# Git

Neni

10 de Agosto de 2019

# Conteúdo

# Listings

# Lista de Figuras



# Lista de Tabelas

## Capítulo 1

# O que é Git

É um Sistema de Controle de Versão Distribuído (DVCS) [2, 4] . Ele regista mudanças feitas em um ou mais arquivos ao longo do tempo de forma que você possa recuperar versões específicas [2, 2] .

## Capítulo 2

# Configurando ambiente

Siga as instruções oficiais do git. Caso utilize o Windows, use o programa git bash instalado para os comandos da apostila.

# Capítulo 3

# Estados de arquivos

Arquivos no git podem estar em 1 dos 4 estágios: untracked, not staged, staged e committed[1].

## 3.1 Untracked

Arquivos que são encontrados pelo git, mas não versionados [7, 5] .

## 3.2 Not Staged

Arquivo que sofreu mudanças mas que ainda não foi consolidado na base de dados [2, 9] .

## 3.3 Staged

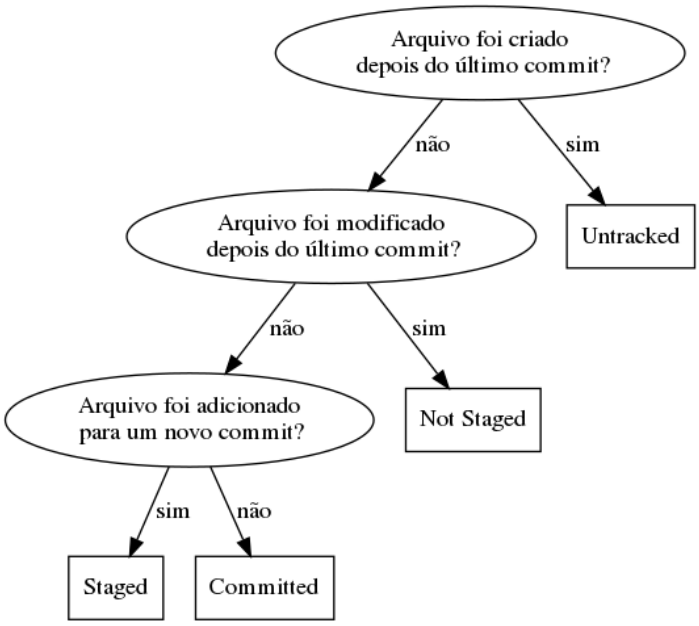
Arquivos que são encontrados pelo git, mas não versionados [7, 5] .

## 3.4 Committed

Arquivos commitados [2, 9] .

[1] Há divergência sobre os estados exigentes entre Wesley (untracked, staged e committed) [7, 5] e Scott (modified, staged e committed) [2, 9]

Figura 3.1: Estado do arquivo



# Capítulo 4

# Uso básico

Neste capítulo abordaremos o uso mais comum do git, os comandos usados com maior frequência. Todo comando é feito pelo pelo terminal (git bash no windows) e iniciadas com `git`, como por exemplo `git commit`. Muitos dos comandos precisarão ser especificados de acordo com a situação, como por exemplo adicionar nomes de arquivos ou remotes - nesses casos será específicado `<valor>` com essa notação. Em qualquer caso de dúvida sobre um comando, pode teclar `--help` para ver o que faz e opções, como abaixo.

Listing 4.1: Lê instruções sobre o comando add

```
1 git add --help
```

Antes de por a mão na massa precisamos configurar minimamente algumas opções da ferramenta (estas serão aprofundadas no Capítulo 8). Digite no git bash o seguinte:

Listing 4.2: Configuração inicial do git

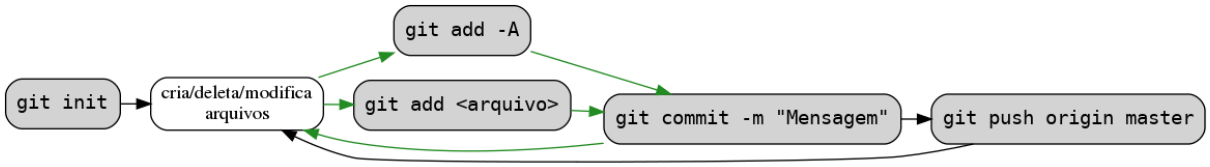
```
1 git config --global user.name "Seu Nome"
2 git config --global user.email "seuemail@mail.com"
3 git config --global color.ui true
4 git config --global core.editor "seu editor favorito"
```

As linhas 1 e 2 são informações que o git necessita para assinar em seu nome as modificações e a 3 habilita a colorização das respostas que alguns comandos podem retornar. A opção `core.editor` especifica qual editor de texto deve ser aberto para escrever commits (Subseção 4.1.4).

## 4.1 Individual

Comandos mais comuns utilizados quando não há mais pessoas no projeto.

Figura 4.1: Fluxo de trabalho básico do git




### 4.1.1 Iniciar repositório git

Todas mudanças realizadas em um projeto ficam armazenadas no diretório `.git` dentro do próprio projeto [7, 5] . Para possuir essa pasta, das duas uma, ou se clona um projeto (baixa projeto a partir do remoto - ver Subseção 4.1.5) ou inicializa git no repositório.

Listing 4.3: Clona repositório

```
1 git clone https://github.com/nenitf/apostilas
```

Listing 4.4: Clona repositório para determinada pasta

```
1 git clone https://github.com/nenitf/apostilas minhas-apostilas
```

Listing 4.5: Inicializa git dentro da pasta do projeto

```
1 git init
```

### 4.1.2 Status

Para verificar a situação dos arquivos do seu projeto (ver Capítulo 3), basta utilizar `git status` [2, 18] . Arquivos committed não aparecem.

### 4.1.3 Stagear arquivos

Para adicionar arquivos para mais tarde efetuar um commit basta usar `git add`.

Listing 4.6: Adiciona arquivos específicos

```
1 git add arquivo1.txt
2 git add arquivo2.txt arquivo3.txt
```

Listing 4.7: Stageando todos arquivos

```
1 git add -A
```

Listing 4.8: Removendo arquivos stageados

```
1 git rm --cached arquivo1.txt
```

### 4.1.4 Commit

O commit é o responsável por gravar as alterações dos arquivos stageados no repositório. É pelo histórico de commits que é possível retroceder até determinada versão do projeto. Os principais atributos que todo commit possui é um hash (identificador), data, autor e mensagem.

Listing 4.9: Commita rapidamente

```
1 git commit -m "Mensagem de commit"
```

#### 4.1.4.1 Mensagens de commits

Mensagens de commit devem ser claras para tornar fácil o entendimento do histórico do projeto (Subseção 4.1.8). O ideal é possuírem um título descritivo de até 50 caracteres, se necessário especificar mais, deve-se pular uma linha e descrever parágrafos com linhas de até 70 caracteres [2, 131] (caso prefira listar tópicos usar hifens, mas sempre respeitando o limite de caracteres por linha). Utilizar verbos no imperativo e tempo verbal de segunda pessoa: ao invés de ”Eu adicionei testes para”ou ”Adicionando testes para”, usar ”Adiciona testes para” [2, 131] . Abaixo exemplo de escrita de commit quando utilizado somente `git commit` em vez de `git commit -m "mensagem"`, retirado do livro Git Pro [2, ] .

Listing 4.10: Exemplo de commit

```
1 Breve (50 caracteres ou menos) resumo das mudanças
2
3 Texto explicativo mais detalhado, se necessário. Separe em linhas de
4 72 caracteres ou menos. Em alguns contextos a primeira linha é
```

```
tratada como assunto do e-mail e o resto como corpo. A linha em branco
separando o resumo do corpo écrítica (a não ser que o corpo não seja
incluído); ferramentas como rebase podem ficar confusas se você usar
os dois colados.

Parágrafos adicionais vem após linhas em branco.

- Tópicos também podem ser usados
- Tipicamente um hífen ou asterisco éutilizado para marcar tópicos,
precedidos de um espaço único, com linhas em branco entre eles, mas
convenções variam nesse item
```

Vale ressaltar que podem existir convenções para mensagens de commit no projeto, uma que se tornou famosa é do site conventionalcommits.org , cujo foi adaptada no projeto do angular .

### 4.1.5 Repositório remoto

O Github hospeda projetos git online [7, 14] . Quando um projeto possui base hospedada, este possui um `remote`. Ao criar um projeto o git não sabe identificar aonde ele será hospedado, portanto é necessário informar seu link com `git remote add <nome> <link>`. Um projeto pode ter inúmeros *remotes* e comumente o seu principal é chamado de `origin`.

Listing 4.11: Cria repositório remoto

```
git remote add origin <url.git>
```

### 4.1.6 Enviando código para repositório remoto

Listing 4.12: Enviando projeto e histórico de comits para o remoto origin na branch master

```
git push origin master
```

### 4.1.7 Trabalhando com tags

Tags são marcadores, usados para gerarem releases (versão do projeto) do projeto [7, 22] . Uma tag simplesmente aponta para um commit [7, 22] . Comumente nomes de tags são 3 digitos precedidos ou não por "v": `0.0.0` ou `v0.0.0`, essa convenção é embasada no conceito de versionamento semântico[1]. Ela consiste no primeiro dígito ser o *MAJOR* (relacionada a mudanças drásticas, quebra de compatibilidade com versões anteriores), depois o *MINOR* (funcionalidades que não causam quebras de contratos com versões anteriores) e por fim o *PATCH* (responsável por correção de bugs). Ficando então como `MAJOR.MINOR.PATCH`.

Listing 4.13: Cria tag localmente

```
git tag 1.0.0
```

Listing 4.14: Adiciona tags para o remote

```
git push <remote-name> master --tags
```

Listing 4.15: Visualiza tags

```
git tag
```

Listing 4.16: Visualiza tags ordenadas por data

```
git tag --sort=taggerdate
```

[1] www.semver.org

Listing 4.17: Apaga tag local

```
1 git tag -d <tag-name>
```

Listing 4.18: Apaga tag remota

```
1 git push <remote-name> :<tag-name>
```

### 4.1.8 Ver Histórico do projeto

Listing 4.19: Vê histórico de commits

```
1 git log
```

Listing 4.20: Vê histórico de commits e branches

```
1 git log --graph
```

Listing 4.21: Vê histórico de commits resumido

```
1 git log --pretty=oneline
```

Listing 4.22: Vê histórico das modificações de commits

```
1 git log -p
```

Listing 4.23: Vê histórico de commits dos arquivos modificados

```
1 git log --stat
```

#### 4.1.8.1 Alterando histórico do projeto

Listing 4.24: Stageando todos arquivos

```
1 git rebase -i
```

### 4.1.9 Retrocedendo commits

Listing 4.25: Voltando versões a partir do hash (É criada uma branch temporária)

```
1 git checkout <hash-do-commit>
```

Listing 4.26: Volta um commit (preserva modificações dos commits posteriores na área stageada)

```
1 git reset HEAD~1 --soft
```

Listing 4.27: Volta dois commits (preserva modificações dos commits posteriores na área stageada)

```
1 git reset HEAD~2 --soft
```

Listing 4.28: Volta um commit (perde modificações dos commits posteriores)

```
1 git reset HEAD~1 --hard
```

## 4.2 Em time

Complemento dos comandos mais utilizados, comumente utilizados por estar com mais pessoas em um projeto.

### 4.2.1 Atualizar código local de acordo com remoto

Quando se está trabalhando em um time é necessário verificar se há código novo a ser atualizado. O `git pull` faz exatamente isso, verifica se o código local (histórico de commits) é o mesmo do remoto, caso não ele tentará atualizar.

Listing 4.29: Stageando todos arquivos

```
1 git pull
```

Listing 4.30: Adiciona todos arquivos

```
1 git fetch
```

### 4.2.2 Branches

Branches são ramificações do desenvolvimento [7, 11] . É recomendado criar nova branch a cada nova funcionalidade sendo desenvolvida [7, 12] .

Listing 4.31: Cria branch

```
1 git branch nova-branch
```

Listing 4.32: Cria branch e acessa ela

```
1 git checkout -b <nova-branch>
```

Listing 4.33: Cria branch local a partir da remota e a acessa

```
1 git checkout -b <branch-remota> <remote>/<branch-remota>
```

Listing 4.34: Troca de branch

```
1 git checkout <branch-existente>
```

Listing 4.35: Lista branches locais

```
1 git branch
```

Listing 4.36: Lista branches locais e remotas

```
1 git branch -a
```

Listing 4.37: Apaga branch local

```
1 git branch -d <branch-existente>
```

Listing 4.38: Apaga branch remota

```
1 git push <remote-name> :<branch>
```

#### 4.2.2.1 Juntar branches localmente

- Com `git merge` é criado um commit adicional caso ambas branches possuam commits em distintos.
- Com `git rebase` não é criado um commit adicional na mesma situação do `git merge`.

Listing 4.39: Mescla branches

```
1 git merge <branch>
```

Listing 4.40: Mescla branches

```
1 git rebase <branch>
```

#### 4.2.2.2 Conflito

Em caso de conflito ¡¡¡¡ HEAD AAAA ===== BBBB ¿¿¿¿ master

#### 4.2.2.3 Contribuindo para projetos abertos

# Capítulo 5

# Fluxos de trabalho

## 5.1 Comum

## 5.2 Gitflow

Gitflow é uma extensão do git, também pode ser considerado uma filosofia de trabalho [3, 1] . É uma metodologia que especifica comandos do git para serem aplicados no contexto da modificação do projeto [3, 2] - também possui cli que abstrai o git[1].

Com gitflow teremos duas branches principais no remoto, a *master* e a *develop* [3, 2] - a primeira só é alterada através de merge (normalmente com a develop) e a segunda serve para preparar o código da próxima para a master [3, 2] . Demais branches podem e serão criadas, porem todas temporárias que devem ser apagadas, elas podem ser de: features, releases, hotfixes, bugfixes ou supports [3, 2] .

### 5.2.1 Dissecando gitflow

Cada comando do gitflow terá seu funcionamento explicado e ”convertido”para comandos do git.

#### 5.2.1.1 Init

- Inicializa git;
- Especifica branch de produção (master comumente).
- Especifica branch de produção (master comumente).

## 5.3 Pessoal para estudos

[1] danielkummer.github.io/git-flow-cheatsheet

## Capítulo 6

# Resumo dos comandos (cheatsheet)

# Capítulo 7

# Exemplos de cenários específicos

## 7.1 Exemplo 1

### 7.1.1 TLDR

### 7.1.2 Resolução

## 7.2 Exemplo 2

### 7.2.1 TLDR

### 7.2.2 Resolução

## 7.3 Exemplo 3

### 7.3.1 TLDR

### 7.3.2 Resolução

# Capítulo 8

# Configurações

## 8.1 Usuário

### 8.1.1 Gitconfig

#### 8.1.1.1 User

#### 8.1.1.2 Colors

#### 8.1.1.3 Template

#### 8.1.1.4 Alias

### 8.1.2 SSH

Listing 8.1: Gera senha

```
1 ssh-keygen
```

Será criado a pasta .ssh contendo os arquivos *id_rsa* e *id_rsa.pub*. A primeira é privada e nunca deve ter seu conteúdo compartilhado, a segunda é a pública, no qual pode ser compartilhada [7, 15] .

## 8.2 Projeto

### 8.2.1 Gitignore

Para fazer o git ignorar arquivos (normalmente de configuração de IDE, logs e etc) é criado na raiz do projeto o arquivo `.gitignore`.

Listing 8.2: Exemplo de .gitignore

```
1  # latex
2  *.aux
3  *.bbl
4  *.blg
5  *.fls
6  *.lof
7  *.log
8  *.lol
9  *.lot
10
11 # python
12 __pycache__/
13
14 # npm
15 .DS_Store
16 node_modules
```

```
17
18 # php
19 .phpunit.result.cache
20 composer.lock
21 vendor/
```

### 8.2.2 Gitconfig

### 8.2.3 Remotos

#### 8.2.3.1 Readme

#### 8.2.3.2 Template de issue e PR

#### 8.2.3.3 Licença

# Capítulo 9

# Ferramentas

## 9.1 Extensões de Editores/IDEs

### 9.1.1 Vim/Neovim

#### 9.1.1.1 vim-fugitive

#### 9.1.1.2 gv.vim

#### 9.1.1.3 vim-gitgutter

## 9.2 GUI

# Bibliografia


[1] A. Aquiles and R. Ferreira. *Controlando versões com Git e Github*. Casa do Código, 2016.

[2] S. Chacon. *Pro-Git*. Apress, 5 2014. Traduzido por Eric Douglas, disponível em:.

[3] L. C. Diniz. Git flow na prática v2. https://www.schoolofnet.com/curso/git/controle-de-versao/git-flow-na-pratica-v2/, 2017. Curso Online.

[4] Git. *Git Documentation*. http://git-scm.com.

[5] W. Justen. Git e github na vida real. https://www.udemy.com/git-e-github-na-vida-real/, 2018. Curso Online.

[6] W. Justen. Git e github para iniciantes. https://www.udemy.com/git-e-github-para-iniciantes/, 2018. Curso Online.

[7] W. Willians. Git e github. https://www.schoolofnet.com/curso/git/controle-de-versao/git-e-github/, 2015. Curso Online.