**Comandos Git mais utilizados e como configurar**



Figurando entre os sistemas de controle de versão open source mais usados no mundo, o Git é visto hoje como uma ferramenta fundamental para a otimização de projetos. Mas, para usá-lo com sucesso, é preciso estar familiarizado com os principais **comandos Git**.

Para quem está iniciando nesta jornada, o começo provavelmente será bastante confuso e desafiador. As chances de você usar um comando e ele não dar certo logo de cara é bem grande no início, mas isso não é motivo para desanimar: o processo é o mesmo pra todo mundo!

Embora “quebrar a cabeça” para descobrir o que deu errado seja parte do caminho, não significa que a gente não possa te ajudar a passar por ele com mais facilidade, né? Pensando nisso, preparei um compilado sobre os comandos Git mais usados.

Quer saber como potencializar o trabalho do dia a dia e garantir uma melhor experiência e maior produtividade? Então vamos nessa!

A partir de agora, você vai aprender:

Git, SVN e CVS— comparação das[principais ferramentas de controle de versão de software](https://blog.geekhunter.com.br/git-svn-e-cvs-comparacao-dos-principais-vcs/).

**Conteúdo**  [ocultar](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/)

[1 O que é o Git](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#O_que_e_o_Git)

[2 Porque preciso dos comandos Git](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#Porque_preciso_dos_comandos_Git)

[3 Como começar com comandos Git](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#Como_comecar_com_comandos_Git)

[3.1 Git Status](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#Git_Status)

[3.2 Os commits](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#Os_commits)

[3.3 Git log](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#Git_log)

[4 13 comandos Git mais usados](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#13_comandos_Git_mais_usados)

[4.1 Git init](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#Git_init)

[4.2 Git clone](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#Git_clone)

[4.3 Git branch](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#Git_branch)

[4.4 Git checkout](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#Git_checkout)

[4.5 Git status](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#Git_status)

[4.6 Git diff](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#Git_diff)

[4.7 Git add](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#Git_add)

[4.8 Git commit](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#Git_commit)

[4.9 Git push](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#Git_push)

[4.10 Git pull](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#Git_pull)

[4.11 Git revert](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#Git_revert)

[4.12 Git merge](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#Git_merge)

[4.13 Git stash](https://blog.geekhunter.com.br/comandos-git-mais-utilizados/#Git_stash)

O que é o Git

Como dissemos na abertura deste post, o Git é um **sistema distribuído *opensource* de controle de versão**– o mais popular do mundo, diga-se de passagem.

Apesar de ser usado para registrar o histórico de edições de qualquer tipo de arquivo, incluindo trabalhos de TCC e monografias de doutorado, a adoção mais ampla desta ferramenta se dá na área de desenvolvimento de software.

Dentro deste universo, o objetivo do Git é **registrar todas as mudanças feitas no código-fonte de um projeto**, evitando que algo importante se perca no meio do caminho.

Para entender melhor como isso funciona, vale destrinchar o significado de “**sistema distribuído opensource de controle de versão”**:

* Ser ***opensource***significa que o Git é um software de livre utilização, ou seja, permite que qualquer pessoa com conhecimentos de programação use-o, modifique-o e aprimore-o;
* Dizer que o Git faz **controle de versões** é a mesma coisa que dizer que ele armazena conteúdo e mantém um histórico das alterações feitas nele (seja por um usuário só ou muitos);
* E, finalmente, o fato de ser um **sistema distribuído** significa que cada diretório de trabalho do Git é um repositório com um histórico completo e habilidade total de acompanhamento das revisões, independente de acesso a uma rede ou a um servidor central. Cada desenvolvedor tem, na sua máquina, acesso ao histórico completo do código trabalhado.

Ficou claro? Então vamos em frente!

Porque preciso dos comandos Git

Antes de falarmos sobre os **melhores comandos Git**, é oportuno esclarecer essa dúvida tão comum que, inclusive, explica um pouco do sucesso desta ferramenta.

Bom, se você desenvolve há algum tempo, sabe que é muito comum que empresas tenham mais de um dev trabalhando paralelamente no mesmo projeto, certo?

**Então, em primeiro lugar, sistemas como o Git existem para o código não virar uma bagunça quando há mais de uma pessoa trabalhando nele simultaneamente.**

Pode ser também que, ao longo do projeto, surja a necessidade de voltar para uma versão anterior. Logo, ter esse controle das alterações dá muito mais segurança pra quem está trabalhando.

Outra situação comum é o de equipes trabalharem paralelamente com projetos que tenham uma base de código comum. Neste caso, a capacidade do Git de criar ramificações passa a ser bastante útil!

**Como começar com comandos Git**

A partir de agora, vou explicar como usar os comandos git, de fato. E, para isso, utilizarei o VSCode como editor padrão.

Caso você queira configurá-lo aí também, confira o link a seguir: [**Visual Studio Code + Git: como transformar o VSCode no editor padrão**](https://blog.geekhunter.com.br/visual-studio-code-como-editor-do-git/)**.**

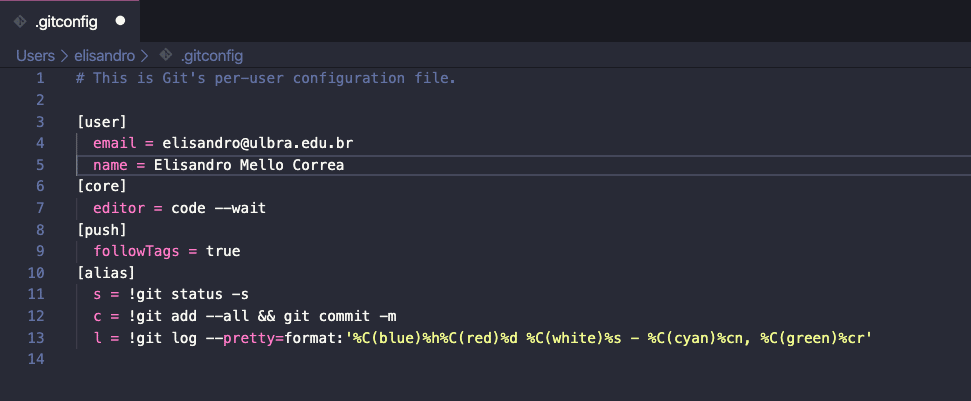
Feito isso, o próximo passo é abrir o arquivo **.gitconfig**, responsável pelas opções de configuração do GIT, utilizando no terminal o comando:

git config --global -e

No arquivo de configuração, devemos encontrar a sessão **Alias**.

Caso essa sessão não exista no .gitconfig, realize a declaração desta nova sessão no final do arquivo, procedendo com a inclusão da expressão **alias entre colchetes**:

[alias]



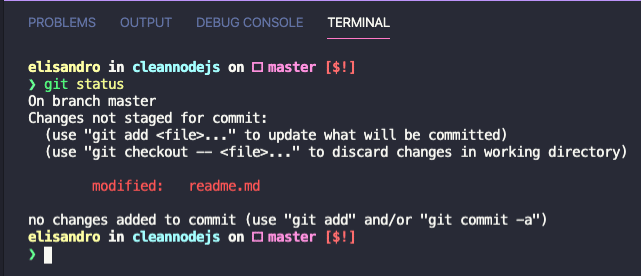
Após a declaração da sessão, podemos proceder com a configuração de nosso primeiro atalho para os métodos do Git.

Git Status

Para começar, iremos criar um atalho para o comando **git status**, cujo objetivo é exibir as condições do diretório de trabalho e da área de *staging*.

Em outras palavras, ele permite inspecionar quais alterações foram despreparadas, quais não foram e quais arquivos não estão sendo monitorados pelo Git.

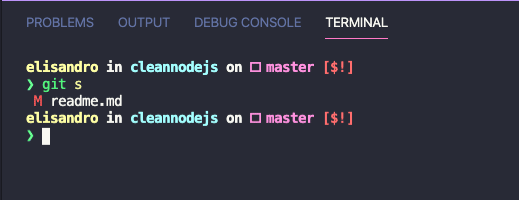
Ao executar o comando git status, é possível obter o status completo do repositório (diretório onde os arquivos do seu projeto serão armazenados).



Mas ao adotar o parâmetro **-s** no final do atalho criado, cria-se um status resumido, resultando em maior dinamismo.

Veja como fica:

s = !git status -s



Obs.: é recomendável verificar as condições do repositório antes das alterações receberem *commit,* sobre o qual falamos na sequência.

Os commits

O segundo atalho que vamos criar será para o nosso processo de *commit*, nome dado a um conjunto de alterações feitas em um projeto.

Nesta configuração, utilizaremos os parâmetros **–all**, que adiciona todos os arquivos para o *staged* de mudanças, e logo em seguida **(&&)** executar o *commit* com o parâmetro **–m**, que permite incluir a mensagem descritiva do nosso *commit*.

c = !git add --all && git commit -m

Lembre-se que caso você não queira adicionar todos os arquivos alterados para o processo de *commit,* deve-se incluir individualmente os arquivos que desejar com o comando tradicional **git add [nome do arquivo]**e depois executar o comando **git commit -m “[Mensagem Descritiva]”**.

Ou seja, o processo tradicional.

Git log

Também incluiremos o atalho para o comando **git log** (que permite ver o histórico de commits ou um específico), utilizando opções de personalização do comando para uma visualização mais amigável e descritiva.

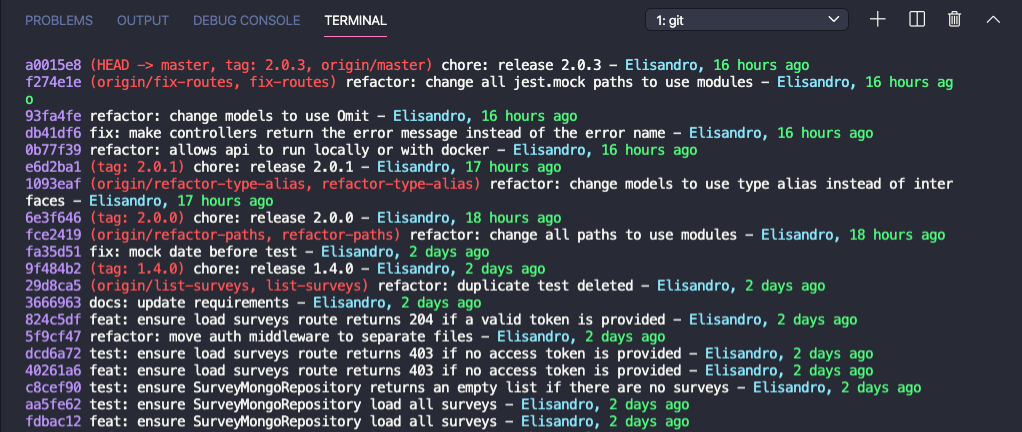
Para isso utilizaremos o parâmetro **–prety=format:**que permitirá customizarmos a apresentação de nossos*logs* a partir do atalho que criarmos, com algumas opções específicas:

* **%h:**o percentual com a letra **h** minúscula irá apresentar o *hash* reduzido do *commit;*
* **%d:** o percentual com a letra **d** minúscula irá mostrar a *branch* e também a *tag* do *commit*(caso exista)*;*
* **%s:** o percentual coma letra**s**minúscula (**de subject**) irá demonstrar a mensagem do *commit*;
* **%cn:**nome da pessoa que realizou o *commit*;
* **%cr:**a data relativa do *commit.*

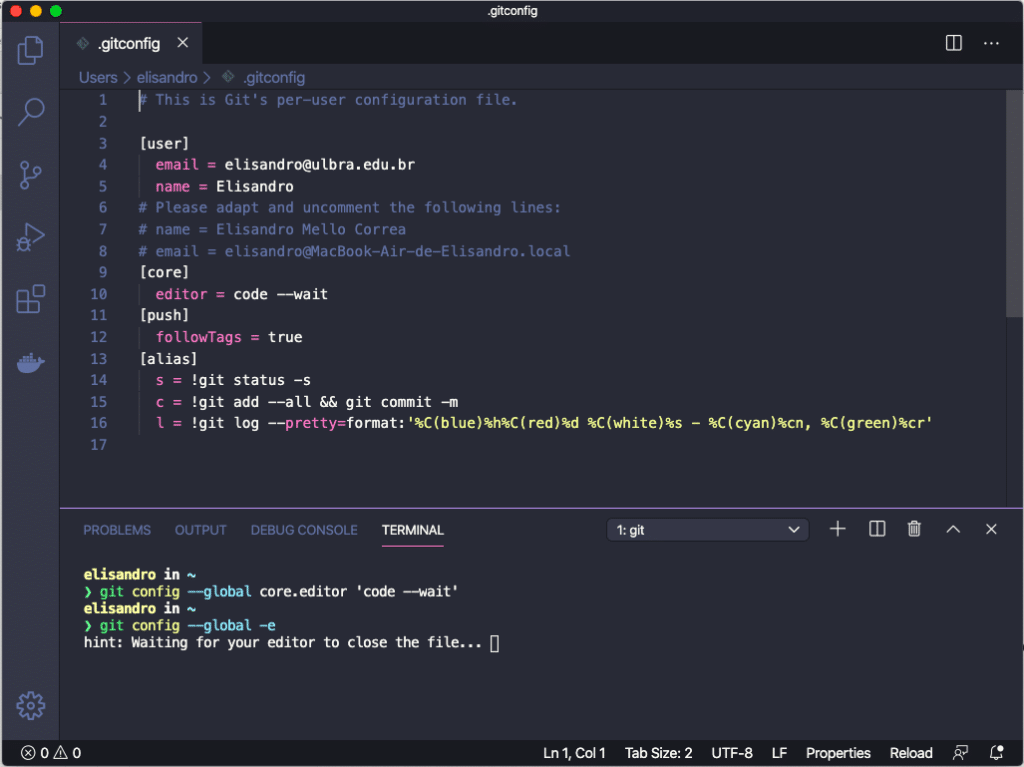
Também utilizaremos a opção de utilizar cores diferentes para cada coluna, o que facilita a visualização, utilizando o parâmetro **%** com a letra **C** maiúscula**(%C)**seguido da cor que desejamos entre parênteses**(blue)**.

Assim:

l = !git log --pretty=format:'%C(blue)%h%C(red)%d %C(white)%s - %C(cyan)%cn, %C(green)%cr'



Após todas essa etapas, o arquivo**.gitconfig** ficará desta forma:



Assim, concluímos a configuração dos atalhos básicos. Mas é claro que não vamos parar por aí…

Agora que você já sabe quais são os **comandos básicos Git**, chegou a hora de conhecer também outros atalhos que farão toda a diferença no seu dia a dia de trabalho. Confira!

Encontre a oportunidade de trabalho ideal para você! [Vagas para desenvolvedor](https://www.geekhunter.com.br/vagas?utm_source=blog&utm_medium=texto_ancora&utm_campaign=comandos-git) com o seu perfil. Confira!

13 comandos Git mais usados

Git init

Para começar um projeto que ainda não seja um repositório (ou repo), o Git Init costuma ser o comando mais indicado.

Basicamente, ele cria um repositório vazio ou transforma uma pasta que você já tem e que não está com controle de versão em um repositório:

git init

Com sua pasta de trabalho devidamente iniciada, é hora de começar a preenchê-la.

Git clone

O Git clone é um comando para baixar o código-fonte existente de um repositório remoto (como o Github, por exemplo).

Existem algumas maneiras de baixar o código-fonte, mas eu prefiro o clone com o modo https:

git clone <https://url-do-link>

Quando você clonar um repositório, o código é copiado para a o seu computador e continua linkado ao original, como foi explicado lá na descrição do que é um sistema distribuído.

Se você quiser desvincular a sua cópia do original, rode o comando abaixo:

git remote rm origin

Git branch

Com as *branches* (ou ramificações), vários desenvolvedores podem trabalhar paralelamente no mesmo projeto. Assim, cada um pode codar a sua parte sem que haja confusão!

Funciona basicamente da seguinte forma: a cada alteração realizada no código, seja para adicionar um novo recurso ou até mesmo corrigir um erro, cria-se um novo ponto de ramificação que consolida essas mudanças, sem interferir nos arquivos originais.

Considerado um dos comandos Git mais importantes, o branch pode ser usado para três finalidades diferentes: criar, listar e excluir ramificações. Veja, na sequência, como trabalhar com cada uma delas.Para criar uma nova *branch*local, digite:

git branch <nome-da-branch>

Este comando criará uma *branch*local. Para upar a nova *branch* para o repositório remoto, você precisa usar o seguinte comando:

git push -u <remote> <nome-da-branch>

Para ver as ramificações, por sua vez, use:

git branch

ou

git branch --list

Por último, delete uma *branch*a partir do comando*:*

git branch -d <nome-da-branch>

Git checkout

Esse é um dos comandos git mais utilizados. Para trabalhar em uma *branch*, primeiro você precisa mudar para ela. Apesar de parecer algo óbvio, não ir para a *branch* que você acabou de criar e na qual quer trabalhar é um erro bastante comum no começo.

Neste contexto, o principal objetivo do*git checkout* é ajudar você a mudar de uma *branch* para outra ou então verificar arquivos e *commits*:

git checkout <nome-da-ramificação>

Há ainda um comando de atalho que te permite criar e ir para um *branch* de uma vez só:

git checkout -b <nome-da-branch>

Git status

Sim, já falamos sobre ele, mas vale retomar: o comando **Git status**serve para fornecer algumas informações importantes sobre a *branch*em que você estiver no momento, incluindo se ela está atualizada em relação à master e quais arquivos foram alterados.

git status

Git diff

Apesar de o git status fornecer muitas informações relevantes, às vezes é preciso ir além.

Caso você queira, por exemplo, saber exatamente o que você alterou (e não apenas quais arquivos foram alterados), o melhor a se fazer é utilizar o comando:

git diff

Basicamente, sua execução realiza uma função de comparação nas fontes de dados Git e mostra quais linhas foram adicionadas e removidas.

Git add

Quando criamos, modificamos ou excluímos um arquivo, essas alterações ocorrerão em nosso ambiente local e não serão incluídas no próximo *commit* (a menos que alteremos as configurações).

Para incluir as alterações de um arquivo em nosso próximo *commit*, será preciso usar o comando **git add**.

Para adicionar apenas um arquivo:

git add <arquivo>

Para adicionar, de uma vez, todos os arquivos modificados:

git add -A

Vale lembrar que o comando **git add** não altera o repositório e as alterações não são salvas até usarmos o *git commit*.

Git commit

Relembrando, esse comando visa definir um ponto de verificação no processo de desenvolvimento, para o qual você pode voltar mais tarde, se necessário.

git commit -m "mensagem explicando a mudança no código"

Git push

Após confirmar as alterações, a próxima coisa que você deseja fazer é enviar as alterações para o servidor remoto usando o comando **git push**:

git push <remote> <nome-do-branch>

No entanto, se seu branch for criado recentemente, você também precisará fazer upload do branch com o seguinte comando:

git push -u origin <nome-do-branch>

Git pull

O **git pull** é usado para obter atualizações do repositório remoto.

Trata-se de um comando que depende do referencial de onde ele foi feito, ou seja, um git pull feito da sua máquina vai puxar informações do repositório original para ela. Mas um git pull feito a partir do repositório original vai puxar as informações da sua máquina. Percebe?

Este comando é uma combinação de *git fetch*(baixa as alterações do repositório remoto, mas não as mescla com o seu)e*git merge* (que mescla tudo junto).

Isso significa que, quando usamos o **git pull**, ele recebe as atualizações do repositório remoto (**git fetch**) e aplica imediatamente as alterações mais recentes no seu local (**git merge**).

git pull <remote>

Git revert

Existem várias maneiras de desfazer nossas alterações local ou remotamente (dependendo da necessidade), mas devemos usar esses comandos com cuidado para evitar problemas.

Uma maneira segura de desfazer os *commits* é usando **git revert**.

git revert 'número do hash'

O número do hash pode ser conseguido pelo comando:

git log -- oneline

Git merge

Quando você conclui o desenvolvimento em sua *branch*e tudo funciona bem, sem conflitos, a etapa final é mesclar as *branches*, isso é feito com o comando **git merge**.

Como falamos no tópico sobre git pull, esse comando vai mesclar, no seu repositório local, todas as alterações feitas.

git merge <nome-da-branch>

Git stash

Sabe quando você está trabalhando em uma parte do projeto que não está fluindo como deveria e, então, decide mudar de branch por um tempo para trabalhar em outra coisa?

Bom, quando isso acontece não é preciso necessariamente fazer o commit. Com o comando **git stash**, existe a possibilidade de apenas arquivar as alterações que você fez na cópia de trabalho durante um determinado período para que seja possível retornar a ela mais tarde.

Para salvar as alterações sem *commit*, basta executar:

git stash

Já para ver todas as stashes que você guardou, use:

git stash list

Por último, você também pode aplicar stashes antigos, especificando:

it stash apply stash@{2}.

Se você chegou até aqui, saiba que essa é jornada que está apenas começando: existem diversos outros tipos de comandos e configurações, que podem ser verificadas na [documentação oficial do Git](https://git-scm.com/docs/git-config).

Mas agora que você já conhece os melhores comandos git, te convido a ler o próximo post dessa trilha de conhecimento: [**O que é Commit e como usar Commits Semânticos**](https://geekblogti.wpengine.com/o-que-e-commit-e-como-usar-commits-semanticos/). Até lá!

Disponível em: <https://blog.geekhunter.com.br/comandos-git-mais-utilizados/>