

## SUMÁRIO

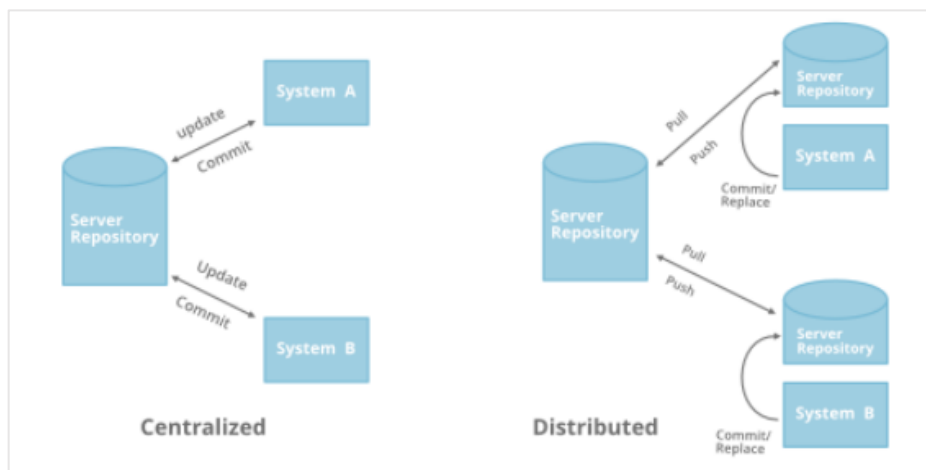
<b>1. INTRODUÇÃO: O QUE É GIT .....</b>	<b>1</b>
1.1. O GIT É UM DVCS: SISTEMA DE CONTROLE DE VERSÃO DISTRIBUÍDO .....	1
1.2. GIT NÃO É IGUAL A GITHUB .....	1
<b>2. INSTALAÇÃO DO GIT .....</b>	<b>2</b>
<b>3. CONCEITUANDO O FUNCIONAMENTO DO GIT .....</b>	<b>2</b>
3.1. SISTEMA DE MONITORAMENTO.....	2
3.2. CICLO DE VIDA DO STATUS DOS ARQUIVOS .....	2
3.2.1. STATUS UNTRACKED – NÃO RASTREADO .....	2
3.2.2. STATUS TRACKED – RASTREADO .....	3
3.2.3. STATUS MODIFIED – MODIFICADO .....	3
3.2.4. STATUS STAGED – PRONTO .....	3
3.3. ARQUIVO COM EXTENSÃO .GITIGNORE.....	3
3.4. PROMPT DE COMANDOS DO GIT .....	3
3.4.1. COMANDO GIT INIT .....	3
3.4.2. COMANDO GIT CONFIG – MELHORANDO O CONTROLE DO REPOSITÓRIO .....	4
3.4.3. COMANDO GIT STATUS E .GITIGNORE NA PRÁTICA .....	5
3.4.4. COMANDO GIT TOUCH – CRIANDO UM ARQUIVO .....	5
3.4.5. COMANDO GIT ADD <ARQUIVO> - ADICIONANDO AO STAGING AREA .....	5
3.4.6. COMANDO GIT COMMIT.....	6
3.4.7. COMANDO GIT LOG.....	7
3.4.8. COMANDO GIT BRANCH .....	7
3.4.9. COMANDO GIT CHECKOUT .....	7
3.4.10. COMANDO GIT MERGE .....	7
3.4.11. COMANDOS CLONE E PUSH – DO CLONE PARA A ORIGEM .....	8
3.4.12. COMANDOS FETCH E PULL – DA ORIGEM PARA O CLONE .....	8
3.4.13. REPOSITÓRIO BARE – UM REPOSITÓRIO “PUSHABLE” .....	9
3.5. TAGS DE COMMIT.....	9
3.6. GITHUB, GITLAB E BITBUCKET – HOSPEDAGENS DE REPOSITÓRIOS .....	10
<b>FONTE .....</b>	<b>11</b>

## 1. INTRODUÇÃO: O QUE É GIT

O Git é um projeto de código aberto maduro e com manutenção ativa desenvolvido em 2005 por Linus Torvalds, o famoso criador do kernel do sistema operacional Linux. Um número impressionante de projetos de software depende do Git para controle de versão, incluindo projetos comerciais e de código-fonte aberto.

### 1.1. O GIT É UM DVCS: SISTEMA DE CONTROLE DE VERSÃO DISTRIBUÍDO

Tendo uma arquitetura distribuída, **o Git é um exemplo de DVCS** (portanto, Sistema de Controle de Versão Distribuído). Em vez de ter apenas um único local para o histórico completo da versão do software, como é comum em sistemas de controle de versão outrora populares, como CVS ou Subversion (também conhecido como SVN), no Git, a cópia de trabalho de todo desenvolvedor do código também é um repositório que pode conter o histórico completo de todas as alterações.



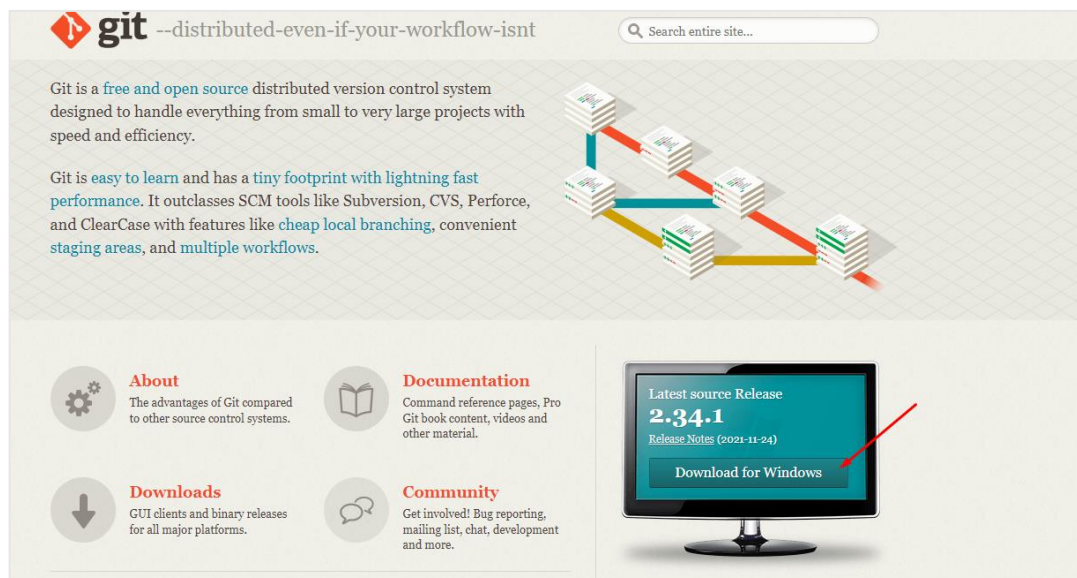
### 1.2. GIT NÃO É IGUAL A GITHUB

Como já explicado, o Git é um programa que serve para gerenciar o controle e o versionamento de códigos e documento, enquanto o Github é uma plataforma web que serve para centralizar repositórios na Internet, e que opera utilizando os recursos do Git.



## 2. INSTALAÇÃO DO GIT

Para instalar o Git é preciso acessar esse site, e realizar o download de acordo com o sistema operacional utilizado.



No caso do Windows, será utilizado um instalador, e, na maioria dos casos, as configurações padronizadas são suficientes. Assim, **basta clicar em “next” em todas as seções do instalador**, para avançar até o script de instalação em si do programa. Quando concluído, para conferir se foi instalado, é preciso digitar “git –version” no prompt de comando do Windows.

## 3. CONCEITUANDO O FUNCIONAMENTO DO GIT

### 3.1. SISTEMA DE MONITORAMENTO

Por ser um sistema de gerenciamento de versões de arquivos, o Git possui recursos que permitem o monitoramento e acompanhamento. Após a inicialização do monitoramento, ele passa a construir um **histórico das fases de desenvolvimento do arquivo**.

### 3.2. CICLO DE VIDA DO STATUS DOS ARQUIVOS

Esse processo, de quantização e registro das fases de alteração dos arquivos monitorados, é chamado de “ciclo de vida do estado do arquivo” ou “ciclo de vida do status do arquivo”, e existem, ao todo, **quatro status ou estados** de um arquivo Git: **Untracked, Tracked, Modified e Staged**.

#### 3.2.1. STATUS UNTRACKED – NÃO RASTREADO

Esse status seria de um arquivo que, fazendo parte ou não de um projeto, ainda não foi ou não está sendo monitorado, e, portanto, tendo seu versionamento realizado.

### 3.2.2. STATUS TRACKED – RASTREADO

Um arquivo com esse status é um que passou a ser monitorado pelo Git, permitindo que seu desenvolvimento e quaisquer alterações realizadas sejam registradas.

### 3.2.3. STATUS MODIFIED – MODIFICADO

Esse status é atribuído a um arquivo sofreu uma alteração de estado detectada pelo sistema de monitoramento do Git.

### 3.2.4. STATUS STAGED – PRONTO

Esse status é atribuído a um arquivo que está sendo rastreado, foi modificado, está finalizado e preparado para ser enviado para o repositório.

O processo de confirmar e salvar o estado atual do arquivo no histórico de fases do projeto, com um nome e uma descrição, é chamado de “Commit”. Brasileiros comumente utilizam o termo “Commitar” para essa ação.

## 3.3. ARQUIVO COM EXTENSÃO .GITIGNORE

Esse tipo de arquivo, com essa extensão, serve para configurar os arquivos que não devem ser monitorados dentro do projeto. Geralmente é utilizado na plataforma Github para preservar o conteúdo de arquivos que possuem informações sensíveis.

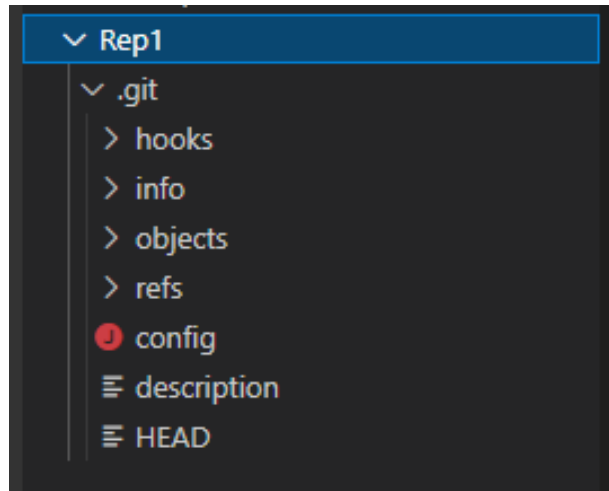
## 3.4. PROMPT DE COMANDOS DO GIT

**Na prática o uso do Git se dá por linha de comando**, utilizando o terminal do próprio sistema operacional, Gitbash que é um terminal incluso na instalação do Git, ou até mesmo o terminal da IDE utilizada, como o do Visual Studio Code.

### 3.4.1. COMANDO GIT INIT

**Este comando é usado para criar um repositório GIT local.** Se utilizado, é criado um repositório de nome oculto, .git, no caminho que foi configurado para criação padrão de repositórios.

Acessando o caminho de uma pasta de nome “Rep1”, para exemplo, executando o comando “git init” e acessando a pasta pelo Vs Code, devem ser encontrados esses arquivos:



### 3.4.2. COMANDO GIT CONFIG – MELHORANDO O CONTROLE DO REPOSITÓRIO

Um dos comandos git mais usados é o **git config** que pode ser usado para definir valores de configuração específicos do repositório, como usuário, e-mail, formato de arquivo etc. Elas não serão usadas para fins de autenticação, e sim para melhor controle das alterações realizadas nos arquivos. Um exemplo de uso prático dessas definições, é em um projeto com vários contribuintes trabalhando em diferentes repositórios.

Após criado o repositório com “git init” pode ser utilizado o comando “git config user.name ‘Nome Completo’” para criar o atributo “nome” do criador do repositório, e o comando “git config user.email ‘email@domain’” para definir o seu email.

O **arquivo de configurações** é gerado na criação do repositório, e os comandos executados pelo próprio criador, como esse citado, são dados adicionais que são adicionados a esse arquivo.

```

Rep1 > .git > config
1  [core]
2      repositoryformatversion = 0
3      filemode = false
4      bare = false
5      logallrefupdates = true
6      symlinks = false
7      ignorecase = true
8  [user]
9      name = Marcelo de Souza Moreira
10     email = profissionaismbr@gmail.com
11
    
```

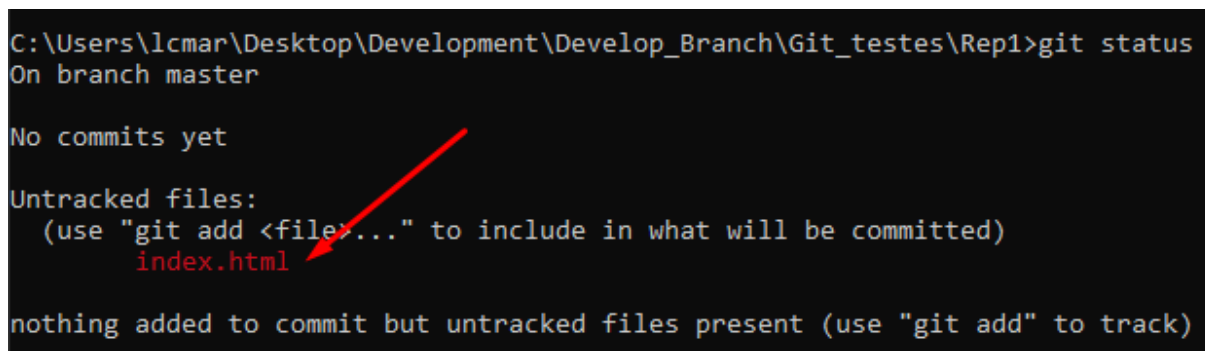
ATRIBUTOS ADICIONADOS

Se utilizada a flag `--global` antes do tipo do atributo que será adicionado, a definição não será pertencente ao repositório local apenas, mas se dará em nível global. Isso deve ser evitado quando, por exemplo, o computador é utilizado por múltiplos usuários.

### 3.4.3. COMANDO GIT STATUS E .GITIGNORE NA PRÁTICA

Existindo um repositório Git, pode ser utilizado o comando `Git status` para verificar algumas informações do repositório, como a branch atual, assunto que será tratado posteriormente, se já foram realizados commits e os arquivos monitorados e não monitorados dentro do projeto.

Por exemplo, criando um `“index.html”` dentro do repositório criado, e realizando esse comando, será informando que esse arquivo foi detectado, mas está com status `“untracked”`, isto é, não rastreado.



```
C:\Users\lcmar\Desktop\Development\Develop_Branch\Git_testes\Rep1>git status
On branch master

No commits yet

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

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

Agora, se criado um arquivo `.gitignore`, e se escrito nele `“index.html”`, sem aspas, com um novo comando `“git status”` esse arquivo não constará como untracked, porque por estar sendo ignorado, realmente não deve ser rastreado.

### 3.4.4. COMANDO GIT TOUCH – CRIANDO UM ARQUIVO

Com o comando `“git touch <nome do arquivo>.<extensão>”` um arquivo pode ser criado no caminho atual, referido pelo terminal. O arquivo também pode ser criado manualmente, como foi feito na seção anterior com o `“index.html”`.

### 3.4.5. COMANDO GIT ADD <ARQUIVO> - ADICIONANDO AO STAGING AREA

O comando `“git add”` pode ser usado para adicionar arquivos ao monitoramento e fazendo com que tenham o status `“tracked”` ou `“rastreados”`. Para ser rastreado, o arquivo `“index.html”`, criado na seção anterior, deveria ser utilizado esse comando.

Essa área onde o Git armazena os arquivos rastreados é popularmente chamada de `“staging”` ou `“staging área”`.

```
C:\Users\lcmar\Desktop\Development\Develop_Branch\Git_testes\Rep1>git add index.html

C:\Users\lcmar\Desktop\Development\Develop_Branch\Git_testes\Rep1>git status
On branch master

No commits yet

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

AGORA ESTÁ SENDO MONITORADO

Uma variação desse comando é o “git add .” que serve para adicionar todos os arquivos rastreáveis mas que não foram rastreados ao sistema de monitoramento.

### 3.4.6. COMANDO GIT COMMIT

Os commits são pacotes de um cronograma ou histórico de projeto Git. **Podem ser considerados screenshots** ou marcos ao longo do cronograma de um projeto Git. São criados com o comando git commit para capturar o estado de um projeto naquele momento, e **são acompanhados de um nome e uma descrição das alterações realizadas**.

Por exemplo, se modifico o arquivo “index.html”, agora monitorado pelo Git, primeiro, com “git status”, seria impressa a mensagem de que o arquivo foi modificado.

```
C:\Users\lcmar\Desktop\Development\Develop_Branch\Git_testes\Rep1>git status
On branch master

No commits yet

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

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

C:\Users\lcmar\Desktop\Development\Develop_Branch\Git_testes\Rep1>_
```

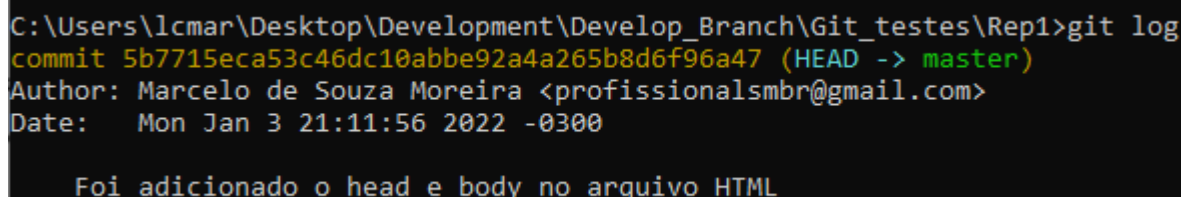
Após a alteração, e antes do Commit, é necessário que o comando “add” seja realizado novamente. Portanto: “git add index.html”. Após isso, então, é realizado o commit.

```
C:\Users\lcmar\Desktop\Development\Develop_Branch\Git_testes\Rep1>git commit -m "Foi adicionado o head e body no arquivo HTML"
[master 5b7715e] Foi adicionado o head e body no arquivo HTML
 1 file changed, 12 insertions(+)

C:\Users\lcmar\Desktop\Development\Develop_Branch\Git_testes\Rep1>_
```

### 3.4.7. COMANDO GIT LOG

É possível ver que o commit, quando realizado, tem um identificador, ou um número de hash de 7 caracteres. Na verdade, são os 7 primeiros caracteres de um hash maior, e que pode ser verificado com o comando “git log” que imprime os commits realizados e registrados pelo sistema.



```
C:\Users\lcmar\Desktop\Development\Develop_Branch\Git_testes\Rep1>git log
commit 5b7715eca53c46dc10abbe92a4a265b8d6f96a47 (HEAD -> master)
Author: Marcelo de Souza Moreira <professionalsmbr@gmail.com>
Date: Mon Jan 3 21:11:56 2022 -0300

Foi adicionado o head e body no arquivo HTML
```

Existem mais variações úteis do comando “git log” e que podem ser estudadas na [documentação](#).

### 3.4.8. COMANDO GIT BRANCH

O comando git branch pode ser usado para listar, criar ou excluir ramificações do projeto. Branches são vertentes diferentes de um mesmo código.

Por padrão, o ramo principal é o “master”, que possui o código inteiro, original, atualizado, testado, pronto para produção, mas poderiam ser criados outros ramos para que o desenvolvimento ocorresse em partes paralelas, e que no final seriam reunidas no ramo Master.

Para listar os ramos deve ser utilizado o comando “git branch”, para adicionar um ramo, “git branch <nome da branch>” e para excluir seria “git branch -d <nome da branch>”.

### 3.4.9. COMANDO GIT CHECKOUT

Na listagem de branches, a que estiver colorida e marcada será a atualmente em uso. Para alternar entre ramos deve ser utilizado o comando “git checkout <nome da branch>”.

### 3.4.10. COMANDO GIT MERGE

O comando git merge é usado para mesclar uma ramificação no ramo ativo. Por exemplo, se a branch atual é a “Master”, utilizando o comando “git merge <X>”, a branch “X” informada será mesclada com a Master. O resultado disso será a transferência das alterações realizadas no ramo “X” para o ramo “Master”.

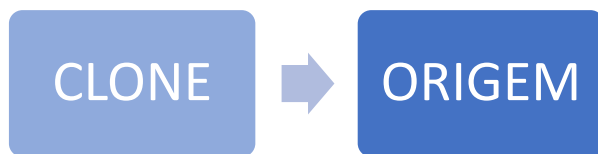


#### 3.4.11. COMANDOS CLONE E PUSH – DO CLONE PARA A ORIGEM

O comando **“git clone”** é usado para copiar um repositório existente em um novo diretório local. A ação de clone do Git criará um diretório local independente para o repositório clonado, copiará todo o conteúdo o seu conteúdo, criará os seus branches e fará o checkout de um branch inicial localmente.

Se realizado uma alteração no repositório clone, e, também um commit, o comando **“git status”** irá informar que o clone estará a frente do original, faltando apenas “uma submissão” para que se encontrem no mesmo estado.

Essa submissão é comando **“git push”**, que significa “git empurrar”, e que serve para, como o nome denota, “empurrar” o conteúdo de um repositório para o outro – neste caso, do clone para o original. Mas, para que seja possível, o repositório original deve ser do tipo **“bare”** – coisa que será abordada em seguida.

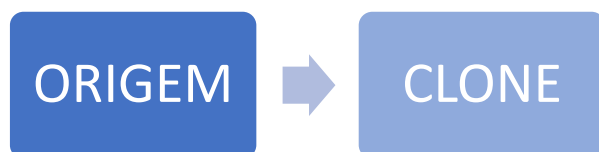


#### 3.4.12. COMANDOS FETCH E PULL – DA ORIGEM PARA O CLONE

O comando **“git fetch”** permite que um usuário obtenha todos os objetos do repositório remoto que atualmente não residem no diretório de trabalho local. Mas ele faz apenas o download dos dados.

Por sua vez, é com o comando **“git pull”** que os recursos baixados com o “git fetch” são “guardados” no repositório local.

Esse caso é exatamente o oposto do anterior. Aquele era do clone para a origem, e esse é da origem para o clone.

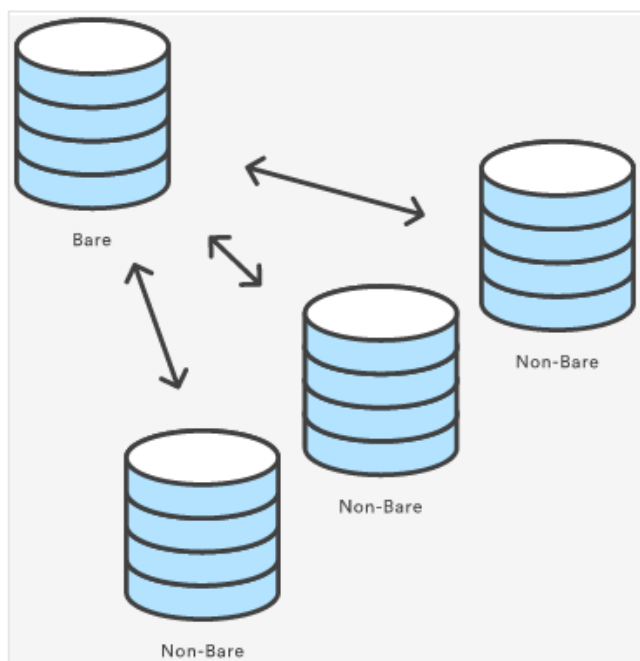


### 3.4.13. REPOSITÓRIO BARE – UM REPOSITÓRIO “PUSHABLE”

**Geralmente os repositórios bare são criados no servidor e são considerados repositórios para armazenamento**, em contraste com os repositórios que existem nas máquinas dos desenvolvedores que seriam os de desenvolvimento, criados com o comando `git init`, e sem a flag `--bare`.

Apesar do GIT ser um sistema de controle de versionamento distribuído, **é muito comum que exista um repositório central que facilite a troca de informações entre os desenvolvedores**, evitando a necessidade de que os computadores dos desenvolvedores se comuniquem diretamente.

**Assim, é necessário que um repositório seja do tipo “bare” para que seja possível “empurrar” conteúdo para ele**, com o comando “`git push`”. Mesmo que ele tenha um clone, não será possível empurrar o conteúdo desse clone para ele, se não for do tipo “bare”.



Podemos dizer que os **repositórios do Github**, que serão abordados em outro documento, são do tipo “bare”, necessariamente.

### 3.5. TAGS DE COMMIT

As tags são etiquetas que demarcam um ponto (commit) que representa alguma mudança significativa no seu código, ou seja, uma versão (ou release) do seu projeto.

**Existem dois tipos de tag: annotated e lightweight.** De maneira bem resumida, podemos dizer que tags annotated armazenam detalhes sobre o estado do repositório naquele

momento, enquanto tags lightweight armazenam apenas o checksum do commit em que foram geradas.

**Para exemplo, será criada aqui uma tag annotated.** A sintaxe é bem simples e parecida com a do commit. Chamaremos a versão de v1.0 e incluiremos a mensagem “v1.0 First Version”.

**git tag -a v1.0 -m “First Version”**

Agora, se executado o comando “**git tag**”, a tag criada será mostrada – v1.0. Para obter informações adicionais sobre a tag, basta digitar, sem aspas, o comando “**git show v1.0**”. Dentre as informações que serão mostradas, estarão o nome de usuário que criou a tag, o checksum do commit que a originou, conteúdo das alterações etc.

Se a tag tiver sido criada em um repositório clone, ela deve ser enviada para o repositório bare com o comando “**git push origin v1.0**”.

Em um serviço de hospedagem, como o Github, uma tag versionada, isto é, criada e registrada pelo Git, é expressa como um “Release” na plataforma. Veja o exemplo abaixo:



Figura 5 — Detalhe do primeiro release

### 3.6. GITHUB, GITLAB E BITBUCKET – HOSPEDAGENS DE REPOSITÓRIOS

Esses nomes se referem a serviços de hospedagem de repositórios na nuvem e que utilizam, pelo menos os dois primeiros, como o próprio nome denota, o programa de versionamento “Git”.

Ou seja: são gerenciadores de repositórios Git com interfaces gráficas web e serviços adicionais relacionados a gerenciamento de repositórios e equipes.

**FONTE**

<https://www.udemy.com/course/curso-de-git-e-github-essencial/>

<https://medium.com/rafaeltardivo/git-github-uma-introdu%C3%A7%C3%A3o-pr%C3%A1tica-fed91ee56b8d>