# Git e GitHub

Hoje em dia, em geral, ninguém programa sozinho. O git é uma ferramenta criada para facilitar na colaboração distribuição de códigos. O git pode ser usado em servidores privados, mas no geral, os códigos são enviados para o GitHub, um site que hospeda os códigos e que também tem uma interface gráfica simples, que facilita a visualização de mudanças feitas por outras pessoas.

## Começando

O git é instalado como qualquer outro programa, seja por um instalador do Windows ou pelo package manager do linux, pelo comando `sudo apt-get install git` (dependendo da sua distribuição). Depois de instalado, ele já pode ser usado em um terminal, independente do sistema operacional. Uma das primeiras coisas a se fazer é configurar sua conta do GitHub. Isso é importante para seus commits serem mais facilmente identificados. Para isso, é preciso executar 2 comandos:

```
git config --global user.name "Seu Nome"
git config --global user.email seuemail@exemplo.com
```

Depois disso, já podemos começar a usar o git.

## Importando um repositório

Cada projeto tem seu repositório, como é chamada a *"pasta"* que guarda os arquivos no servidor. Para começar a trabalhar em qualquer repositório, precisamos importar ele no seu computador. Há dois modos de fazer isso: usando HTTPS e usando SSH. Uma vantagem de HTTPS é que não é necessário configurar nada previamente no sistema e no perfil do GitHub. Uma desvantagem, porém, é que será necessário digitar seu usuário e senha toda vez que aplicar as mudanças locais no repositório online ( `git push` ). Já usando SSH, não será necessário digitar as informações de sua conta toda vez que executar `git push` , mas é preciso configurar algumas coisas no computador antes (lembrando que, provavelmente, só será necessário fazer essa configuração uma vez, e não é difícil).

### Usando HTTPS

1. Navegue até a pasta em que você quer importar o repositório, usando o comando `cd` (uso: `cd <caminho/para/pasta>` ).

2. `git clone <URL_do_repositório>`. Esse URL pode ser obtido indo até o repositório no GitHub, e clicando em *Clone or Download*. Por padrão, a janela mostrada é clonar com HTTPS, mas caso esteja escrito *Clone with SSH* no título dessa janela, clique em *Use HTTPS*.

No caso desse repositório, o comando seria:

```
git clone https://gitlab.com/letscode-academy/sitelc.git
```

### Usando SSH

Para usar SSH, é necessário gerar uma chave SSH (ou usar uma existente) e adicionar essa chave à sua conta no GitHub. Esse guia explica como fazer isso e testar se a configuração está correta. Feita essa configuração, o processo de clonar o repositório é basicamente o mesmo.

1. `cd <caminho/para/pasta>`.
2. `git clone <URL_do_repositório>`. Esse URL pode ser obtido indo até o repositório no GitHub e clicando em *Clone or Dowload*. Em seguida, clique em `Use SSH` (se o título dessa janela for *Clone with SSH*, já está certo). O endereço mostrado é o URL do repositório necessário.

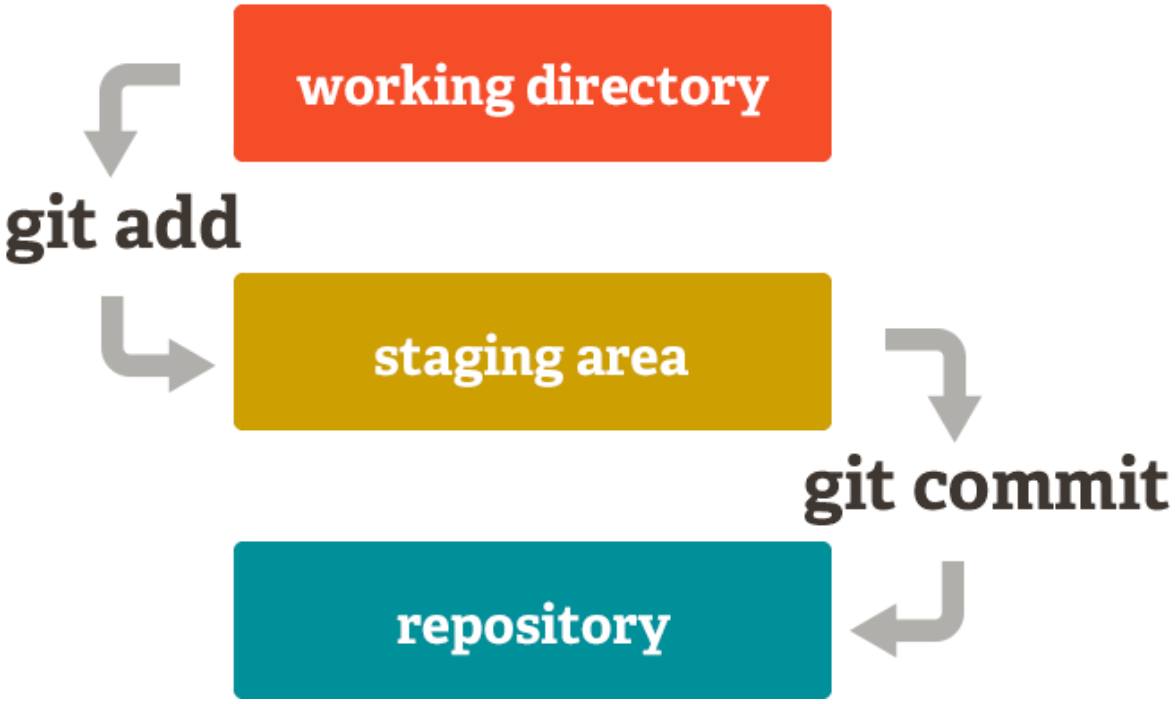
No caso desse repositório, o comando seria:

```
git clone git@gitlab.com:letscode-academy/sitelc.git
```

## Trabalhando com o repositório

Agora que você já tem uma cópia do repositório no seu computador, já podemos começar a trabalhar com o código, fazendo as alterações que quisermos. No entanto, é sempre bom executar um `git pull` (que baixa as últimas alterações no repositório) antes de começar a trabalhar(para evitar conflitos por seu repositório estar desatualizado). Depois que tivermos feitas as mudanças e estivermos satisfeitos com o código, podemos enviá-lo para o GitHub.

O git trabalha da seguinte forma: você tem uma cópia local do repositório, na qual ficam os arquivos e você faz alterações. Depois, você adiciona essas mudanças para a staging area, onde elas ficam prontas para serem enviadas em um "bloco", o commit, para o repositório. Caso você queira enviar também as mudanças para um repositório remoto, é preciso usar o comando push.




Primeiro, precisamos adicionar as nossas mudanças para a staging area. Para vermos essas mudanças, usamos os comandos `git status` (que mostra um resumo) ou `git diff` (que mostra as mudanças linha por linha). Para realmente adicionar no commit, usamos o comando `git add`. Com esse comando, podemos selecionar especificamente quais arquivos ou pastas queremos adicionar no commit, ou podemos usar `git add .` ou `git add -A` para adicionar todos de uma vez. Usando o comando `git diff --cached` ou `git diff --staged`, podemos ver as mudanças feitas apenas nos arquivos que já estão na staging area, ou seja, que vão ser enviados no próximo commit.

Depois disso, para criar um commit, usamos o comando `git commit`. Esse comando abre uma janela do editor de texto padrão do sistema operacional, para adicionar uma mensagem de resumo do commit. Esses editores costumam ser meio difíceis de usar, então, pessoalmente, gosto de o modificador `-m`, que permite enviar a mensagem diretamente. O comando fica assim:

```
git commit -m "Essa é a mensagem do commit"
```

Os commits são locais, ou seja, eles são só da sua máquina. Eles normalmente são enviados um por um para o servidor, mas você pode usá-los pra se organizar, sem enviar. Depois que você tiver feito quantos commit quiser, e achar que está pronto para enviar seu código para o GitHub, basta executar o comando `git push`. No caso de algum erro, provavelmente seu repositório local está desatualizado, e isso pode ser resolvido com um pull. Caso haja problemas no auto-merge, eu ensino a resolver na parte final da aula. Caso não dê nenhum erro, o seu código já está no GitHub. Em resumo, um commit fica normalmente assim:

```
git add .
git commit -m "Mensagem"
git push
```

*PS: Podemos adicionar diretamente ao commit o modificador `-a`, que faz a função do add. No entanto, isso só funciona para modificações de arquivos, sendo que para arquivos que foram criados nesse commit o comando add ainda é necessário*

## Usando branches

O que passei até agora é o básico do GitHub. É o necessário para trabalhar em uma única "versão" do código por vez. No entanto, no git existe a possibilidade de se fazer alterações no código (e compartilhar com outros usuários), sem comprometer um código que já está funcionando. O código do branch *master* (o principal, e o único que existe por padrão) deve ser sempre estável e funcional, com testes sendo feitos em outros branches.


Para criar um novo branch, primeiro execute um `pull`, para garantir que os arquivos estão atualizados. Então, execute `git checkout -b [Nome do branch]`. O modificador `-b` serve para criar um novo branch. Agora, você está em um branch diferente do master, podendo fazer mudanças e commits sem alterar o código original. Para mandar esse branch para o GitHub e criar um branch remoto, usamos o comando `git push origin -u [Nome do branch]`. Depois da primeira vez, podemos usar o push direto(desde que estejamos no branch certo).

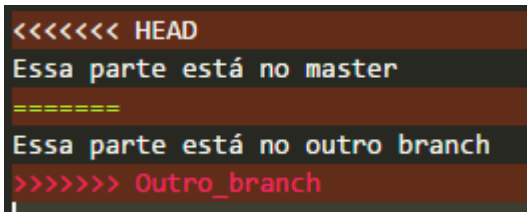
Para mudar de branch, podemos usar o comando `git checkout [Nome do branch]`. Tome cuidado antes de fazer isso, qualquer mudança que você tiver feito e não estiver em um commit será jogada fora. Para evitar isso, crie um commit ou use o comando `git stash` que guarda todas as mudanças em um commit temporário, que pode ser recuperado facilmente, por exemplo, com o comando `git cherry-pick [código do commit]`, que "pula" o estado do repositório para um certo commit. Uma lista de modificações do repositório pode ser acessada com o comando `git reflog`, e dele, você seleciona o commit criado pelo stash.

Se você estiver satisfeito com o trabalho de um branch, e quiser juntá-lo ao master, basta voltar para o master e usar o comando `git merge [Nome do branch]`. No entanto, caso a mesma parte de um mesmo arquivo tenha sido modificada nos 2 branches, o auto-merge vai falhar, e te avisar quais os arquivos problemáticos. (o mesmo pode acontecer quando você altera coisas sem antes dar pull). Nesses arquivos, ficam as duas versões diferentes daquela parte. Você pode escolher uma delas, e então usar o add e criar um commit para resolver o problema. Agora, você já pode deletar o seu branch antigo, pois tudo está no master.

Com o comando `git branch -a` podemos ver todos os branches, locais ou remotos, do nosso repositório. Para deletar um branch local, usamos `git branch -d [Nome do branch]`. Já para deletar um branch remoto, usamos `git push origin :[Nome do branch]` (note os dois pontos).

## Resolvendo conflitos de merge

Quando você tenta dar um merge em dois branches ou dar pull nas mudanças remotas do seu repositório, o git tenta misturar as duas versões do arquivo. No entanto, se tiver alterações na mesma parte do programa nos dois arquivos, o auto-merge vai falhar, e você vai ter que resolver os problemas manualmente. O git vai te avisar quais os arquivos problemáticos, e em cada um deles vai criar uma estrutura assim:




A parte de cima é a que está no seu arquivo original, e a parte de baixo é a que estava no outro arquivo, seja o de um outro branch ou o que veio de um repositório remoto depois de um pull. Para resolver, você tem que escolher a versão que você quer, ou uma mistura das duas. Então, você precisa remover os marcadores do conflito ( `<<<<<<<` , `=======` e `>>>>>>>` ). Depois de fazer isso em todos os conflitos, você precisa usar o `git add` e o `git commit` para criar um commit com essas mudanças, e o conflito está resolvido.

## Aliases

Aliases são atalhos para comandos do git. Esses atalhos são úteis principalmente para comandos muito longos ou comandos que você não considera intuitivos. Há duas formas de criar aliases:

1. Usando o comando `git config --global alias.<atalho> <comando>` .
2. Editando o arquivo `.gitconfig` no diretório home ( `~` ), adicionando linhas dessa forma:

```
[alias]
    <atalho1> = <comando1>
    <atalho2> = <comando2>
    (...)
```

Exemplos: Executar o comando `git config --global alias.graph log --all --graph --decorate --color --oneline` ou editar o arquivo `.gitconfig` , adicionando o abaixo:

Tópico anterior Próximo Tópico

```
[alias]
```