treinadev

Guia básico de Git

CAMPUS CODE

Guia básico de Git

Introdução

Normalmente, profissionais que trabalham com desenvolvimento de software utilizam no seu dia a dia alguma ferramenta de controle de versão para ajudá-los a criar novas versões e rastrear o histórico de alterações do seu código, sem a necessidade de fazer cópias manualmente.

Existem várias ferramentas de controle de versão, tais como:

- Git
- Subversion
- Mercurial
- Merant PVCS
- Visual Source Safe

Cada uma delas possui suas vantagens e desvantagens, mas não vamos falar sobre elas. Aqui, vamos tratar especialmente do Git, um sistema usado para controlar as versões do código do Kernel do Linux que se tornou extremamente popular entre os programadores. Um fato interessante é que tanto Git quanto Linux foram criados pela mesma pessoa: Linus Torvalds.

Existem aplicações com interface visual para controle de Git, cada uma com suas características particulares, mas aqui vamos focar nos comandos Git para o Terminal.

O intuito é apenas apresentar os principais conceitos básicos de forma que você seja capaz de conhecer o fluxo de trabalho inicial para trabalhar com Git. Veremos alguns comandos como: `config`, `add`, `commit` e `push`.

Instalação

O primeiro passo é instalar o Git. Se você estiver usando um Sistema Operacional como o Ubuntu (ou outra distribuição de Linux que use `apt-get` como gerenciador de pacotes), a instalação pode ser feita por meio do Terminal com o comando abaixo:

```
$ apt-get install git
```

Para instalação no Mac e no Windows, você pode seguir as instruções no link: Primeiros passos – Instalando Git.

Configuração de usuário

Após instalar o Git na sua máquina, chegou a hora de se identificar, isto é, dizer ao Git quem você é para que ele saiba reconhecer quem fez o registro do código. Para isso, use os seguintes comandos no Terminal:

```
$ git config --global user.name "Seu nome"
$ git config --global user.email "seu e-mail"
```

Colocando a opção `--global` estamos dizendo que, para todos os repositórios Git nessa máquina, valerá essa opção, a menos que ela seja redefinida dentro do diretório de um repositório específico. Por exemplo, se para um determinado projeto você estiver usando um outro usuário, com e-mail diferente, é possível configurar esse diretório específico com o comando:

```
$ git config --local user.name "Seu nome"
$ git config --local user.email "seu e-mail"
```

Para verificar qual e-mail e nome de usuário estão configurados para o repositório corrente, basta usar o comando:

```
$ git config user.name
$ git config user.email
```

Os comandos retornam, respectivamente, o nome e o e-mail configurados.

Estrutura de um comando Git

Sempre que falarmos de um comando, a estrutura será a seguinte:

- comando `git`;
- nome do comando git que queremos executar, por exemplo: `git config`;
- opções, por exemplo: `git config --global`, explicado acima;
- argumentos, por exemplo: `git config user.email`, para mostrar o e-mail definido.

Se quiser ver todas as configurações definidas na sua máquina, use o comando:

```
$ git config --list
```

ou

```
$ git config --global --list
```

Obtendo ajuda

No Git é muito fácil conseguir mais informações sobre o funcionamento de algum comando específico:

```
$ git help
```

Esse comando mostra uma lista geral com todos os comandos e o que eles fazem. Se precisar de ajuda para um comando específico, basta dar `git help` seguido do comando específico que você quiser pesquisar.

```
$ git help config
```

Mostra uma descrição detalhada das opções para o comando específico citado, nesse caso o `config`.

Iniciando um repositório

Sugerimos que você crie um diretório na sua pasta HOME para guardar seus projetos e o utilize para seus estudos em programação. Crie o diretório `workspace` (caso ainda não tenha) e dentro dele crie um diretório `meu_projeto`.

```
$ cd ~
$ mkdir workspace
$ cd workspace
$ mkdir meu_projeto
```

Neste diretório, vamos dar o comando `git init`, que vai dizer ao Git que queremos que ele comece a monitorar os arquivos nesta pasta:

```
$ cd meu_projeto
$ git init
$ ls -a
. .. .git
```

Como pode ver, o Git criou um subdiretório `.git`. Esse subdiretório tem tudo que o Git precisa para trabalhar e indica que está sendo monitorado pelo Git.

Dependendo da versão do Git que você estiver usando, é possível que tenha aparecido a seguinte mensagem depois de rodar o comando `git init`:

```
$ git init
hint: Using 'master' as the name for the initial branch. This default
branch name
hint: is subject to change. To configure the initial branch name to
use in all
hint: of your new repositories, which will suppress this warning,
call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk'
and
hint: 'development'. The just-created branch can be renamed via this
command:
hint:
hint:   git branch -m <name>
```

No momento em que o comando de inicialização acima é rodado em um diretório, por padrão o Git precisa dar um nome para a ramificação principal em que o histórico de versões será armazenado. Originalmente, o nome dado a essa ramificação era `master`. Atualmente, considerando o significado histórico do termo, atrelado à escravidão, a comunidade tem dado preferência para outros termos, como `main`, por exemplo. A mensagem mostra como podemos renomear a ramificação e como configurar o nome padrão, para que seja sempre utilizado esse nome ao inicializar o Git em um diretório.

Você pode conferir o nome atual rodando o comando:

```
$ git branch --show-current
master
```

Vamos seguir a recomendação adotada pela comunidade e trocar o nome para `main` rodando o comando:

```
$ git branch -m main
$ git branch --show-current
main
```

Se o nome padrão já estiver configurado como `main` no seu computador, não é necessário fazer essa alteração. No momento o que você precisa saber é que a ramificação principal será chamada de `main` e é lá que será armazenado o histórico de modificações de código do diretório em que estamos trabalhando. Vamos falar mais sobre ramificações e como trabalhar com elas no final desta apostila.

Adicionando arquivos e fazendo o *commit*

Aposto que você mal pode esperar para ver como o Git controla a versão dos arquivos do seu projeto. Primeiro vamos criar um arquivo `README.md` com o comando `touch` e ver o que o Git nos diz com o comando `git status`:

```
$ touch README.md
$ git status
On branch main

Initial commit

Untracked files:
        README.md

nothing added to commit but untracked files present
```

Ele nos diz que há um arquivo que ainda não está sendo rastreado pelo Git (*untracked*). Em outras palavras, o Git ainda não possui um registro histórico desse arquivo. Então agora vamos adicioná-lo à área de seleção do Git:

```
$ git add README.md
$ git status

On branch main

Initial commit

Changes to be committed:
        new file:   README.md
```

Agora ele nos diz que há mudanças a serem registradas, sendo assim vamos fazer o *commit,* ou “comitar” como *devs* costumam dizer!

```
$ git commit -m "Meu primeiro commit"
$ git status

On branch main
nothing to commit, working directory clean
```

Pronto, fizemos nosso primeiro *commit* no Git! O comando `git commit` grava nossas modificações e a opção `-m` é para o Git receber uma mensagem para nosso *commit*. Sem mensagem o Git reclama!

Visualizando *commits*

Para ver o nosso *commit*, podemos usar o comando `git log`, que mostra todo o histórico de *commits* desse projeto:

```
$ git log
commit 405ef53eec6995d6be608d5b419774eed321adcf
Author: Seu nome <email@gmail.com>
Date:   Mon Apr  4 17:58:17 2015 -0300

    Meu primeiro commit
```

Digite `q` para sair da lista de registros do log.

Muito simples até aqui, não é? Agora poderemos ver o poder do controle de versão.

Vamos abrir o arquivo `README.md` e fazer algumas alterações. Você pode abrir o arquivo em qualquer editor de texto. Se você tiver o Atom instalado, pode usar o seguinte comando, por exemplo:

```
$ atom README.md
```

Altere o arquivo e salve. Em seguida, volte ao Terminal e dê o comando:

```
$ git status
On branch main
Changes not staged for commit:
        modified:   README.md

no changes added to commit
```

Veja que interessante, o Git nos diz que há mudanças que ainda não foram selecionadas para *commit*, sendo assim, vamos adicioná-las com o `git add`:

```
$ git add README.md
$ git status
On branch main
Changes to be committed:
        modified:   README.md
```

E agora o arquivo está pronto para um novo *commit*:

```
$ git commit -m "Altera o README.md"
```

Se você esquecer o trecho `-m` com a mensagem do *commit*, o Git vai abrir o editor definido em `git config core.editor` e solicitar a inclusão de uma mensagem. Em muitos casos, o editor padrão é o vim, mas podemos alterá-lo para o nano, por exemplo, com o seguinte comando:

```
$ git config core.editor
core.editor=vim
$ git config core.editor "nano"
```

Altere mais uma vez o arquivo `README.md` e faça o *commit* com o comando:

```
$ git add README.md
$ git commit
```

Isso vai abrir o editor nano. Escreva sua mensagem de *commit*, salve com `Ctrl + x` e depois aperte `y` para confirmar.

Até aqui, vimos como instalar o Git e aprendemos como registrar as alterações realizadas no nosso repositório. Mas o Git é muito mais poderoso que isso. Ele também permite enviar as alterações para um repositório remoto, retornar o código ao estado de um *commit* anterior, criar ramificações de um projeto, entre outras funcionalidades. A seguir, vamos falar sobre a comunicação entre nossos computadores e os repositórios remotos.

Chave SSH

Para garantir a comunicação segura entre seu computador e os repositórios remotos Git – como o GitHub e o GitLab –, você precisa configurar chaves SSH. É muito fácil configurá-las e tem diversos tutoriais por aí. Vamos deixar links no final do texto para você consultar.

Novamente, vamos para a linha de comando. Acostume-se com isso!

Abra seu Terminal e dê o comando: `ssh-keygen`. Assim que o comando é executado, deve aparecer a seguinte mensagem:

```
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/seu_user/.ssh/id_rsa):
```

O Terminal está perguntando onde você gostaria de salvar o arquivo das chaves. Normalmente utilizamos o local padrão, para isso você pode simplesmente apertar `ENTER`. Em seguida, uma nova mensagem pedindo para que uma senha seja configurada, mas você pode deixá-la em branco, pressionando `ENTER`. Caso você opte por uma senha, apenas a repita e aperte `ENTER` novamente.

```
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

Finalmente uma mensagem com sua chave é apresentada:

```
Your identification has been saved in /Users/seu_user/.ssh/id_rsa.
Your public key has been saved in /Users/seu_user/.ssh/id_rsa.pub.
The key fingerprint is:
4a:a5:34:59:d5:a5:56:29:d3:5e:6e:12:e8:7b:74:da seu_user@sua_maquina
The key's randomart image is:
+--[ RSA 2048]----+
|     o+o+o+o.o |
|     .....=o o o|
|           oo . o |
|     . .  o . .|
|     . S   o + |
|     . .   . o E|
|     .     .   |
|                 |
|                 |
+-----------------+
```

Resumindo os passos acima para criar a chave SSH:

- Rode o comando `ssh-keygen`.
- Para o caminho da chave, dê somente enter. O caminho apresentado é o padrão.
- Para senha, deixe em branco ou digite uma caso queira mais segurança ao utilizar a chave.
- Repita a senha.

Pronto, sua chave está criada.

Agora precisamos adicioná-la no serviço Git que escolher. Aqui, vamos descrever o processo no GitHub. Depois de fazer o login, clique na imagem do seu perfil e em seguida em *Settings*. Na barra lateral, clique em *SSH and GPG keys* e depois em *New SSH key*. Dê um nome para a chave para que você consiga identificar onde ela está sendo usada, como por exemplo "Trabalho".

No Terminal, digite:

```
$ cat ~/.ssh/id_rsa.pub
```

Copie **todo** o conteúdo desse arquivo, ele irá aparecer no Terminal, e cole no campo *key*. Clique em *Add SSH key*.

No Windows

Para copiar o conteúdo do arquivo RSA no Windows, você pode usar o comando abaixo:

```
$ clip < ~/.ssh/id_rsa.pub
```

Git remoto

Já entendemos os conceitos básicos de Git, agora é hora de entender como compartilhar nosso código, como mantê-lo em um servidor e/ou serviço como o GitHub e BitBucket, entre outros.

Observação: este tutorial só pode ser feito se você já adicionou sua chave no GitHub. Recomendamos também que você já tenha lido nosso conteúdo sobre Terminal.

Clone

Primeiro vamos ver o comando para clonar, isto é, obter uma cópia de um projeto que já está hospedado em um servidor Git. Isso pode ser feito com o comando `git clone` conforme a seguir:

```
$ git clone git@github.com:SEU_LOGIN/exemplo_git.git
Cloning into 'exemplo_git'...
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
Checking connectivity... done.
```

Este comando executa muitas ações, então vamos entender passo a passo. Primeiro ele realiza um comando `mkdir exemplo_git`, criando esse diretório. Sendo assim, vamos testar isso com:

```
$ cd exemplo_git
~/exemplo_git
```

Boa, temos um diretório `exemplo_git`!

Além disso, esse repositório Git já está inicializado e com o conteúdo salvo no servidor. Vamos ver:

```
$ ls -a
.  ..  .git/  README.md
```

Temos o diretório `.git` e o arquivo `README.md` e agora vamos ver o histórico de *commits* desse repositório com:

```
$ git log
commit a1e153f4d7096a048644935995c6ec6fb08ede88
Author: Enzo Rafael C. Batista<enzorcb@email.com>
Date:   Tue Apr 21 15:03:45 2015 -0300

      Initial Commit
```

Com o `git clone`, o Git, além de criar um diretório com o nome do projeto, traz todo o conteúdo do projeto e seu histórico!

Uma opção do `git clone`, entre muitas, é a de especificar o nome da pasta que o Git deve criar. Veja:

```
$ cd ~/workspace
$ git clone git@github.com:SEU_LOGIN/exemplo_git.git
teste_projeto_git
Cloning into 'teste_projeto_git'...
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
Checking connectivity... done.

$ cd teste_projeto_git
```

Ou este exemplo:

```
$ cd ~/workspace
$ mkdir teste_denovo_git
$ cd teste_denovo_git
$ git clone git@github.com:SEU_LOGIN/exemplo_git.git .
Cloning into '.'...
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
Checking connectivity... done.
```

Esse último comando clona o projeto na pasta atual. Você lembra que `.` corresponde à localização atual, não é?

Remote

Agora vamos ver algumas informações sobre o nosso repositório remoto com o comando `git remote`:

```
$ git remote -v
origin  git@github.com:SEU_LOGIN/exemplo_git.git (fetch)
origin  git@github.com:SEU_LOGIN/exemplo_git.git (push)
```

Ele nos mostra que temos um repositório remoto, que o nome dele é `origin` e que podemos fazer `push` e `fetch` nele. Esse é o nome padrão que o Git dá ao repositório. Depois veremos como alterar esse nome.

Falando em `fetch` e `push`, vamos ver o que é isso?

Fetch

Esse comando pega todos os dados de um determinado repositório remoto:

```
$ git fetch origin
```

Onde `origin` deve ser o nome do remoto que você quer pegar, podendo ser outro diferente de `origin` caso você tenha mais de um ou tenha mudado o nome do `origin`.

Para ver isso funcionando:

```
$ cd ~/workspace/exemplo_git
$ touch novo_arquivo.txt
```

Abra o arquivo no editor de código da sua preferência, faça algumas alterações e salve. Na sequência, continue com:

```
$ git add novo_arquivo.txt README.md
$ git commit -m "Algumas alterações simples"
[main c27e355] Algumas alterações simples
2 files changed, 1 insertion(+)
 create mode 100644 novo_arquivo.txt
```

Depois, rode o comando a seguir (ele será explicado em breve):

```
$ git push origin main
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 359 bytes | 0 bytes/s, done.
Total 4 (delta 1), reused 0 (delta 0)
To git@git.codesaga.com.br:exemplo_git
   a1e153f..c27e355  main -> main
```

Agora vamos simular o que aconteceria se outra pessoa (ou mesmo você em outra máquina) acessasse as alterações feitas no repositório:

```
$ cd teste_denovo_git
$ git fetch origin
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From git.codesaga.com.br:exemplo_git
   a1e153f..c27e355  main     -> origin/main
```

Veja que o Git obteve as alterações, mas vamos ver se elas já estão disponíveis:

```
$ git log
commit a1e153f4d7096a048644935995c6ec6fb08ede88
Author: Enzo Rafael C. Batista<enzorcb@email.com>
Date:   Tue Apr 21 15:03:45 2015 -0300

      Initial Commit
```

Hum, parece que não, né?

Por padrão, o Git traz as novas alterações para o repositório, mas não modifica nada ainda. Para isso você precisa incluir essas alterações com o comando `git merge` ou dar um comando `git pull` que faz um `fetch` e `merge` para você. Vamos fazer isso então:

```
$ git pull origin main
From git.codesaga.com.br:exemplo_git
 * branch         main  -> FETCH_HEAD
Updating a1e153f..c27e355
Fast-forward
 README.md        | 1 +
 novo_arquivo.txt | 0
 2 files changed, 1 insertion(+)
 create mode 100644 novo_arquivo.txt
```

Onde `origin` é o nome do repositório e `main` é o nome da *branch*. Em resumo, *branch* é uma ramificação do seu projeto. Vamos falar mais sobre isso no futuro.

Agora temos todas as alterações no nosso repositório, veja:

```
$ ls -l
total 4
-rw-rw-r-- 1 enzorcb enzorcb   0 Abr 21 16:35 novo_arquivo.txt
-rw-rw-r-- 1 enzorcb enzorcb 108 Abr 21 16:35 README.md
$ git log
commit c27e355574d97309ac938f731e069a840e32ac15
Author: Enzo Rafael C. Batista<enzorcb@email.com>
Date:   Tue Apr 21 16:29:22 2015 -0300

      Algumas alterações simples

commit a1e153f4d7096a048644935995c6ec6fb08ede88
Author: Enzo Rafael C. Batista<enzorcb@email.com>
Date:   Tue Apr 21 15:03:45 2015 -0300

      Initial Commit
```

Agora, sim! Todas as alterações estão disponíveis!

Push

Já vimos o `push` em ação quando estávamos falando do `fetch`, mas vamos explicar um pouco mais sobre ele.

Após aplicar alterações nos seus arquivos e fazer o *commit* dessas modificações, podemos enviar para o servidor com o comando `git push`, veja:

```
$ touch mais_um_arquivo.txt
$ git add mais_um_arquivo.txt
$ git commit -m "mais um arquivo adicionado"
[main 4e86ea6] mais um arquivo adicionado
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 mais_um_arquivo.txt

$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
nothing to commit, working directory clean
```

Adicionamos mais um arquivo, fizemos o *commit* e verificamos o status. O Git nos mostra que estamos um *commit* na frente do repositório, sendo assim, vamos sincronizar:

```
$ git push origin main
Counting objects: 2, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 302 bytes | 0 bytes/s, done.
Total 2 (delta 0), reused 0 (delta 0)
To git@git.codesaga.com.br:exemplo_git
   c27e355..4e86ea6  main -> main
```

Feito! O Git enviou nossas alterações para o servidor remoto. Os parâmetros `origin` e `main` são os mesmos usados no `git pull` – `origin` para o nome do remoto e `main` para o nome da *branch*.

Configuração de repositórios remotos

Além de sincronizar seu código com repositórios remotos, o Git permite que você use mais de um repositório para seu projeto local. Se você está desenvolvendo um projeto que está armazenado em alguma plataforma Git, como o GitLab, mas agora quer usar o GitHub, pode possuir os dois *remotes* sincronizados na sua máquina. Para isso, basta usar o comando:

```
$ git remote rename origin gitlab
$ git remote add origin git@github.com:usuario/projeto.git
```

No exemplo, o remote antigo (da plataforma GitLab) é renomeado para `gitlab` e, em seguida, adicionamos um segundo *remote* do GitHub com o nome `origin`. Você pode escolher os nomes que preferir, o importante é utilizar o endereço correto. Agora, se rodarmos o comando `git remote -v` serão listados os seguintes repositórios remotos:

```
$ gitlab git@gitlab.com:usuario/projeto (fetch)
$ gitlab git@gitlab.com:usuario/projeto (push)

$ origin git@github.com:usuario/projeto (fetch)
$ origin git@github.com:usuario/projeto (push)
```

Com o novo remote configurado, fica disponível o comando para realizar o envio do código:

```
$ git push origin main
```

Ou para mandar para o remote antigo:

```
$ git push gitlab main
```

Trabalhando com *branches*

Vimos que o Git registra o estado da sua aplicação a cada *commit* realizado, construindo com isso uma linha do tempo do progresso do seu código com todas as modificações realizadas. Com o Git também podemos criar ramificações do nosso projeto para que um grupo de pessoas possa trabalhar em diferentes funções da sua aplicação simultaneamente em diferentes computadores sem que um código afete o outro.

A *branch* principal na qual mantemos o código funcional é chamada de `main`. Cada ramificação criada a partir dela tem seu nome próprio e consiste num conjunto de modificações que mais tarde serão fundidas com a `main`. Dessa maneira, podemos fazer experimentos em nosso código o quanto quisermos, mantendo a `main` livre de erros, conflitos, etc.

Você nunca deve enviar códigos com problema para a main! Se isso acontecer, outras pessoas podem replicar os problemas para outras *branches*, o que vai criar um efeito em cascata. Além disso, a `main` é onde mantemos o código que será utilizado para fazer o *deploy* da sua aplicação, ou seja, é o que será mostrado ao mundo. Por isso ele deve ser estável, seguro e livre de erros.

Como criar uma *branch*

Antes de criarmos uma nova *branch*, vamos pesquisar quais *branches* já existem para o projeto que você está trabalhando. Para isso, utilizamos o comando:

```
$ git branch
* main
  admin_manages_users
```

Esse comando lista as *branches* locais criadas para esse projeto. Nesse caso temos só duas: a `main` e `admin_manages_users`. O sinal `*` indica em qual `branch` você está atualmente. Agora, vamos criar uma nova `branch` com o nome `new_feature`.

```
$ git branch new_feature
$ git branch
  admin_manages_users
* main
  new_feature
```

Temos uma nova `branch` e precisamos falar para o Git que queremos trabalhar nela, já que no momento ainda estamos na `main`. Para isso, usamos o comando `checkout`:

```
$ git checkout new_feature
$ git branch
  admin_manages_users
  main
* new_feature
```

Agora `new_feature` é a *branch* em que as modificações serão aplicadas. Ao final do trabalho, você vai fundir essas alterações com a `main`, com o comando `merge`.

Como fazer o merge

Você está trabalhando numa *branch* em uma nova função da sua aplicação e ela está completamente finalizada, com todos os testes passando. Agora você só precisa fundir seu código à `main`. Vamos ver como funciona esse processo?

O comando `merge` aplica na *branch* atual todos os *commits* de outra *branch*. Seguindo o exemplo anterior, para aplicar as modificações de `new_feature` na `main`:

```
$ git checkout main
$ git merge `new_feature`
Updating e33979f..3a215c9
Fast-forward
 new_feature_file.txt | 0
 1 file changed, 12 insertions(+), 2 deletions(-)
 create mode 100644 new_feature_file.txt
```

Vamos analisar as mensagens geradas pelo comando `merge`. Primeiro ele nos informa que o `commit e33979f` foi atualizado com o `commit 3a215c9`. Em Git usa-se um conceito chamado de "apontador". O apontador indica o *commit* ativo (ou corrente). Neste caso, como não ocorreram conflitos entre os códigos, o Git está "apontando" para o `commit 3a215c9`. Esse processo é indicado pelo termo *Fast-forward*, já que os códigos foram fundidos automaticamente e o apontador agora aponta para o `commit 3a215c9`.

Continuando com as mensagens do `merge`:

```
new_feature_file.txt | 0
1 file changed, 12 insertions(+), 0 deletions(-)
```

Indica que um arquivo foi adicionado `new_feature_file.txt`, e a quantidade de inserções e deleções que ocorreram. Caso o `merge` tenha encontrado conflitos, você vai precisar resolvê-los e um novo *commit* de `merge` deve ser criado.

Considerações finais

O Git é uma ferramenta muito complexa e poderosa. Existem diversas maneiras de executar a mesma ação e você pode encontrar na internet muitos tutoriais que vão mostrar alternativas aos exemplos descritos aqui. O intuito deste material é apenas apresentar conceitos básicos de forma simples, para que os iniciantes se sintam mais à vontade com o Git, uma vez que ele pode parecer extremamente difícil no primeiro contato.

Indo além

Quando você estiver mais confiante ou sentir necessidade de outras funções, procure a documentação do Git ou tutoriais na internet. Alguns comandos úteis são: `rebase`, `revert`, `reset` e `diff`, entre outros.

Abaixo algumas sugestões. No nosso site, você encontra alguns artigos com mais informações.

- Erros mais comuns com Git e como corrigí-los
- Git: configuração de repositórios remotos no seu projeto
- Connecting to GitHub with SSH
- Git on the Server - Generating Your SSH Public Key

Esse material está em constante evolução e sua opinião é muito importante. Se tiver sugestões ou dúvidas que gostaria de nos enviar, entre em contato pelo nosso e-mail: treinadev@campuscode.com.br.

Versão	Data de atualização
1.1.2	11/01/2022

treinadev

é um programa gratuito de
formação de devs da Campus Code

Apoio:

VINDI

RBASE

PORTAL solar

smart fit

konduto

CAMPUS CODE

Telefone/Whatsapp: [11] 2369-3476