonte: próprio do autor.

Observe que antes do nome do arquivo existe uma denominação "new file", isso significa que o arquivo não existia no staging area e nem no git directory e portanto deve ser criado um novo instantâneo para o mesmo quando este for comitado. Quando feito o add o arquivo está em dois locais diferentes, existe um arquivo no working directory e um no staging area, ambos os arquivos são iguais e portanto o commit pode ser realizado já que a soma de verificação não aponta nenhuma incompatibilidade entre os

arquivos. Contudo note que se o arquivo for aberto e seu conteúdo alterado de "Hello world" para "Hello world 2" a soma de verificação notará uma modificação do arquivo e portanto o arquivo na camada do staging área será diferente do arquivo do working directory. Quando realizado um git status será notado que existe uma modificação no arquivo do working directory e antes do nome do mesmo estará o nome "modified" simbolizando a modificação.

```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git status
On branch master

No commits yet

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

        new file:   meu_primeiro_arquivo_git.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   meu_primeiro_arquivo_git.txt
```

Figura 11 - Mostrando a existência de dois arquivos em diferentes diretórios.

Fonte: próprio do autor.

- **git diff**

A diferença entre os arquivos dessas duas camadas pode ser verificada a partir do comando "git diff [nome_arquivo]", ele irá retornar todas as linhas alteradas e o conteúdo do arquivo anterior e do arquivo modificado. Esse comando pode ser utilizado de inúmeras formas diferentes. É possível por exemplo verificar a diferença entre duas versões de projeto como será mostrado mais a frente, é possível verificar a diferença de tamanho em caracteres, entre outras coisas que fogem do foco de um guia rápido e intuitivo.

```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git diff meu_primeiro_arquivo_git.txt
diff --git a/meu_primeiro_arquivo_git.txt b/meu_primeiro_arquivo_git.txt
index 95d09f2..93ab6b3 100644
--- a/meu_primeiro_arquivo_git.txt
+++ b/meu_primeiro_arquivo_git.txt
@@ -1,1 @@
-Hello world
\ No newline at end of file
+Hello world 2
\ No newline at end of file
```

Figura 12 - Exibindo a diferença entre duas versões de um arquivo.

Fonte: próprio do autor.

Note que o "--- a" e "+++ b" representam os arquivos comparados, o "+++" simboliza o arquivo alterado, que possui a nova versão do arquivo, da mesma forma o "---" representa o arquivo antigo, que nesse caso é o arquivo situado no staging área, os números em azul simbolizam as linhas que estão sendo modificadas, -1 simboliza que uma linha será substituída e o mais +1 significa que uma linha será adicionada, em termos mais simples ele irá sobrescrever essa linha pela linha no arquivo novo caso seja realizado um git add. caso exista uma linha nova no arquivo modificado como por exemplo "ola mundo", será descrito o número de linhas novas somadas as linhas substituídas, algo como o seguinte:

```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git diff meu_primeiro_arquivo_git.txt
diff --git a/meu_primeiro_arquivo_git.txt b/meu_primeiro_arquivo_git.txt
index 95d09f2..2d10aa9 100644
--- a/meu_primeiro_arquivo_git.txt
+++ b/meu_primeiro_arquivo_git.txt
@@ -1,2 @@
-hello world
\ No newline at end of file
+hello world 2
+ola mundo
\ No newline at end of file
```

Figura 13 - Exibindo a diferença entre duas versões de um arquivo após alteração.

Fonte: próprio do autor.

- **git commit**

Após colocar todos os arquivos que se deseja commitar na staging area, o seu projeto está pronto para ser comitado. Para realizar o commit basta utilizar o comando "git commit -m 'Mensagem_Commit'", seus arquivos que estão situados na staging area irão gerar uma versão de projeto, a mensagem 'Mensagem_Commit' é apenas um texto que você colocar para identificar o propósito do commit, ele serve para o entendimento do que foi realizado na versão do commit.

```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git commit -m "Primeiro commit"
[master (root-commit) e338193] Primeiro commit
1 file changed, 2 insertions(+)
create mode 100644 meu_primeiro_arquivo_git.txt

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git status
On branch master
nothing to commit, working tree clean
```

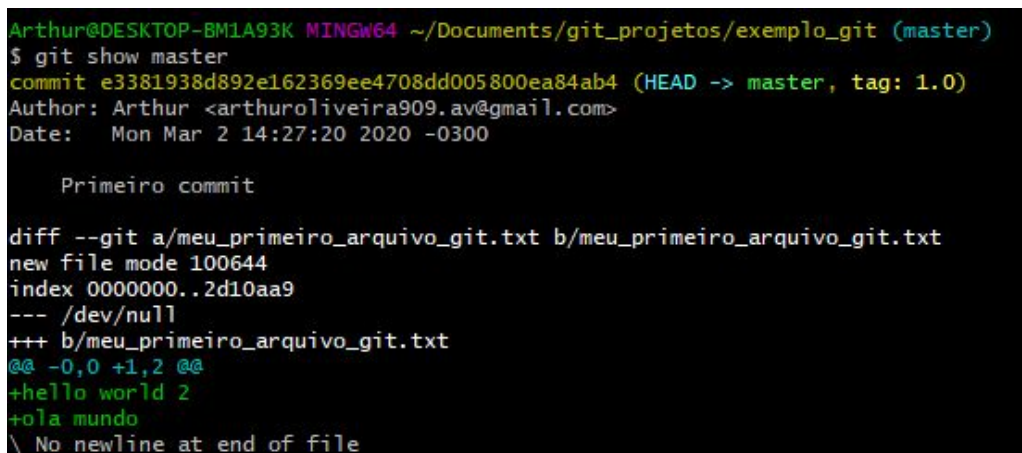
Figura 14 - Executando o commit.

Fonte: próprio do autor.

O git commit tem como retorno a quantidade de arquivos do commit, a branch, e a mensagem de commit. Note que após o commit a staging area é esvaziada, pois os arquivos contidos nela foram salvos no git directory.

- **git show**

O comando git show como o próprio nome sugere, serve para verificar o que existe em uma branch ou versão de um determinado projeto. O comando inteiro é o seguinte: "git show [branch ou tag]". O git show retorna a versão atual do projeto, o autor, a data do projeto, a mensagem do commit e características dos arquivos presentes no projeto.



```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git show master
commit e3381938d892e162369ee4708dd005800ea84ab4 (HEAD -> master, tag: 1.0)
Author: Arthur <arthuroliveira909.av@gmail.com>
Date:   Mon Mar 2 14:27:20 2020 -0300

    Primeiro commit

diff --git a/meu_primeiro_arquivo_git.txt b/meu_primeiro_arquivo_git.txt
new file mode 100644
index 0000000..2d10aa9
--- /dev/null
+++ b/meu_primeiro_arquivo_git.txt
@@ -0,0 +1,2 @@
+hello world 2
+ola mundo
\ No newline at end of file
```

Figura 15 - Exibindo a versão do projeto apontada pela branch master.

Fonte: próprio do autor.

Controle de versão e marcadores:

- **git log**

Após realizar um commit o git criará uma versão de projeto baseada em instantâneo como já citado na teoria, essas versões podem ser vistas a partir do comando "git log". O git log é um registro que contém todo o histórico de commit, merge, etc, ele retornar dentro do git bash uma lista com tudo que ocorreu com o projeto desde se sua criação até o presente momento. Tomando como exemplo o projeto utilizado no comando anterior temos que foi realizado um commit de um (1) arquivo com a mensagem de commit "Primeiro commit", ao utilizar o comando git log o git bash irá retornar o seguinte:

```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git log
commit e3381938d892e162369ee4708dd005800ea84ab4 (HEAD -> master)
Author: Arthur <arthuroliveira909.av@gmail.com>
Date: Mon Mar 2 14:27:20 2020 -0300

    Primeiro commit
```

Figura 16 - Exibindo o registro de versão do projeto.

Fonte: próprio do autor.

O git log assim como todos os comandos no git tem várias variações e formas de se utilizar, os argumentos podem exibir diferenças entre versões, exibir merges de forma estruturada entre outras coisas.

- **git tag**

O comando git tag serve para referenciar uma versão a partir de uma tag, normalmente ao utilizar o comando checkout seria necessário passar o hash SHA-1 para que o git saiba qual versão deve carregar no seu diretório de trabalho, contudo observe que uma string de 40 caracteres não é algo trivial de ficar se trabalhando. As tags comportam um atalho que ligam uma versão a uma string mais familiar ao programador como 1.0 ou v1.0 para determinar a versão 1.0 de um projeto. Utilizar tags não é algo obrigatório, é apenas considerado boas práticas, a não ser que se utilize algum padrão de projeto ou algo do tipo.

O comando git tag possui três (3) argumentos extremamente utilizados, o primeiro é uma string "git tag [nome_tag]" ele cria uma tag e vincula ao projeto atual.

```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git tag 1,0

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git log
commit e3381938d892e162369ee4708dd005800ea84ab4 (HEAD -> master, tag: 1.0)
Author: Arthur <arthuroliveira909.av@gmail.com>
Date: Mon Mar 2 14:27:20 2020 -0300

    Primeiro commit
```

Figura 17 - Criando uma tag.

Fonte: próprio do autor.

O segundo argumento é o -l que lista todas as tags vinculadas ao projeto, o comando completo é o seguinte "git tag -l".

```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git tag -l
1,0
```

Figura 18 - Listando as tags existentes.

Fonte: próprio do autor.

O terceiro argumento é o `-d [Nome_tag]` que deleta a tag cujo o nome for referenciada, o comando completo é o seguinte `"git tag -d [nome_tag]"`.

```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git tag -d 1,0
Deleted tag '1,0' (was e338193)
```

Figura 19 - Deletando uma tag.

Fonte: próprio do autor.

● git branch

Como já citado anteriormente todo projeto gerenciado pelo git reside inicialmente em um branch denominada "master", e como também citado é normal serem criadas várias branches ao longo da vida de um projeto. Desta forma o comando `"git branch [nome_branch]"` tem como finalidade criar uma nova branch copiando os arquivos da branch atual. Essa nova branch terá um git directory, staging area e work directory próprios.

```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git branch teste
```

Figura 19 - Criando uma branch.

Fonte: próprio do autor.

Para verificar as branches existentes basta utilizar o argumento `-l` e para excluir uma branch, basta utilizar o comando `-d` como demonstrado abaixo.

```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git branch -l
* master
  teste
```

Figura 20 - Listando as branches existentes.

Fonte: próprio do autor.

```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git branch -d teste
Deleted branch teste (was 15665f4).

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git branch -l
* master
```

Figura 21 - Deletando uma branch.

Fonte: próprio do autor.

- **git checkout**

Para acessar uma branch criada e carregar seus arquivos no git directory se utiliza o comando checkout, este por sua vez irá carregar o instantâneo da versão atual da branch no seu working directory. O git checkout também é utilizado para carregar versões passadas de um projeto, isso pode ser feito com o comando "git checkout [hash SHA-1 ou tag]".

```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git checkout teste
Switched to branch 'teste'

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (teste)
$ |
```

Figura 22 - Trocando de branch utilizando o checkout.

Fonte: próprio do autor.

```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git log --pretty=oneline
15665f40a9ad0c34ff42acc00df27d6068e102f7 (HEAD -> master, tag: 1.0.1, teste) Alteração
e3381938d892e162369ee4708dd005800ea84ab4 (tag: 1.0) Primeiro commit

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git checkout 1.0
Note: checking out '1.0'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

    git checkout -b <new-branch-name>

HEAD is now at e338193 Primeiro commit

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git ((1.0))
$ git log --pretty=oneline
e3381938d892e162369ee4708dd005800ea84ab4 (HEAD, tag: 1.0) Primeiro commit
```

Figura 23 - Carregando uma versão anterior no working directory.

Fonte: próprio do autor.

- **git merge**

A partir do comando git merge é possível unir duas branch distintas em uma. Naturalmente irá surgir conflitos ao tentar unir duas branch distintas, esses por sua vez devem ser tratados com cuidado, pois esses conflitos devem ser tratados manualmente pela pessoa que está realizando o merge. Por exemplo, o arquivo "meu_primeiro_arquivo_git.txt" contém na branch master uma cadeia de string, se for criado uma nova branch com a cópia dos arquivos da branch master e na atual master for realizado um commit alterando uma linha dessa string, similar a master o mesmo arquivo for alterado na outra branch e comitado, ao tentar realizar um

merge aparecerá um conflito no arquivo. Abaixo segue o exemplo do conflito:

```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git branch -l
* master
  teste

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git add *

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git commit -m "Teste merge"
[master 92a201e] Teste merge
1 file changed, 1 insertion(+), 1 deletion(-)

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git checkout teste
Switched to branch 'teste'

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (teste)
$ git add *

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (teste)
$ git commit -m "Alteração teste merge"
[teste 029fd55] Alteração teste merge
1 file changed, 1 insertion(+), 1 deletion(-)

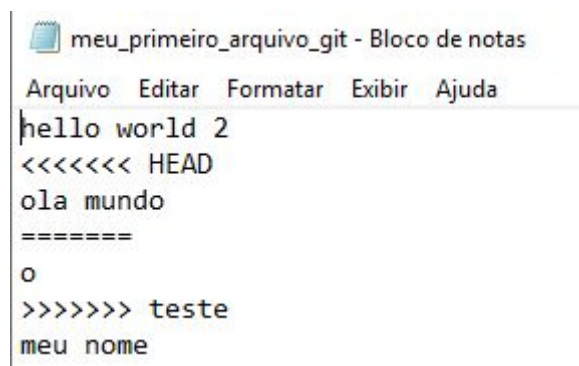
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (teste)
$ git checkout master
Switched to branch 'master'

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git merge teste
Auto-merging meu_primeiro_arquivo_git.txt
CONFLICT (content): Merge conflict in meu_primeiro_arquivo_git.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Figura 24 - Conflito ao tentar realizar o merge.

Fonte: próprio do autor.

O git marcará no arquivo "meu_primeiro_arquivo_git.txt" a linha com o conflito e deve ser escolhido dentro do arquivo uma das duas opções marcadas pelo git.



```
meu_primeiro_arquivo_git - Bloco de notas
Arquivo  Editar  Formatar  Exibir  Ajuda
hello world 2
<<<<<<< HEAD
ola mundo
=====
o
>>>>>> teste
meu nome
```

Figura 25 - Marcação no arquivo feito pelo git.

Fonte: próprio do autor.

Após alterar o arquivo é necessário que seja adicionado a staging area do "master|MERGING" e comitado para o mesmo. Após realizar isso o git voltará para a branch master e o conflito terá sido corrigido.

```

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/emplo_git (master|MERGING)
$ git add meu_primeiro_arquivo_git.txt

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/emplo_git (master|MERGING)
$ git commit
[master c094a9e] Merge branch 'teste'

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/emplo_git (master)
$ git log
commit c094a9e5c3e3a5742dfeae64af24127f10ec6a6e (HEAD -> master)
Merge: 92a201e 029fd55
Author: Arthur <arthuroliveira909.av@gmail.com>
Date: Tue Mar 3 20:22:07 2020 -0300

    Merge branch 'teste'

commit 029fd55aebd44365f987de260da519c00952a9a6 (teste)
Author: Arthur <arthuroliveira909.av@gmail.com>
Date: Tue Mar 3 20:12:23 2020 -0300

    Alteração teste merge

commit 92a201e3083ff8eca4dfb55e3828432a1b850f39
Author: Arthur <arthuroliveira909.av@gmail.com>
Date: Tue Mar 3 20:11:44 2020 -0300

    Teste merge

commit 0cbc72e96b206daacd29bc80bc7e2af71e02b0bf
Author: Arthur <arthuroliveira909.av@gmail.com>
Date: Tue Mar 3 20:08:53 2020 -0300

    Mudança

commit 15665f40a9ad0c34ff42acc00df27d6068e102f7 (tag: 1.0.1)
Author: Arthur <arthuroliveira909.av@gmail.com>
Date: Tue Mar 3 16:03:25 2020 -0300

    Alteração

commit e3381938d892e162369ee4708dd005800ea84ab4 (tag: 1.0)
Author: Arthur <arthuroliveira909.av@gmail.com>
Date: Mon Mar 2 14:27:20 2020 -0300

    Primeiro commit

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/emplo_git (master)
$ |

```

Figura 26 - Corrigindo o conflito.

Fonte: próprio do autor.

Note que no log do arquivo aparece que a versão “Alteração teste merge” foi criado pela branch teste e a atual versão é tida como Merge branch ‘teste’ para simbolizar que foi realizado um merge.

● git reset

Caso uma operação de merge por exemplo tenham conflitos complexos ou uma operação de adição tenha sido efetuado de maneira errada e você deseje retroceder essas ou até mesmo limpar a staging area o comando reset irá realizar tal desejo.

```

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git merge teste
Auto-merging meu_primeiro_arquivo_git.txt
CONFLICT (content): Merge conflict in meu_primeiro_arquivo_git.txt
Automatic merge failed; fix conflicts and then commit the result.

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master|MERGING)
$ git reset --merge

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ |

```

Figura 27 - Resetando a área de merge.

Fonte: próprio do autor.

```

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git add meu_primeiro_arquivo_git.txt

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   meu_primeiro_arquivo_git.txt

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git reset HEAD
Unstaged changes after reset:
M       meu_primeiro_arquivo_git.txt

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   meu_primeiro_arquivo_git.txt

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

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ |

```

Figura 28 - Resetando a staging area.

Fonte: próprio do autor.

Repositórios Remotos

Configurando o repositório remoto no github e gitlab:

O git como citado tem a capacidade de se conectar com repositórios remotos, mas para que seja possível tal conexão, é necessário que seja criado uma chave de acesso denominada ssh que contém os dados da máquina local que irá acessar o repositório. Essa chave deve ser configurada no repositório para que o acesso seja possível.

A maneira mais simples de se gerar uma ssh é a partir comando ssh-keygen, que deve executado no git bash. O preenchimento dos

campos que serão requisitados não é obrigatório, portanto em todas as perguntas se pode apertar enter e prosseguir.

```
Arthur@DESKTOP-BM1A93K MINGW64 ~  
$ ssh-keygen  
Generating public/private rsa key pair.  
Enter file in which to save the key (/c/Users/Arthur/.ssh/id_rsa):  
/c/Users/Arthur/.ssh/id_rsa already exists.  
Overwrite (y/n)? y  
Enter passphrase (empty for no passphrase):  
Enter same passphrase again:  
Your identification has been saved in /c/Users/Arthur/.ssh/id_rsa.  
Your public key has been saved in /c/Users/Arthur/.ssh/id_rsa.pub.
```

Figura 29 - Gerando uma chave ssh.
Fonte: próprio do autor.

Após executar o comando será criado (se não for alterado o caminho padrão do git) uma pasta `.ssh` que estará em `"c:\Nome_Usuario\.ssh"`, dentro desta pasta está a chamada ssh pública que deve ter seu conteúdo copiado para a configuração da conta github ou gitlab.

Independente de qual plataforma você decida usar, os caminhos para a configuração são os mesmos. Após logado na plataforma se deve seguir o caminho "Settings > SSH and GPG keys ou SSH keys", estando no caminho, basta clicar em new ssh, colar o conteúdo da ssh pública no espaço destinado a ela e dar um nome qualquer para a ssh, depois basta clicar sobre um botão de adicionar ssh e pronto. Desde que a sua máquina local tenha uma conexão com a internet você já pode upar seus projetos em repositórios remotos criados nestas plataformas.

Gerenciamento de repositório remoto :

● git clone

Quando se deseja carregar um repositório remoto se utiliza o comando `"git clone [repositório]"`, este realiza uma conexão com o servidor passado e clonar o conteúdo para a pasta do diretório em que o git bash está situado. Ao executar esse comando ele também cria um "remoto" que é nada mais que o caminho do repositório. Remotos podem ser visualizados a partir do comando `"git remote"`, nesse caso ele cria um remoto padrão denominado "origin". Para demonstrar este exemplo vamos clonar um repositório meu situado no github e que pode ser acessado a partir do link (<https://github.com/ArthurOlive/Maquina-de-Turing-em-c>).

```

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exempleRemote
$ git clone https://github.com/ArthurOlive/Maquina-de-Turing-em-c.git
Cloning into 'Maquina-de-Turing-em-c'...
remote: Enumerating objects: 30, done.
remote: Counting objects: 100% (30/30), done.
remote: Compressing objects: 100% (20/20), done.
remote: Total 30 (delta 14), reused 22 (delta 9), pack-reused 0
Unpacking objects: 100% (30/30), done.

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exempleRemote
$ ls
Maquina-de-Turing-em-c/

```

Figura 30 - clonando um repositório.

Fonte: próprio do autor.

```

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exempleRemote
$ cd Maquina-de-Turing-em-c/

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exempleRemote/Maquina-de-Turing-em-c (master)
$ git remote show
origin

```

Figura 31 - Exibindo os remotos.

Fonte: próprio do autor.

● git push

Quando se deseja upar um projeto já existente ou até mesmo uma nova versão para o repositório remoto se utiliza o comando "git push [remoto] [branch]". A única observação deste comando é que se o seu ssh não estiver configurado no servidor, será retornado uma mensagem de acesso negado, portanto para este exemplo será necessário criar um repositório no github ou gitlab e configurar assim como descrito na própria plataforma.

...or create a new repository on the command line

```

echo "# exemplo" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/ArthurOlive/exemplo.git
git push -u origin master

```

...or push an existing repository from the command line

```

git remote add origin https://github.com/ArthurOlive/exemplo.git
git push -u origin master

```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

Import code

Figura 32 - Exemplo de como configurar um novo remoto.

Fonte: github.com.

```

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git remote add origin https://github.com/ArthurOlive/exemplo

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exemplo_git (master)
$ git push origin master
Enumerating objects: 17, done.
Counting objects: 100% (17/17), done.
Delta compression using up to 4 threads
Compressing objects: 100% (12/12), done.
Writing objects: 100% (17/17), 1.50 KiB | 307.00 KiB/s, done.
Total 17 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), done.
To https://github.com/ArthurOlive/exemplo
 * [new branch]      master -> master

```

Figura 33 - Demonstrando um git push.

Fonte: próprio do autor.

● git remote

Como citado anteriormente para se ter acesso a um repositório remoto é necessário existir um remoto que contenha o acesso ao repositório, algo similar a uma porta de um servidor onde será realizada requisições http. Para adicionar um novo remoto ao seu projeto basta utilizar o comando "git remote add [nome_remoto] [link]", para visualizar basta digitar "git remote" e o git irá listar os seus arquivos remotos existentes.

```

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exempleRemote/Maquina-de-Turing-em-c (master)
$ git remote add origin git@github.com:ArthurOlive/Maquina-de-Turing-em-c.git

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exempleRemote/Maquina-de-Turing-em-c (master)
$ git remote
origin

```

Figura 34 - Criando um remoto.

Fonte: próprio do autor.

Quando se desejar excluir um remoto basta utilizar o argumento "rm" seguido do nome do remoto, e para visualizar as informações do mesmo basta utilizar o argumento "show" seguido do nome do remoto.

```

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exempleRemote/Maquina-de-Turing-em-c (master)
$ git remote show origin
* remote origin
Fetch URL: git@github.com:ArthurOlive/Maquina-de-Turing-em-c.git
Push URL: git@github.com:ArthurOlive/Maquina-de-Turing-em-c.git
HEAD branch: master
Remote branch:
  master new (next fetch will store in remotes/origin)
Local ref configured for 'git push':
  master pushes to master (local out of date)

```

Figura 36 - Exibindo as características do remoto.

Fonte: próprio do autor.

● git pull

Se um projeto for atualizado e os arquivos presente no servidor não forem iguais a da sua máquina local, o git não

permitirá um git push vindo desta máquina. Isso ocorre para que o git mantenha a integridade do repositório do servidor, nada pode ser perdido quando se executa um push. Desta forma para que seja possível upar a atual versão do projeto no git é necessário que seja realizado o comando "git pull [remoto] [branch]" que irá verificar os arquivos do servidor e sobrescrever os seus arquivos locais pelo que estiver lá.

Para que não seja perdido aquilo que se tem em máquina é necessário que seja feito uma branch para receber a versão atualizada, e posteriormente efetuar um merge para unir as duas versões. Abaixo está um exemplo simples de pull.

```
Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exempleRemote/Maquina-de-Turing-em-c (master)
$ git pull origin master
From github.com:ArthurOlive/Maquina-de-Turing-em-c
* branch      master      -> FETCH_HEAD
Auto-merging README.md
Merge made by the 'recursive' strategy.
 README.md | 2 ++
 1 file changed, 2 insertions(+)

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exempleRemote/Maquina-de-Turing-em-c (master)
$ git log
commit e31bb464ee6b87509acfbfb720f0e5fa457bc800 (HEAD -> master)
Merge: 498cd75 a5edfb5
Author: Arthur <arthuroliveira909.av@gmail.com>
Date:   Mon Mar 9 20:41:55 2020 -0300

    Merge branch 'master' of github.com:ArthurOlive/Maquina-de-Turing-em-c

commit 498cd756b1e0f4b9e18d305ec22a64a273160f50 (copia)
Author: Arthur <arthuroliveira909.av@gmail.com>
Date:   Mon Mar 9 20:39:32 2020 -0300

    Teste
```

Figura 37 - Git pull.

Fonte: próprio do autor.

● git fetch

Por fim o git fetch é o comando que se utiliza quando não se tem certeza do que realmente foi alterado no servidor remoto, ele cria uma branch para aquilo que estiver no servidor e mantém seus arquivos no diretório. Posteriormente para unir se utiliza o comando "git merge [remoto]/[branch]".

```

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exempleRemote/Maquina-de-Turing-em-c (master)
$ git fetch

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exempleRemote/Maquina-de-Turing-em-c (master)
$ git status
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.
(use "git pull" to update your local branch)

nothing to commit, working tree clean

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/exempleRemote/Maquina-de-Turing-em-c (master)
$ git log origin/master
commit a5edfb52da129a5578ebbf2a6d7f19f6ed8d9022 (origin/master, origin/HEAD)
Author: Arthur Oliveira <39603069+ArthurOlive@users.noreply.github.com>
Date: Mon Mar 9 20:11:34 2020 -0300

    atualizacao

commit 5238430721b090b6f8a02f0c8b2f606bed01d072 (HEAD -> master)
Author: Arthur <arthuroliveira909.av@gmail.com>
Date: Thu Jan 30 17:17:11 2020 -0300

    Alteração do README.md

```

Figura 38 - Git fetch.

Fonte: próprio do autor.

```

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/Maquina_Turing_Configuravel_c (master)
$ git merge origin/master
Updating 5238430..a5edfb5
Fast-forward
 README.md | 2 ++
 1 file changed, 2 insertions(+)

Arthur@DESKTOP-BM1A93K MINGW64 ~/Documents/git_projetos/Maquina_Turing_Configuravel_c (master)
$ git log
commit a5edfb52da129a5578ebbf2a6d7f19f6ed8d9022 (HEAD -> master, origin/master)
Author: Arthur Oliveira <39603069+ArthurOlive@users.noreply.github.com>
Date: Mon Mar 9 20:11:34 2020 -0300

    atualizacao

```

Figura 39 - Unir duas versões após fetch.

Fonte: próprio do autor.

Bibliografia

[1] Pro Git Scott Chacon, Ben Straub Version 2.1.176, 2019-11-15 <disponível em : <https://git-scm.com/doc> >