

PROGRAMAÇÃO BACK END II

Maurício de Oliveira Saraiva



SOLUÇÕES
EDUCACIONAIS
INTEGRADAS



Uso de um sistema de controle de versões distribuído

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Usar os comandos `commit`, `status`, `add` e `log`.
- Transferir o repositório remoto para o repositório local.
- Atualizar o repositório local.

Introdução

A colaboração entre desenvolvedores que atuam em projetos compartilhados requer o uso de um *software* de controle de versão adequado. O Git é um sistema distribuído que permite manipular o repositório local e sua conexão com repositórios remotos de modo fácil e simples.

Entre as características do uso do Git está sua interface por *prompt* de comando, que permite que desenvolvedores entrem com instruções escritas, que executam as principais tarefas de gerenciamento de repositórios.

Neste capítulo, você irá estudar sobre os principais comandos para manipular um repositório local com Git, assim como aprenderá a transferir e atualizar repositórios.

Comandos `commit`, `status`, `add` e `log`

O Git é um sistema de controle de versão distribuído que permite controlar as modificações em projetos de desenvolvimento de *software*. Sua forma de atuação envolve a cópia completa de todo o projeto em cada estação de trabalho das pessoas que participam deste projeto (GIT HANDBOOK, 2017).

Isso significa que todos os desenvolvedores terão uma cópia do projeto localmente, mas que poderão enviar e/ou receber as modificações feitas nos arquivos de código-fonte a qualquer momento, por meio de comandos específicos.

Existem diversos comandos no Git que realizam operações com os arquivos de um repositório existente. Por meio desses comandos, o Git permite que desenvolvedores adicionem arquivos (`add`), visualizem o estado de arquivos (`status`), consolidem arquivos (`commit`) e acessem o histórico das modificações que foram consolidadas (`log`).

Criando um repositório

Criar um repositório Git é o primeiro passo para iniciar o versionamento de arquivos de um projeto de *software*. A criação de um repositório Git por *prompt* de comando é uma atividade bastante simples.

Assim que o sistema Git for instalado, acesse a pasta do diretório, por *prompt* de comando, e digite a seguinte instrução:

```
git init
```

Assim, um repositório é criado e uma pasta `.git` é adicionada, contendo todas as informações que o repositório precisa para controlar as versões do projeto.



Link

Acesse o *link* a seguir para informações sobre como baixar e instalar o Git, assim como configurá-lo por *prompt* de comando.

<https://qr.go.page.link/Sc5hF>

Comando `add`

A inclusão de arquivos no monitoramento de um repositório local deve ser realizada de modo explícito, pois o sistema Git dá ao desenvolvedor a liberdade de incluir somente os arquivos que achar interessante no versionamento.

Utilize o comando `add` para iniciar o monitoramento de novos arquivos que foram criados ou depositados nas pastas do repositório local, assim como para preparar os arquivos que já estão sendo monitorados e que foram modificados após determinado `commit` (GIT..., 2019a).

São exemplos de execução do comando `add` pelo *prompt* de comando:

```
git add helloWorld.c
git add *.c
git add .
```

O primeiro comando adiciona o arquivo `helloWorld.c` ao repositório. O segundo comando inclui todos os arquivos de código-fonte da linguagem C, por meio de sua extensão. Já o terceiro comando adiciona todos os arquivos existentes na pasta ao monitoramento.

Nenhuma saída é apresentada na execução deste comando, mas é possível verificar o estado do monitoramento dos arquivos modificados por meio do comando `status`.

Comando `status`

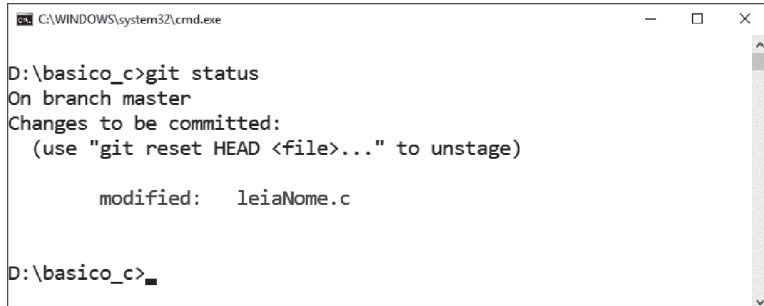
O comando `status` pode ser utilizado para verificar o estado em que se encontram os arquivos depositados no repositório local. Quando um repositório for criado e nenhum arquivo for adicionado, ou quando já existirem arquivos no repositório que não foram modificados, o estado indica que não há nada para realizar no comando `commit`.

Entretanto, o `status` apresentará uma relação de arquivos se houver arquivos novos, criados após a criação do repositório, ou arquivos que foram modificados após a realização de um `commit`.

Para visualizar o *status* dos arquivos do repositório local, entre com a seguinte instrução no *prompt* de comando (GIT..., 2019h):

```
git status
```

O resultado, ilustrado na Figura 1, apresenta o arquivo `leiaNome.c` que foi modificado após a última versão consolidada e preparado por meio do comando `add`, mas que ainda não foi inserido no comando `commit`.



```
C:\WINDOWS\system32\cmd.exe

D:\basico_c>git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   leiaNome.c

D:\basico_c>
```

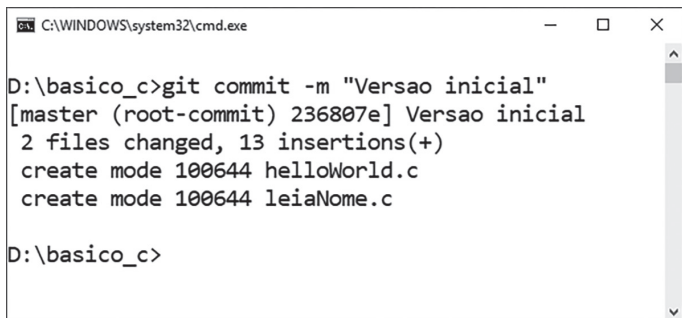
Figura 1. Resultado da execução do comando *status*.

Comando `commit`

Uma vez que os arquivos modificados já tenham sido adicionados por meio do comando `add`, o repositório local estará preparado para consolidar as mudanças — criar um *snapshot*. Isso significa que o Git criará uma versão da situação atual desses arquivos modificados como uma espécie de fotografia, registrando como eles estavam naquele exato momento (GIT..., 2019c).

Para efetuar a consolidação dos arquivos de um repositório, entre com a instrução `commit` no *prompt* de comando conforme a seguir:

```
git commit -m "Versao inicial"
```



```
C:\WINDOWS\system32\cmd.exe

D:\basico_c>git commit -m "Versao inicial"
[master (root-commit) 236807e] Versao inicial
 2 files changed, 13 insertions(+)
 create mode 100644 helloWorld.c
 create mode 100644 leiaNome.c

D:\basico_c>
```

Figura 2. Execução do comando *commit*.

A Figura 2 apresenta a consolidação que foi realizada, incluindo os arquivos que haviam sido modificados: `helloWorld.c` e `leiaNome.c`. O parâmetro `-m` define uma mensagem que pode servir como identificação para a versão que foi criada, por exemplo, “Versão inicial” quando se tratar do primeiro `commit` realizado, ou “Implementação do *sprint* #3”, que se refere a alguma implementação específica.



Fique atento

Arquivos modificados após determinada consolidação, que não forem adicionados pelo comando `add`, não entrarão no versionamento executado pelo `commit` atual. Contudo, é possível pular o passo de adicionar os arquivos antes de executar `commit`. Para isso, utilize a opção `-a`:

```
git commit -a -m "mensagem do commit".
```

O comando `commit` cria uma versão do projeto no repositório local. Se houver interesse em publicar essa versão no GitHub, é preciso utilizar o comando `push`.



Link

Acesse o *link* a seguir e saiba mais sobre o comando `push` do Git.

<https://qr.go.page.link/fXUyq>

Comando `log`

Normalmente, os projetos de desenvolvimento de sistemas, gerenciados por *software* de controle de versão, possuem diversas versões produzidas ao longo do tempo por meio dos *commits* que foram realizados.

No entanto, pode ser necessário consultar o histórico desses *commits* para se obter mais detalhes, como a data em que foi realizado e o responsável pela execução. O comando utilizado para acessar a essas informações é `log`, conforme apresentado a seguir (GIT..., 2019f).

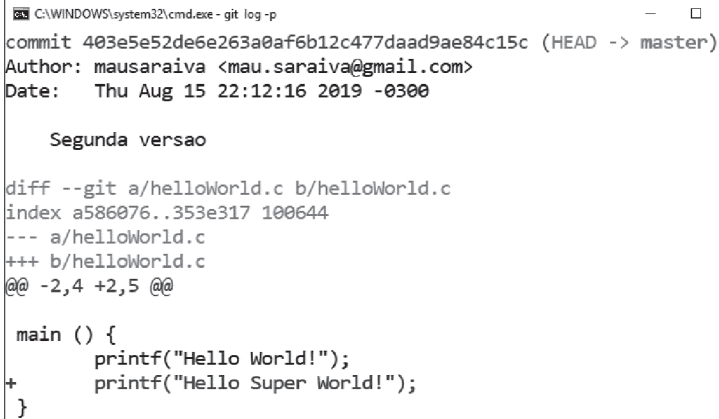
```
git log
```

A Figura 3 apresenta o histórico de *commits* realizado no repositório utilizado como modelo. Pode-se observar um identificador do *commit*, assim como o autor, a data, a hora e a descrição. Neste `log` é possível identificar que três *commits* foram realizados: a versão inicial e a segunda versão, alguns minutos após.



O parâmetro `-p` do comando `log` apresenta as diferenças encontradas nos arquivos que foram modificados. A Figura 4 apresenta as diferenças encontradas no arquivo `helloWorld.c`: a inclusão da linha `printf("Hello Super World!")`.

```
git log -p
```



```

C:\WINDOWS\system32\cmd.exe - git log -p
commit 403e5e52de6e263a0af6b12c477daad9ae84c15c (HEAD -> master)
Author: mausaraiva <mau.saraiva@gmail.com>
Date: Thu Aug 15 22:12:16 2019 -0300

    Segunda versao

diff --git a/helloWorld.c b/helloWorld.c
index a586076..353e317 100644
--- a/helloWorld.c
+++ b/helloWorld.c
@@ -2,4 +2,5 @@

main () {
    printf("Hello World!");
+   printf("Hello Super World!");
}

```

Figura 4. Diferenças no arquivo `helloWorld.c`.

Transferência do repositório remoto para o local

No repositório remoto (GitHub) de um projeto, fica armazenada sua coleção de arquivos e pastas, contendo todas as informações sobre as alterações que foram realizadas, os responsáveis pelas modificações e quando estas foram feitas.

Esse repositório remoto do GitHub pode ser copiado/clonado para um repositório local (Git) por meio de uma operação chamada `clone`. Essa operação traz para o computador local uma cópia de todos os arquivos e pastas do projeto, juntamente com todos os *commits* que foram realizados e registrados pelo controle de versão (GIT..., 2019b).

Um repositório local que recebe os arquivos e as pastas de um projeto remoto, por meio de uma clonagem, atua como uma cópia de segurança — *backup*. Isso significa que no caso de alguma pane no servidor remoto, o repositório local conterá o histórico de modificações de todos os arquivos do projeto.



Fique atento

O GitHub fornece duas opções para baixar um projeto: por clonagem e por *download zip*. A diferença entre estas opções é que a clonagem traz todas as versões (*commits* e *branches*) e o *download zip* traz apenas os arquivos e as pastas da última versão.

A Figura 5 apresenta um repositório remoto do GitHub, chamado *pulse-animation*, que contém as pastas *normal* e *withJS*.

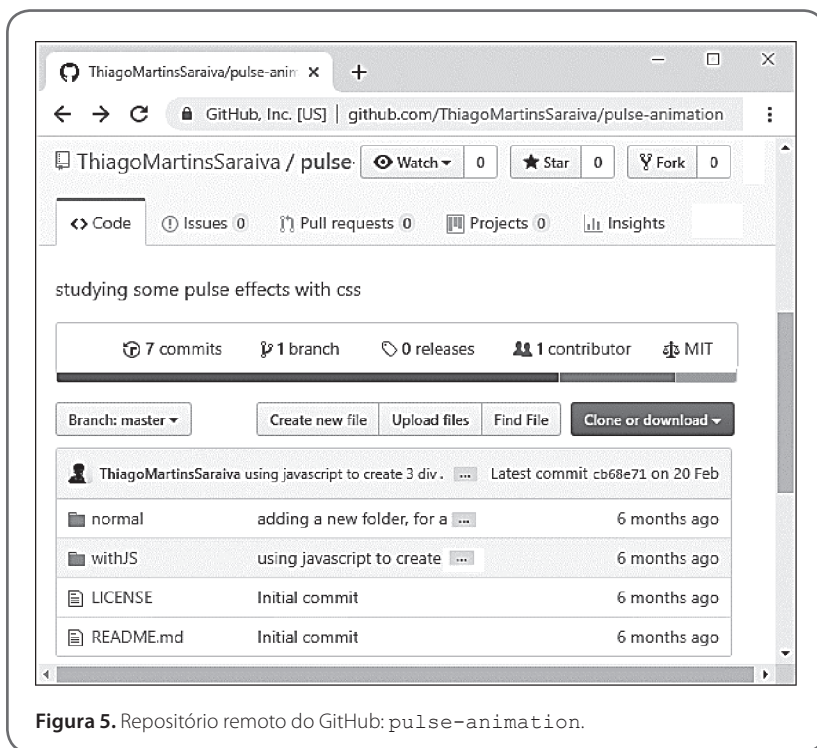
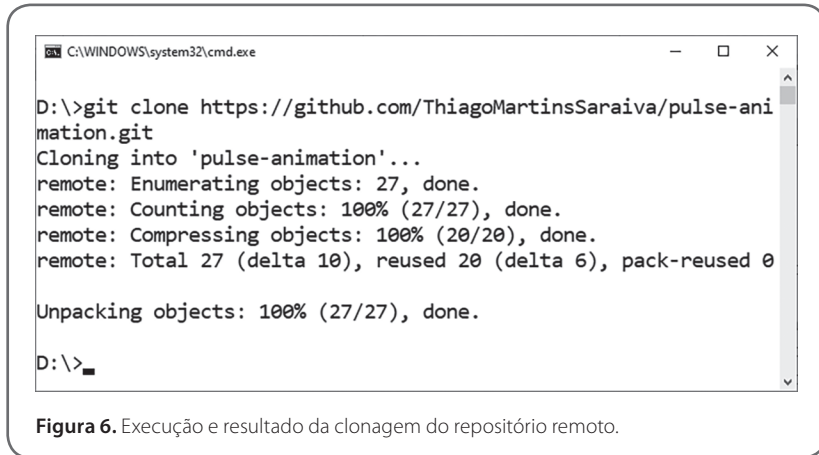


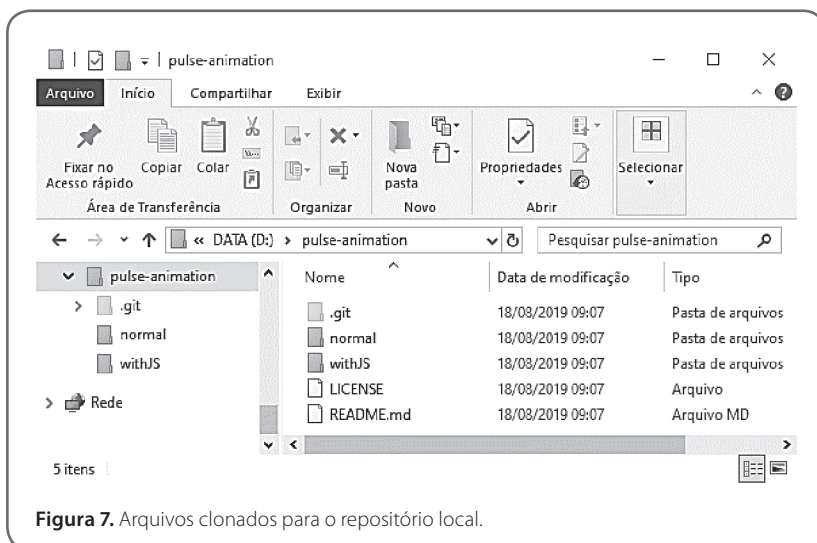
Figura 5. Repositório remoto do GitHub: *pulse-animation*.

Para clonar este repositório remoto por linha de comando, abra uma janela de terminal e entre com a seguinte instrução:

```
git clone https://github.com/ThiagoMartinsSaraiva/pulse-animation.git
```



Após a execução do comando, todas as pastas e os arquivos do repositório remoto estarão no repositório local, conforme ilustra a Figura 7. O histórico de versões, incluindo todos os *commits* que foram realizados, pode ser conferido por meio do comando `git log`.



Atualização do repositório local

Vimos que a clonagem de um repositório remoto traz todos os arquivos e as pastas para o repositório local. No entanto, em determinado momento será necessário trazer apenas o que já foi modificado por outros colaboradores no repositório remoto, pois você já estará trabalhando no repositório local que havia sido clonado.

Os repositórios remotos vinculados aos repositórios locais podem ser visualizados por meio do comando `remote`. Para trazer as atualizações do repositório remoto para o repositório local, dois comandos podem ser utilizados: `fetch` e `pull`.

Comando `remote`

Este comando visualiza a lista de remotos e permite adicionar novos remotos. Quando um clone é realizado, automaticamente o Git coloca o nome de `origin` para o repositório (GIT..., 2019g). A instrução a seguir apresenta o repositório remoto que foi clonado anteriormente, conforme ilustrado na Figura 8:

```
git remote -v
```



Figura 8. Relação de repositórios remotos.

Para incluir o repositório remoto chamado `angular/angular` do GitHub, entre com a seguinte instrução no *prompt* de comando, informando um nome para ele e seu endereço/caminho. Note que o parâmetro `angular` após `add` serve como um apelido para a vinculação dos repositórios, e não precisa necessariamente ser idêntico ao nome do repositório:

```
git remote add angular https://github.com/angular/angular
```

Isso faz com que `angular` seja incluído na relação de repositórios remotos que podem ser manipulados pelo comando `remote`. Além disso, é possível excluir e renomear repositórios remotos da lista, por meio dos parâmetros `rm` e `rename`, respectivamente, da instrução `remote`.

Comando `fetch`

O comando `fetch` busca todos os dados do repositório remoto que ainda não constam no seu repositório local. No entanto, ele não realiza o *merge* automaticamente com os seus arquivos do repositório local. Se for necessário fazer o *merge*, você deverá fazê-lo manualmente (GIT..., 2019d).

Para rodar o comando `fetch`, utilize a seguinte instrução no *prompt* de comando:

```
git fetch origin
```

Note que `origin` representa o repositório `pulse-animation` que foi clonado anteriormente. A instrução também poderia usar o endereço do repositório.



Link

Acesse o *link* a seguir para mais informações sobre como trabalhar com *branches* e *merge* no Git.

<https://qr.go.page.link/hnvjU>

Comando `pull`

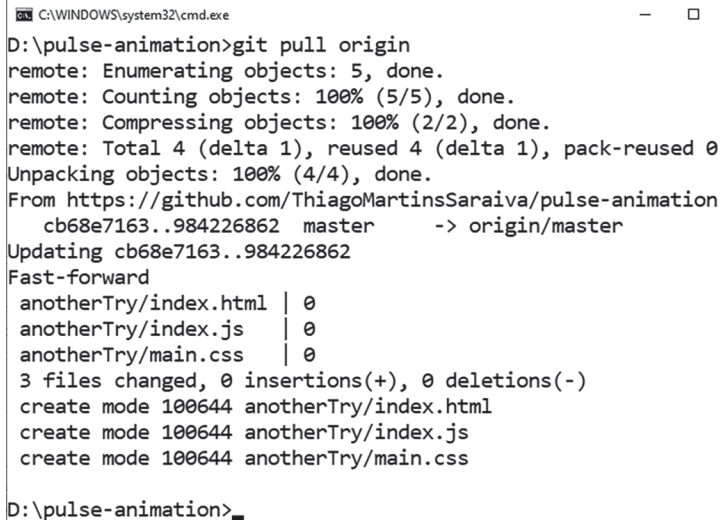
Assim como `fetch`, o comando `pull` traz as atualizações do repositório remoto para o repositório local. A diferença é que `pull` faz o *merge* automaticamente com os dados do repositório local (GIT..., 2019f).

Quando um repositório remoto é clonado, o Git automaticamente vincula o *branch* `master` desses repositórios, configurando a instrução `pull` para fazer o *download* e o *merge* desse *branch* automaticamente, facilitando o trabalho do desenvolvedor.

Para executar o comando `pull`, utilize a seguinte instrução no *prompt* de comando, na qual pode-se utilizar tanto o nome do repositório quanto o seu endereço:

```
git pull origin
```

A Figura 9, que ilustra o resultado do comando `pull`, apresenta as modificações que foram baixadas do repositório remoto após a execução do clone. Nesta atualização, três arquivos foram modificados. Estas modificações podem ser visualizadas por meio do comando `git log`.



```
D:\pulse-animation>git pull origin
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 1), reused 4 (delta 1), pack-reused 0
Unpacking objects: 100% (4/4), done.
From https://github.com/ThiagoMartinsSaraiva/pulse-animation
   cb68e7163..984226862  master    -> origin/master
Updating cb68e7163..984226862
Fast-forward
   anotherTry/index.html | 0
   anotherTry/index.js   | 0
   anotherTry/main.css   | 0
   3 files changed, 0 insertions(+), 0 deletions(-)
   create mode 100644 anotherTry/index.html
   create mode 100644 anotherTry/index.js
   create mode 100644 anotherTry/main.css
D:\pulse-animation>
```

Figura 9. Resultado da execução do comando `pull`.



Referências

GIT - git-add Documentation. *Git*, [S. l.], 9 Aug. 2019a. Disponível em: <https://git-scm.com/docs/git-add>. Acesso em: 26 ago. 2019.

GIT - git-clone Documentation. *Git*, [S. l.], 9 Aug. 2019b. Disponível em: <https://git-scm.com/docs/git-clone>. Acesso em: 26 ago. 2019.

GIT - git-commit Documentation. *Git*, [S. l.], 9 Aug. 2019c. Disponível em: <https://git-scm.com/docs/git-commit>. Acesso em: 26 ago. 2019.

GIT - git-fetch Documentation. *Git*, [S. l.], 9 Aug. 2019d. Disponível em: <https://git-scm.com/docs/git-fetch>. Acesso em: 26 ago. 2019.

GIT - git-log Documentation. *Git*, [S. l.], 9 Aug. 2019e. Disponível em: <https://git-scm.com/docs/git-log>. Acesso em: 26 ago. 2019.

GIT - git-pull Documentation. *Git*, [S. l.], 9 Aug. 2019f. Disponível em: <https://git-scm.com/docs/git-pull>. Acesso em: 26 ago. 2019.

GIT - git-remote Documentation. *Git*, [S. l.], 9 Aug. 2019g. Disponível em: <https://git-scm.com/docs/git-remote>. Acesso em: 26 ago. 2019.

GIT - git-status Documentation. *Git*, [S. l.], 9 Aug. 2019h. Disponível em: <https://git-scm.com/docs/git-status>. Acesso em: 26 ago. 2019.

GIT Handbook. *GitHub*, San Francisco, 25 Sep. 2017. Disponível em: <https://guides.github.com/introduction/git-handbook/>. Acesso em: 26 ago. 2019.

Encerra aqui o trecho do livro disponibilizado para esta Unidade de Aprendizagem. Na Biblioteca Virtual da Instituição, você encontra a obra na íntegra.

Conteúdo:

