

CONTROLE DE VERSÃO

Git e GitHub para iniciantes – Tutorial completo

Por Wesley Willians em 21/08/2019

Falando sobre sistemas de controle de versão

Para os que não conhecem o Git, ele é um sistema de controle de versão de código. Daqui pra frente, todo código que forem desenvolver, vocês deverão utilizar este sistema, pois, dessa maneira, nunca mais correrão o risco de perder todo o trabalho de horas, ou até mesmo, meses.

Sempre haverá a dúvida se vale a pena trabalhar com sistema de controle de versão. Será que isso não é só uma moda que vai passar?

A resposta é **NÃO**. De forma alguma.

Trabalhar com controle de versões, é algo imprescindível, para um projeto. Daremos algumas razões para que vocês se convençam.

Organização

Conforme vocês desenvolvem trechos de código, vão salvando e, caso necessitem, podem voltar para versões anteriores.

Desenvolvam suas aplicações baseadas em funcionalidades e cada funcionalidade poderá pertencer a um **branch**.

Trabalho em equipe

Quando trabalhamos sozinhos, o procedimento é de uma forma, quando trabalhamos em equipe, tudo muda, e o controle de versão nos ajuda neste aspecto. Acabou aquela história de dois desenvolvedores alterarem um mesmo arquivo, ao mesmo tempo, por estarem desenvolvendo um mesmo projeto.

Segurança

Vocês desenvolverão, todos os projetos, sem medo de perder código ou acabar errando alguma atualização, sem ter como voltar.

Release

Há a possibilidade de criar releases como: **1.0**, **1.0.1**, **1.0.2**, e assim por diante. Podendo assim, entregarem códigos completos, em diferentes versões.

Preparem-se para entrarem no mundo de sistemas controladores de versões.

História do Git

Quando falamos em Git, não podemos deixar de falar do Kernel, do Linux. Não sei se vocês conhecem este assunto, ou se acompanharam o modo que o Kernel era desenvolvido. Basicamente, era um processo insano.

Se eu quisesse contribuir com este Kernel, desenvolveria uma funcionalidade e geraria um patch desta funcionalidade para que os desenvolvedores, da época, pudessem avaliar o código e depois aplicar ao Kernel Central. Vejam que, cada desenvolvedor que estivesse trabalhando com o Linux, teria que desenvolver, gerar um patch e depois enviar para análise e, somente depois, seria aplicada a atualização ou melhoria.

Esta logística era insana, complexa e demorada. Em 2002, mais ou menos, a comunidade, que mantinha o Kernel do Linux, fez um acordo com uma empresa que mantinha um sistema de controle de versão, chamado BitKeeper.

Entre 2002 e 2005, a equipe do Kernel do Linux, só utilizava o BitKeeper, como sistema de controle de versão. Houve um desentendimento entre a empresa, responsável pelo BitKeeper, e a comunidade. Depois disso, toda comunidade recebeu a notícia que, quem quisesse utilizar o BitKeeper, a partir daquele desentendimento, deveria pagar pelo serviço.

Devido a este fato, o Linus Torvalds, que é o criador do Linux, decidiu unir a equipe e verificar o que eles haviam aprendido, em todo aquele tempo, em relação aos sistemas de controle de versão e também sobre o BitKeeper. Com isso, ele decidiu criar um hiper sistema de controle de versão, mas um sistema de controle de versão que fosse totalmente simples, robusto e que não juntasse responsabilidade, demais.

Em 2005, aproximadamente, surgiu o projeto do Git, que vem até hoje, sendo amadurecido, cada dia mais. E este projeto já se espalhou por todas as áreas de desenvolvimento de software.

Quando pensarem em Git, é importante entenderem como este sistema surgiu e que ele quebrou diversos paradigmas de controle de versões, trabalhando de forma, descentralizada.

Sistema centralizado vs descentralizado

Para começarmos a explicar, podemos falar de dois sistemas: CVS e SVN, ou Subversion. Nestes modelos de sistemas, normalmente, existe um repositório central, que armazenamos todo o código. Quando precisamos fazer alguma alteração, baixamos o código, fazemos a alteração e depois, fazemos um **commit**. O commit é o momento em que informamos, ao sistema, que estamos gerando uma versão daquela alteração que acabamos de fazer.

Quando o sistema é centralizado, no momento em que executamos um commit, o sistema fará um upload do arquivo para o servidor e quando, qualquer outro desenvolvedor, for fazer qualquer alteração, ele terá que baixar, novamente, o código, gerando um **update**. Dessa forma, ele recebe as alterações que foram executadas no repositório, para, somente depois, começar a desenvolver qualquer outra atualização ou melhoria.

Notem que, vocês só conseguirão executar commits no repositório, se tiverem acesso à internet, uma vez que o sistema é centralizado e depende do servidor online, para fazer a atualização. Se estiverem trabalhando em algum lugar, que não tenha acesso à internet, vocês não conseguirão fazer um commit.

Isso é um grande transtorno, porque, quanto mais commits fizerem, melhor será para o projeto. Trabalhando com conceito de micro commits, facilita na hora de acompanharem a evolução do projeto. Suponham desenvolver o dia inteiro e no final, fazem o commit. Qual a vantagem que tiveram? Se precisarem voltar uma versão, perderão todo o código desenvolvido, sendo que, muitas coisas poderiam ser reaproveitadas daquele dia inteiro de trabalho.

O grande problema é que, sempre que precisarem evoluir o controle de versão, deverão estar conectados à internet para enviarem para o repositório central. Isso é, extremamente, ruim para quem trabalha com desenvolvimento.

Quando o Git surgiu, ele veio quebrando este paradigma. Ele trabalha como, se cada máquina em que estiver instalado, fosse um servidor, ou seja, é um repositório. O dono deste repositório pode dar quantos commits ele quiser.

Suponham pegar o repositório de alguém e fazer um clone para máquina. Vocês poderão fazer alterações e dar commits, localmente, sem precisar da internet, inclusive. Quando acharem que chegou o momento de enviarem para o repositório, só precisam rodar o comando **push**, que subirão para o repositório online. O Git tentará unir as alterações com os códigos que estiverem no repositório. Se ele não conseguir, ele pedirá para corrigir alguns detalhes e enviar, novamente. Isto ocorre quando, mais de uma pessoa, está trabalhando em cima do mesmo repositório e as duas tentam dar um push, ao mesmo tempo.

O grande ponto é que, não precisam mais trabalhar de forma centralizada. Cada pessoa pode ter seu próprio seu repositório, localmente, e trabalhar de forma, independente. Outra vantagem é que, desta forma descentralizada, o ganho de produtividade é muito grande.

Além de todas estas vantagens, utilizar o Git faz com que vocês projetem suas aplicações de forma diferente do que as convencionais. Suponham estar desenvolvendo um código, mas não querem que muitas pessoas saibam o processo que fizeram para desenvolver. Para isso, vocês podem criar

um branch, que é uma ramificação do projeto. Não existe uma quantidade máxima de branches a serem criados, logo, vocês podem criar quantos forem necessários para organizarem o desenvolvimento do projeto.

Assim, terão que fazer um **merge**, que unificará os códigos e, somente depois disso, enviarão para o repositório central. O mais interessante é que, quem recebe estes dados do repositório central, não consegue saber que você trabalhou com diversos branches internos, porque isso é particular e você trabalha da sua forma.

No sistema antigo, isso era impossível, porque se criassem um branch, este branch deveria ser enviado de forma pública, a todos.

Além de tudo, o Git oferece uma forma muito simples de trabalhar, vocês verão que, com apenas alguns comandos, já conseguem ter muito poder em suas mãos. Quanto mais utilizarem a ferramenta, mais prática ganharão e não temos dúvidas, de que vocês gostarão muito e nunca mais deixarão de utilizá-la.

Esperamos que tenham entendido a diferença entre sistema de controle de versão centralizado e descentralizado.

Configuração inicial

O primeiro passo será abrir o terminal, se estiverem no Mac ou Linux. Se estiverem utilizando Windows, podem utilizar o Git Bash, que vem junto com o Git. Não utilizem o prompt do MS-DOS, porque ele é muito ruim. O Git Bash simulará um ambiente Unix para facilitar o trabalho.

Em seguida, digitem **git** no terminal, para saberem se está instalado ou não, e apertem o enter. Caso não esteja instalado deverão instalá-lo.

Se estiver instalado, receberão uma listagem de comandos do Git. Uma dica, para quem utiliza Windows é, verificar se a variável de ambiente está devidamente configurada. Esta dica serve para os casos de digitarem git no Git Bash e não obterem nenhum retorno. Muitas vezes o git pode estar instalado, mas o executável não está configurado nas variáveis de ambiente, neste caso o terminal não reconhecerá o comando.

Caso o git esteja instalado, mas não configurado nas variáveis, acessem **Meu Computador**, depois **Variáveis de Ambiente** e procurem a variável **path**. É nela que vocês adicionarão o caminho

completo da pasta onde existe o arquivo executável do Git. Esta é uma dica, mas se estão abrindo o terminal com o Git Bash, provavelmente não terão este problema.

Após a verificação, rodaremos os seguintes comandos:

```
git config --global user.name "seu nome"
git config --global user.email "seu-email@email.com"
```

Fiquem tranquilos, vocês não precisarão decorar estes comandos, porque, só na primeira vez que estiverem utilizando o Git que é necessário e este comando é facilmente encontrado.

Este procedimento é necessário para o Git saber quem são vocês. Quando rodarem qualquer comando, o Git fará o reconhecimento por estas informações.

Outra dica que podemos dar, para facilitar o entendimento visual dos comandos, é rodar o comando abaixo:

```
git config --global color.ui true
```

Este comando configurará o Git para colorir os comandos e resultados de comandos, facilitando o entendimento de cada iteração. Esta configuração só tem efeito no terminal e no Git Bash, caso estejam utilizando o prompt do MS-DOS, esta configuração não surtirá efeito.

Estamos com as configurações em dia e prontos para começar a utilizar o Git. Não fiquem espantados, realmente é muito fácil fazer esta configuração inicial.

Os 3 estágios

Mostraremos como criar o primeiro repositório Git e quais são os três principais estágios do processo. Utilizaremos alguns comandos, que talvez vocês não conheçam, para mostrar o processo.

Criando um repositório

A primeira dúvida que pode surgir é sobre o que é um repositório. Repositório é uma pasta em que ficarão todos os arquivos do projeto. A grande diferença desta pasta é que, quando iniciamos um repositório no Git, os arquivos desta pasta passam a fazer parte do controle de versão. Sempre que falarmos de repositório, daqui pra frente, estaremos falando de uma pasta que está fazendo parte de um controle de versão.

Primeiro passo será criar uma pasta e depois acessar a mesma. Para isso utilizaremos os comandos abaixo:

Para criar a pasta:: `$ mkdir aulagit`

Para acessar a pasta: `$ cd aulagit`

Agora que estamos dentro da pasta criada, basta rodarmos, um simples comando, para que esta pasta passe a ser um repositório. Vejam o comando abaixo:

```
$ git init
```

Ao rodarmos este comando, o Git criará uma pasta oculta chamada **.git**. Para verificar se a pasta, realmente, existe podemos rodar o comando `$ ls -la`. Este comando lista todos os arquivos e pastas, inclusive os ocultos.

Se acessarem este diretório, verão que existem diversos arquivos e pastas. Estes arquivos e pastas fazem parte da estrutura que o Git utilizará para controlar as versões dos projetos. Para verificarem, rodem os comandos abaixo:

Acessar pasta .git: `$ cd .git`

Listar itens internos: `$ ls`

Desta forma, se quiserem destruir um sistema de versão de um projeto, basta apagarem esta pasta completa e não terão mais controle de versão, agindo sobre aquele projeto.

Com a criação do repositório, criaremos um arquivo dentro desta pasta.

Criando arquivo: `$ touch arquivo.txt`

Neste momento, só criamos o arquivo, mas ele não tem conteúdo nenhum. Abram este arquivo com qualquer editor de texto e criem um conteúdo. Nós adicionamos o famoso **Olá Mundo**.

A partir deste momento, conseguimos mostrar para vocês os estágios básicos de um commit. O Git possui 3 estágios para a conclusão do processo.

Primeiro estágio – Untracked Files

O primeiro estágio é quando os arquivos são criados na pasta, mas como o arquivo é novo, ele se encontra no estágio de **Untracked Files**, que é o caso do nosso **arquivo.txt**. Isso acontece porque ele é um arquivo desconhecido pelo git, ou seja, ele está na pasta, mas o git ainda não está controlando a versão do mesmo.

Para verificar este estágio basta digitar o comando `$ git status`. Este comando listará todos os arquivos novos e que sofreram modificações.

Segundo estágio – Changes to be committed

Para o nosso arquivo de exemplo passar para o segundo estágio, precisamos rodar um comando que faça ele ser reconhecido pelo git. Veja, o comando abaixo:

```
$ git add arquivo.txt
```

Depois que rodamos este comando, o arquivo.txt não é mais untracked file, ele passa para uma posição intermediária. Para verificarem, rodem, novamente, um `$ git status`.

Nosso arquivo está no segundo estágio, que é conhecido como **Changes to be committed**, ou seja, os arquivos que estiverem neste estágio, estão prontos para serem commitados. Quando falamos commitados, estamos querendo dizer que este arquivo fará parte, **efetivamente**, do repositório, após este procedimento. Quando executamos um commit, o git cria um hash ou identificação deste commit.

Terceiro estágio – Committed

O terceiro estágio é quando o arquivo já foi commitado. Então rodaremos o comando abaixo para que o nosso arquivo de exemplo faça parte deste último estágio.

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

Observem que estamos rodando o comando **commit** e passando um parâmetro **-m** que serve para adicionarmos uma mensagem de identificação para nosso commit. É esta mensagem que nos permite acompanhar todos os estágios do desenvolvimento de um projeto. Ela deve ser muito explicativa, para quem ler, saber o que realmente foi feito naquele commit.

Neste estágio, se rodarem o comando `$ git status`, verão que teremos uma mensagem dizendo que não existe mais nada para commitar, é como se o processo tivesse sido finalizado e o git está limpo de qualquer interferência. Nós só conseguiremos realizar o procedimento, novamente, caso um novo arquivo seja adicionado, removido ou modificado. Caso um destes três fatores ocorram, teremos o **git status** retornando processos a serem efetivados.

Verificando histórico de commits

Para saberem se este arquivo foi comitado e para verificarem outros commits, anteriores, basta rodarem o seguinte comando:

```
$ git log
```

Este comando retornará uma listagem dos últimos commits com o hash de identificação, a data e também o usuário que foi responsável pelo commit, que no caso será o seu usuário, que configuramos no capítulo anterior.

Vale a pena falar que, em um commit, podem haver um ou mais arquivos que foram criados ou modificados. Em nosso exemplo, commitamos apenas um arquivo, mas poderiam ser vários, e o procedimento seria o mesmo.

procedimento seria o mesmo.

Alterando um arquivo do controle de versão

Para que entendam como funciona a atualização de arquivos, que já façam parte do controle de versão, abram o arquivo de exemplo e façam qualquer alteração nele. Depois de fazerem a alteração, salve e feche-o.

Em seguida, acessem o terminal e rodem o comando `$ git status`. Verão que o git informará que este arquivo faz parte do controle de versão e que ele foi modificado. A partir deste ponto vocês podem retornar ao segundo estágio, basta que o adicionem, novamente, rodando o comando `$ git add arquivo.txt`.

Depois de rodarmos este comando, o adicionamos ao segundo estágio, ou seja, preparamos, novamente, para um segundo commit e é o que faremos agora:

```
$ git commit -m "Alterando o arquivo.txt"
```

Nunca conseguimos pular um dos estágios. Não conseguiremos commitar, sem antes ter adicionado um arquivo ou vários arquivos.

Depois de commitar, novamente, podemos rodar `$ git status` e teremos a seguinte mensagem: **nothing to commit, working directory clean**. Isso quer dizer que, não existem mais processos em aberto para serem comitados e que estamos com o diretório limpo, ou seja, todos arquivos que estiverem lá dentro estão versionados pelo git.

Para analisar os commits já efetuados, lembrem de rodar o comando `$ git log`. Vocês conseguirão ver os dois commits efetuados.

Realizando primeiro commit

Passaremos algumas dicas para que possam concluir o commit sem ter aquele medo que todos tem, quando começam a trabalhar com um sistema de controle de versão.

Para isso, criaremos um novo arquivo dentro da pasta **aulagit**, que criamos anteriormente. Criaremos o arquivo com o nome de **teste.php**. Em seguida, rodaremos o **git status**, para que possam ver o primeiro estágio de **Untracked File**, por se tratar de um arquivo novo e não fazer parte do controle de versão.

O próximo passo é rodar o comando que prepara o(s) arquivo(s), para serem comitados.

```
$ git add teste.php
```


Desta forma, já temos nosso arquivo pronto para ser comitado. Neste ponto, temos duas opções:

1. `$ git commit -m "descricao"`
2. `$ git commit`

Observem que, na primeira opção você pode adicionar uma descrição, diretamente no comando, para representar o commit. Mas no dia a dia, verão que, dificilmente, será desta forma, porque, normalmente, precisamos comitar mais do que um arquivo de uma só vez, logo, a segunda forma é mais indicada.

Quando rodamos o commit, sem passar mensagem, o terminal abrirá um editor de texto forçando que coloquemos uma descrição para o commit. Geralmente, no mac, pode ser que abra com o editor Vim, no Linux pode ser o Nano e no Windows, um bloco de notas. O importante é saberem que o git abrirá o editor padrão, do sistema operacional, para que seja feita a descrição do commit.

Ao abrir o editor, ele mostrará uma lista de todos os arquivos que estão preparados para serem comitados, facilitando assim a sua descrição. Quando utilizam a primeira forma, com o **-m**, vocês podem esquecer algum arquivo que foi editado e acabar esquecendo de incluí-lo na descrição.

Quando trabalhamos em um projeto real e comitamos vários arquivos de uma vez, a descrição deve ser a melhor possível para que todos possam saber, exatamente, o que aquele commit representou para aquela etapa do projeto. Tenham muita calma sempre que forem criar esta descrição, façam com muito carinho.

Uma dica, para melhorar a descrição de commits, principalmente com muitos arquivos, é trabalharem com tópicos. Vejam um exemplo:

- Criando arquivo teste.php
- Fazendo o arquivo teste pegar dar um echo teste

Adicionem o comentário da forma acima, salvem o arquivo e feche-o, concluindo o commit. Depois, basta rodar um **git log** para visualizar os commits e perceber que desta forma a descrição fica muito mais fácil de entender. Esta é apenas uma dica, vocês podem criar a descrição da maneira que for melhor para vocês.

Outra dica que podemos passar a vocês, para que vocês possam ganhar tempo e produtividade, é adicionarem todos os arquivos modificados, e untracked files, de uma só vez.

Já possuímos 2 arquivos em nossa pasta. Para ver o exemplo funcionando, editem os dois arquivos, para que os dois estejam no estado de arquivos modificados.

Em seguida, ao invés de adicionar um por um, vocês rodarão o comando abaixo:

```
$ git add .
```

Desta forma, vocês adicionam todos os arquivos, de uma só vez, preparando-os para serem commitados. Imaginem quando existem 10 arquivos, ou mais, para serem adicionados. Esta dica pode ser muito útil, mas cuidado, vocês podem se perder, caso exista algum arquivo que não era pra ser adicionado ao commit. Desta forma, vocês ganham tempo e caso cometam algum erro, terão como reverter.

Existe uma outra maneira de agilizar o processo e comitar todos arquivos de uma vez. Vejam o comando abaixo:

```
$ git commit -a -m "Comitando arquivo e teste juntos"
```

O comando **-a** faz com que os arquivos sejam todos adicionados automaticamente, passando-os para o segundo estágio e já comita passando mensagem. Todo processo é feito automaticamente. Só rodem este comando, quando tiverem certeza de que todos os arquivos pertencerão ao controle de versão. Caso contrário, terão que voltar ao commit anterior, para fazer alterações.

Quando a maioria dos arquivos deva ser adicionada ao segundo estágio para serem comitados, mas não todos, vocês podem adicionar os arquivos, manualmente, ou podem adicionar tudo e remover, apenas, o que não faz parte.

Para remover um arquivo que já foi adicionado, rodem o comando abaixo:

```
$ git reset HEAD --nome_do_arquivo
```

O **reset HEAD** é responsável para desfazer o comando add.

Vocês podem praticar e fazer seus commits para fixarem o conteúdo.

Acreditem! Vocês utilizarão estes comandos, o tempo todo. Pratiquem muito.

Verificando log

Já vimos o comando **git log** e sabemos que ele lista os commits efetuados.

O git log é uma ferramenta, extremamente, importante e poderosa para utilizarem.

Quando rodamos um git log, conseguimos ver os logs de todos os commits, por ordem decrescente.

O grande ponto é que, vocês devem dominar este comando. Fiquem tranquilos, não estamos pedindo para que vocês decorem todas as opções que este comando oferece, anotar, algumas delas, ajudaria bastante.

A primeira opção que mostraremos é a seguinte:

```
$ git log -p
```

Este comando, além de mostrar os logs, mostrará a alteração que foi feita em cada arquivo comitado. Costumamos chamar estas alterações de diferenças entre o arquivo, antes do commit e depois do commit.

Este comando, ao mesmo tempo que é muito bom, pode acabar se tornando chato, porque vocês podem não querer ver todos os logs e alterações. Para isso, vocês tem a opção de rodar o seguinte comando:

```
$ git log -p -2
```

Neste caso, estamos limitando o número de resultados que o comando mostrará. Como passamos **-2**, como parâmetro, teremos apenas 2 resultados listados. Este comando é muito utilizado quando queremos saber o que foi alterado no, último ou nos últimos, commits do projeto e esta é uma forma rápida de se fazer isso.

Outro comando que vocês podem utilizar é o **stat**:

```
$ git log --stat
```

Este comando mostra o git log, mas com estatística de todos os commits, dizendo qual arquivo entrou e qual saiu ou qual linha entrou e qual linha saiu.

Vocês podem querer deixar a listagem de logs mais enxuta, removendo alguns dados que poluem o seu terminal. Para isso rodem o seguinte comando:

```
$ git log --pretty=oneline
```

Com este comando, teremos todos os logs de uma forma mais amigável, sendo apresentado um por linha.

```
3b8b671e28fad9c4294baef05bf42f5ab3b931f2 Comitando arquivo e teste juntos
a7e09ab07b1f6622a49ebfb59fc76c77358d6caf - Criando arquivo teste - Fazendo arquivo te
ste.oho dar um echo teste
db199caf4d052f4202f5565092887b9b139a9de1 Alterando arquivo.txt
35b7e3166118e41b66cf463bc677880a88840706 Meu primeiro commit
```

Observem que ele mostra, apenas, o hash e o comentário.

Existe outro tipo de comando, utilizando o **pretty**.

```
$ git log --pretty=format:"%h - %an, %ar : %s"
```

```
3b8b671 - Wesley Willians, 11 minutes ago : Comitando arquivo e teste juntos
a7e09ab - Wesley Willians, 13 minutes ago : - Criando arquivo teste - Fazendo arquivo
```

```
teste.oho dar um echo teste
db199ca - Wesley Willians, 14 minutes ago : Alterando arquivo.txt
35b7e31 - Wesley Willians, 15 minutes ago : Meu primeiro commit
```

Observem que esta, é uma forma simplificada de mostrar os logs, porém com mais informações.

Lembrando que vocês não devem decorar todos estes commits, basta saberem que existem estas opções de visualizações e com o tempo vocês gravarão os comandos mais utilizados.

Temos outro comando, muito útil, para quando precisamos do histórico de commits. Suponham que precisem saber quais foram os commits dos dois últimos dias.

`$ git log --since=2.days` ou `$ git log --since=2.weeks`, caso queiram saber as duas últimas semanas.

Gostaríamos de dizer que existem muitas outras opções para o comando git log. Estes, são os mais utilizados.

Como já falamos, não tentem decorar, mas pratiquem muito e deixem alguns comandos, principais, anotados para que possam lembrar, até que estes comandos se tornem familiares a vocês.

Ignorando arquivos

Falaremos dos arquivos indesejados em nossas aplicações, ou seja, existem arquivos ou pastas, que não queremos que façam parte do controle de versão. Porém, estes arquivos podem ser alterados com frequência. Suponham que, toda vez, vocês tenham que adicionar e comitar arquivos que não eram para ser comitados.

Um exemplo clássico destes arquivos que sempre são criados e que não queremos que façam parte do controle de versão, são os arquivos e pastas criados pelas IDEs.

A IDE PHPStorm, por exemplo, cria uma pasta chamada **.idea**, quando criamos um novo projeto. Podemos ignorar este arquivo para que ele não seja reconhecido, nem como untracked file, nem como arquivos modificados, em nosso controle de versão.

Para isso, basta criarmos um arquivo oculto, chamado **.gitignore**, na pasta raiz do projeto. Dentro deste arquivo, adicionaremos todos os arquivos ou pastas, que deverão ser ignorados em nossa aplicação. Desta forma, os arquivos nunca serão comitados e não farão parte do controle de versão.

Vejam o exemplo:

```
.idea/
```

Cada arquivo ou pasta, deverá ser adicionado em uma linha diferente. Em nosso exemplo, estamos

ignorando, apenas, a pasta da IDE, chamada **.idea**, mas vocês podem ignorar quantos arquivos quiserem ou forem necessários.

Para testarem no projeto, criem um arquivo chamado **gitignore.txt**. Em seguida, acessem o terminal e rodem **git status**. Vocês verão que ele se encontra como **untracked file**, porque acabamos de criar.

Agora, no arquivo .gitignore, adicionem este arquivo, para que seja ignorado.

```
.idea/  
gitignore.txt
```

Depois de adicionado o arquivo no .gitignore, rodem o comando **git status**, novamente. Vocês poderão ver que o arquivo, que antes era **untrackad file**, não consta mais para ser comitado. Isso ocorre porque o Git ignora o arquivo e não o reconhece.

Vocês encontrarão o arquivo .gitignore, que criamos para listar os arquivos ignorados, este sim deverá fazer parte do projeto e ser adicionado ao controle de versão.

Vocês acabaram de aprender o conceito do arquivo .gitignore. Para trabalharem com ele, basta adicionarem os arquivos, que deverão ser ignorados, um em cada linha, dentro deste arquivo.

Brincando com fluxo de commits

Suponham que, tenham adicionado o arquivo errado e queriam reverter esta situação. Ou que tenham adicionado todos os arquivos, de uma vez, e queiram remover, apenas um, que não fará parte do commit. Para estes casos, existem os comandos **reset HEAD** e **checkout**. O **reset HEAD**, serve para voltar **untracked files** e o **checkout**, para voltar arquivos que já foram adicionados e estão modificados.

Fiquem atentos, o Git sempre é muito informativo e nos ajuda o tempo todo. Ele sempre trará estas possibilidades e dicas.

Como criamos um novo arquivo, vamos utilizar o **reset HEAD**. Vejam o comando abaixo:

```
$ git reset HEAD teste2.txt
```

Depois de rodarem este comando, se rodarem o **git status**, verão que ele sai do segundo estágio e volta ao primeiro, como **Untracked**.

Simularemos um outro caso, muito normal no dia a dia do programador, que é adicionar todos os

arquivos de uma vez e remover, apenas, os que não fazem parte. Suponham que temos dez arquivos, oito deles devam fazer parte do commit e dois não. Logicamente que, é mais fácil adicionarmos todos e removermos, apenas dois, do que ter que adicionarmos oito, manualmente. Vamos fazer este teste, mas com menos arquivos.

```
$ touch teste3.txt
```

```
$ touch teste4.txt
```

Se rodarmos o `git status`, teremos 3 arquivos no estágio `untracked files`. Adicionaremos todos, com o comando `$ git add .` e depois removeremos, apenas, um deles.

```
valls:aulagit son$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   teste2.txt
    new file:   teste3.txt
    new file:   teste4.txt

valls:aulagit son$
```

Removeremos o `teste2.txt`, para que ele não faça parte do commit. Notem que, o próprio Git está passando o comando para remover.

```
$ git reset HEAD teste2.txt
```

```
valls:aulagit son$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   teste3.txt
    new file:   teste4.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    teste2.txt
```

Após prepararem todo o commit, deixando somente os arquivos que deverão fazer parte dele, basta realizarem o commit. Saibam que, depois do commit, vocês não conseguem, fazer mais nada, somente voltar a versão anterior.

Façam tudo que tiverem que fazer, antes do commit.

Voltando versões

Anteriormente, adicionamos 3 arquivos e depois removemos, um deles, utilizando o **reset HEAD**.

Neste momento, estamos com a mesma situação, em nosso terminal.

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

    new file:   teste3.txt
    new file:   teste4.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    teste2.txt
```

Falaremos sobre voltar versões, porque esta é a grande finalidade de trabalhar com versões, se não fosse isso, não teria sentido trabalhar com elas.

Existem diversas formas de voltar uma versão. Umas mais simples, outras mais complicadas. No final, temos o mesmo efeito, que é a volta para alguma versão específica.

Uma das maneiras mais simples é, utilizando o comando **checkout**, passando o hash do commit. Vejam o exemplo abaixo:

```
commit a7e09ab07b1f6622a49ebfb59fc76c77358d6caf
Author: Thiago Valls <thiago@schoolofnet.com>
Date:   Thu Mar 9 04:23:23 2017 -0300

- Criando arquivo teste
- Fazendo arquivo teste.oho dar um echo teste
```

```
$ git checkout a7e09ab07b1f6622a49ebfb59fc76c77358d6caf
```

Observem que pegamos o hash, de um dos commits listados pelo git log. Vocês, com certeza, terão outro hash na listagem, basta executarem o mesmo comando, com algum hash de commit, da listagem.

Após rodarem este comando, temos uma mensagem informando que o nosso estado atual está apontando para o commit que informamos no comando checkout.

```
A teste3.txt
```

```
A teste4.txt
```

```
Note: checking out 'a7e09ab07b1f6622a49ebfb59fc76c77358d6caf'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

```
HEAD is now at a7e09ab... - Criando arquivo teste - Fazendo arquivo teste.oho dar um echo teste
```

Notem que não voltamos, ainda não assumimos, totalmente, este commit como sendo o atual. Para que pudéssemos trabalhar com os arquivos atuais e os arquivos do commit, que selecionamos, o Git cria uma nova camada, chamada **branch**. Falaremos mais sobre branches, mas gostaríamos de mostrar que isso é verdade. Para isso, rodem o comando abaixo:

```
$ git branch
```

Vocês terão o seguinte resultado:

```
* (HEAD detached at a7e09ab)
  master
```

Geralmente, todas as alterações feitas, são no branch **master**, mas vocês podem ver que estamos trabalhando em outro branch, o qual está marcado com um asterísco.

Vamos supor que já viram o que tinham que ver nesta versão que acessaram, basta retornar para a versão master, e ignorar este branch que foi criado.

```
$ git checkout master
```

Outra forma de acessar commits anteriores, é com o seguinte comando:

```
$ git reset HEAD~1 ou $ git reset HEAD~2 ou $ git reset HEAD~3
```

Observem que, vocês voltam, quantos commits quiserem, basta alterarem o último número que é o índice do HEAD. Colocando 1, vocês retornam um commit, 2 vocês retornam dois, assim por diante.

índice do HEAD. Colocando 1, vocês retornam um commit, 2 vocês retornam dois, assim por diante.

Porém, existem dois tipos de **reset HEAD**: soft e hard.

```
$ git reset HEAD~1 --soft ou $ git reset HEAD~1 --hard
```

Soft: remove todos os commits, posteriores, voltando-os para o estágio anterior que é **changed to be committed**, mas ele os mantém na pasta, para que possam ser adicionados, novamente.

Hard: remove todos os commits, posteriores à versão acessada, e também, os arquivos que a eles pertenciam.

Isso significa que, vocês devem tomar muito cuidado com o parâmetro **hard**, a não ser que queiram ignorar todas as alterações feitas, posteriormente, àquela versão que você acessou.

Desta forma, vocês podem trabalhar com as “idas e vindas” de suas versões. Basta praticarem, bastante, e fazerem testes para fixarem o conteúdo.

Falando sobre branches

Este assunto é um pouco mais complexo, mas com certeza vocês entenderão, tranquilamente.

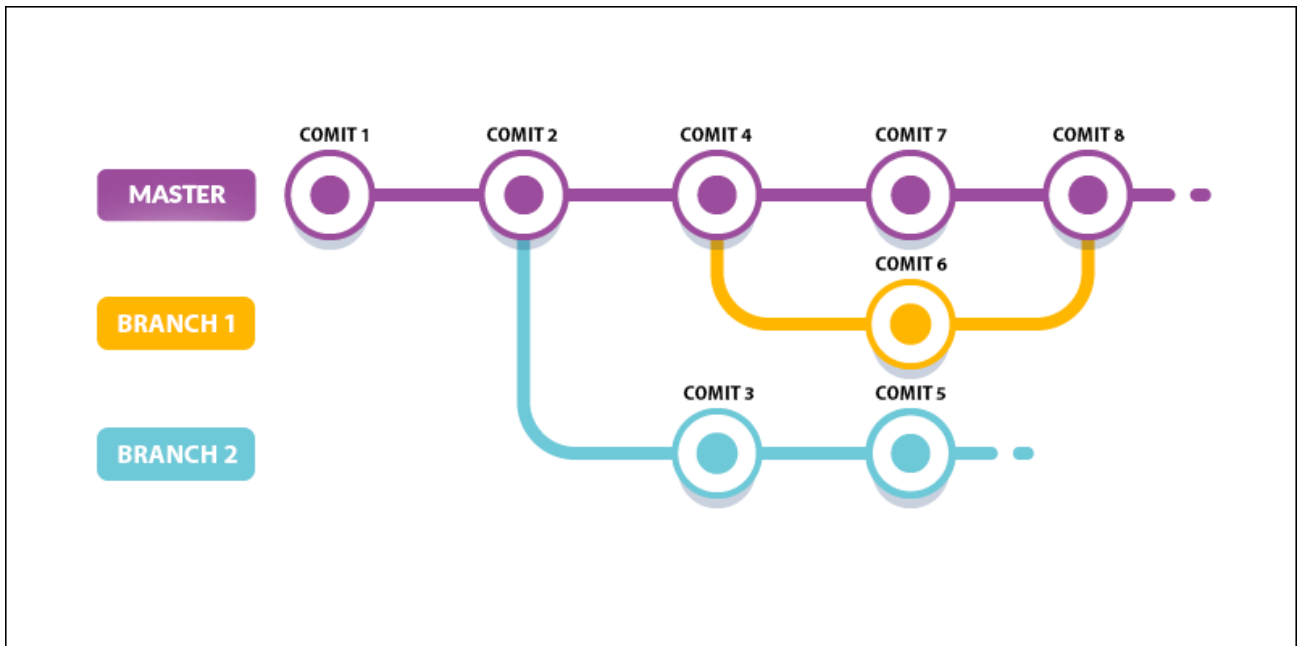
Antes de falarmos sobre branches, de forma técnica, tentaremos mostrar o conceito que existe por trás.

O Git trabalha como se fosse uma linha do tempo e esta linha seria o nosso branch principal, que é conhecido como **master**. Cada commit que fazemos, esta linha do tempo é marcada com este commit.

Suponham que estão desenvolvendo um software e precisam fazer alterações de layout, para o cliente aprovar, e precisam fazer várias cores. Vocês não querem parar o desenvolvimento do software para definição de layout. Neste ponto que o branch é muito indicado, porque vocês criarão uma ramificação do projeto, que trabalhará, independentemente, do projeto principal e não interferirá no desenvolvimento.

Outro exemplo seria: se um design e um programador estivessem trabalhando juntos no branch master, ou pior ainda, se uma equipe inteira estivesse trabalhando, todos no branch master. Cada um, fazendo seus commits, aleatórios, e vocês sem saberem se alguma alteração, que fizeram, quebraria o código.

Temos uma imagem que pode ilustrar muito bem o processo e o caminho de um projeto com branches.



Vejam, na imagem acima, que temos três branches, trabalhando de forma simultânea. O master e mais dois branches, em paralelo.

Criar um branch é fazer um novo commit, contendo todos os commits, anteriores a ele. Então o commit 3, do branch 2, é igual ao commit 2 do branch master, porém o commit 5, tem tudo que o commit dois tem e mais as alterações.

Observem que o commit 8, do branch master, ainda não possui o código do commit 5 do branch 2, por exemplo. Porque ainda não rodamos o comando **merge**.

Vejam que o commit 6, que está no branch 1, foi criado, modificado e no commit 8 foi feito o merge. Isso quer dizer que, o branch 1 já pode ser excluído, porque ele já executou o seu papel e já mesclou com o branch master, fazendo com que o commit 8 possua o código do commit 6. O mesmo não ocorreu, ainda, com o branch 2, que ainda está em desenvolvimento. Assim que for concluído, poderá ser mesclado com o branch master, através do comando merge.

Resumindo, trabalhar com branch é trabalhar em um mesmo projeto de forma assíncrona, de modo que, muitos desenvolvedores possam trabalhar em um mesmo projeto, sem um atrapalhar o outro. O desenvolvimento de várias funcionalidades serão possíveis, sem atrapalhar o fluxo normal de desenvolvimento.

Notem que os commits não são independentes, sempre são progressivos.

Podemos dar exemplos mais complexos. Suponham que, em uma aplicação, seja necessário criar um sistema de cadastro de produtos. Tudo bem, criamos um branch e o desenvolvedor começará a desenvolver, enquanto o restante da aplicação continua no branch master.

Suponham que, temos outro desenvolvedor que criará um sistema de estoque e este sistema dependa do cadastro de produtos. Criaremos um branch para o desenvolvimento do sistema de estoque, mas o branch não será em relação ao master desta vez, este branch terá o branch do sistema de produtos, sendo o master para ele. Criamos uma ramificação da ramificação ou branch de branch.

Esta ramificação não tem limite, vocês podem ramificar o projeto de acordo com a necessidade que tiverem. O merge continua do mesmo jeito, mas, geralmente, costumamos dar um merge, de acordo com a hierarquia. Primeiro, daríamos um merge do sistema de estoque para o sistema de produtos e depois, outro merge, do sistema de produtos para o branch master, que é a aplicação raiz.

Observem que é um assunto mais complexo, mas não é complicado para entender. Basta projetarem, muito bem a aplicação, e dividirem, corretamente, a equipe. Desta forma, o projeto caminha em paralelo, com diversas funcionalidades sendo desenvolvidas, ao mesmo tempo, e de forma organizada. O que é mais interessante, tudo versionado, podendo retornar e arrumar erros, em qualquer ponto do projeto.

Criando primeiro branch

Falamos bastante sobre a ideia principal dos branches e agora veremos a criação.

No repositório atual, temos os seguintes arquivos:

- arquivo.txt
- teste.php
- teste2.txt
- teste3.txt
- teste4.txt

Suponham que este é o nosso projeto e que precisamos criar uma ramificação para o desenvolvimento de uma funcionalidade qualquer, ou seja, um novo branch.

```
$ git checkout -b funcionalidade1
```

Este comando é responsável pela criação do branch. É muito importante não esquecerem de que o branch é criado de acordo com o branch em que você estiver, na hora do comando. Se estiverem no master, criarão um branch a partir do master, se estiverem em outro branch, e derem o comando,

criarão um branch a partir deste branch e, assim por diante. É assim que os projetos vão se ramificando.

Uma forma de saberem em qual branch estão é rodar o comando: **\$ git branch**

Com este comando, tivemos o seguinte resultado:

```
* funcionalidade1  
master
```

Isso acontece porque, quando criamos o branch, ele já acessa, automaticamente. Quando criamos um novo repositório, sempre estaremos no branch master.

Estando no branch funcionalidade1, criaremos um arquivo chamado **funcionalidade1.txt**. Após criarem este arquivo, adicionem-o e deem um commit. Em seguida, voltem para o branch master, com o seguinte comando:

```
$ git checkout master
```

O comando para acessar e criar é o mesmo, mas para criação, passamos o parâmetro **-b**.

No branch master, listem os arquivos, utilizando o comando **\$ ls**. Pronto, vocês verão que este arquivo não faz parte deste branch master, mas sim do branch funcionalidade1, somente. Não estamos falando, apenas, do arquivo estar ou não, fisicamente, na pasta. Estamos falando do **log**, de forma que, se listarem os logs **\$ git log**, vocês não verão o commit que fizeram no outro branch.

O mesmo ocorre se fizerem uma alteração em qualquer arquivo do branch master e comitar. Esta alteração não será enxergada no branch funcionalidade1 e nem fará parte do código.

Cada branch será, totalmente, independente, até que seja feito um merge.

Trabalhar com branch não tem muito segredo e não existem muitos outros comandos, além dos que apresentamos. O mais importante é entenderem a importância que eles tem para um projeto e o quanto eles auxiliam o desenvolvimento em equipe.

Não existem limites para os branches. Vocês podem criar quantos quiserem. Nós indicamos a criação de um branch, sempre que forem desenvolver uma nova funcionalidade, para que não fiquem criando branch, desnecessariamente.

Lembrando que, existir um commit dentro de um branch, não significa que os arquivos estejam naquele branch, eles apenas estão apontados, muitas vezes.

Merge e Rebase

No capítulo passado, vocês aprenderam a criar branches e a trabalhar com eles. Viram a importância dos mesmos.

Merge

Mostraremos como fazer para mesclar os branches, para que as funcionalidades se somem em um mesmo branch. O comando para fazer esta mesclagem chama-se **merge**.

Temos o branch funcionalidade1 e o branch master, em nosso exemplo.

Existem dois tipos de merge: um mais simples e outro um pouco mais complexo. O mais simples é quando criamos um branch e fazemos alteração apenas no branch. Suponham que tivéssemos criado a funcionalidade e no branch master não tivéssemos feito mais nada, nem ter dado commit. Esta é a situação mais simples e o branch mais fácil que existe.

Mas, não foi o que aconteceu em nosso exemplo. Lembrem-se que, no capítulo anterior, além de criarmos um arquivo no branch funcionalidade1, pedimos que alterassem um arquivo, no branch master. Temos alterações e commits, nos dois branches.

Isso significa que, o branch funcionalidade1 não possui um dos commits que o master possui. Acessem o branch master e rodem o comando abaixo:

```
$ git merge funcionalidade1
```

Este comando se comporta como um commit e abre um editor para que adicionemos uma mensagem de identificação. Depois de rodarem este arquivo teremos, tanto os commits existentes no branch funcionalidade1, quanto os arquivos que lá foram criados.

Poderíamos ter feito o inverso, de dentro do branch funcionalidade1, rodar um merge no branch master.

Existe uma forma de gerar um merge, sem que ele gere um commit, porque nem sempre é legal gerar um commit novo a cada merge, no projeto. Este comando só pode ser rodado, localmente, para que não gere confusão para outros usuários do repositório.

Rebase

Para exemplificar o rebase, removeremos o último commit, relacionado ao merge.

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

Voltamos o último commit, temos o mesmo cenário, do início do capítulo anterior.

Lembrando a ordem que fizemos a alteração, podemos dizer que temos primeiro um commit no branch funcionalidade1 e depois um branch no master. Porque, logo que criamos o branch funcionalidade1, já criamos o arquivo e fizemos o commit. Em seguida, fomos para o master, alteramos um arquivo e fizemos o commit.

Por que estamos falando de ordem? Porque o rebase fará a mesma coisa que um merge, mas levando em consideração a ordem dos commits. Por este motivo que ele deve ser feito, somente, em suas máquinas, porque ele fará o merge na sequência correta. Isso quer dizer que, quando vocês derem um **git log**, terão a sequência, exata, de commits que realizaram.

Suponham fazer isso, em um projeto, sem ser localmente. O commit de um desenvolvedor pode ir parar 10 logs abaixo de outro, e isso pode bagunçar o projeto e confundir muita gente, mas, feito localmente, não terão problema.

```
$ git rebase funcionalidade1
```

```
First, rewinding head to replay your work on top of it...
Fast-forwarded master to funcionalidade1.
```

Observem que, primeiro ele organizou os commits, para depois fazer o mesmo trabalho do merge, unindo as modificações do branch funcionalidade1 com o master.

Para verificarem este processo, basta rodarem um git log e terão o commit do branch funcionalidade1, antes do commit do master. Vale lembrar que, não haverá mais o commit do merge. Isso significa que estamos gerando um commit a menos.

Falando sobre o github

Falaremos sobre o Github, que é um serviço disponibilizado na internet e é muito útil para os desenvolvedores e comunidades, de diversos segmentos.

O Github é um serviço online, que disponibiliza repositórios Git, ou seja, é uma forma de trabalhar com o Git, mas de forma centralizada. Podemos desenvolver a aplicação, localmente, e quando quisermos, podemos subí-la para um repositório online e centralizar o conteúdo.

Desta forma, podemos manter o código fora da nossa máquina e também colaborar com outras pessoas. O mais legal do projeto é que hoje, o Github é o serviço online, mais utilizado pelos

desenvolvedores, porque ele, simplesmente, dá uma conta gratuita, para os usuários que pensam em desenvolver códigos open sources. Ele acaba sendo uma comunidade. Vocês conseguem seguir repositórios, criar seus próprios repositórios e inúmeras outras, funcionalidades.

Indicamos o contato com o Github, caso nunca tenham ouvido falar, porque a partir de agora, que aprenderam a trabalhar com Git, vocês podem começar a desenvolver seus projetos de forma organizada. Se for um projeto próprio e particular, existem repositórios privados, que vocês podem contratar o serviço. Nunca mais vocês correrão o risco de manterem o código em suas máquinas e perderem todo trabalho desenvolvido. Mantenham a cópia local e em um repositório online, sempre. Utilizando o Github.

Se vocês ainda não possuem uma conta no Github, pedimos para criarem uma, agora mesmo. Para vocês se acostumarem com a ferramenta. Leiam os conteúdos sobre, mexam muito nos recursos, para que vocês possam conhecer todos os recursos disponíveis, neste serviço maravilhoso.

Quando vocês logam no Github e acessam a página inicial, vocês tem o controle de todos os commits que já executaram. Conseguem ver quantas pessoas o seguem, como estão suas contribuições, com outros projetos e etc.

Com o Github, vocês acabam tendo a oportunidade de participar de grandes projetos, open sources, e acaba sendo uma vitrine enorme, para seus trabalhos. Também, às vezes, tem a oportunidade de participarem de um projeto real, que é muito mais vantajoso do que muitos outros projetos pessoais ou treinamentos e até mesmo, faculdades. Nem sempre isso acontece, mas é sempre muito bom, fazer parte de outros projetos e ajudá-los.

O único problema é que, para você poder participar destes projetos open sources, vocês precisam saber das técnicas e dinâmicas do Github. É esta base que daremos, para que possam contribuir com o projeto que acharem melhor e mais proveitoso, para o crescimento de vocês.

Criando chave para o github

Mostraremos como fazer para gerar uma chave pública, para o Github. Se vocês nunca ouviram falar, não se preocupem, explicaremos.

Toda vez que vocês criarem um repositório no Github e forem fazer um **push**, o Github pedirá autenticação. Como somos desenvolvedores e precisamos enviar estes arquivos o tempo todo, não seria legal ficar digitando toda vez, no terminal, seu login e senha, para conseguirem efetuar este push.

Em vez de fazerem isso, criaremos uma chave de autenticação.

Para este processo rodaremos um comando, em nosso terminal, que gerará dois arquivos.

1. Uma chave criptografada privada
2. Uma chave criptografada pública

Esta chave privada, sempre, ficará no computador e vocês, jamais, compartilharão com ninguém. A pública, vocês podem compartilhar com diversos serviços, inclusive o Github.

Compartilhando esta chave com o Github, sempre que forem executar qualquer operação no Github, o serviço verificará se a chave pública confere com a chave privada da nossa máquina. O Github verificará, se realmente, se trata de um par, caso ele identifique, a operação é liberada, sem precisar de login e senha.

Esta autenticação acaba sendo muito mais segura do que trabalhar com usuário e senha.

Gerando chaves no computador

Entre em seu terminal e digite o comando abaixo:

```
$ ssh-keygen
```

Este comando mostrará o caminho em que as chaves serão geradas e pede confirmação. Caso não exista nenhuma chave criada, ele já criará, automaticamente. Se existir, ele perguntará se quer sobrescrever.

Como vocês sabem se a chave está gerada e onde estão os arquivos? Vocês precisam acessar o caminho informado na criação e listar os arquivos internos. Vejam o comando abaixo:

Primeiro `$ cd ~/.ssh/` depois `$ ls`.

Vocês deverão encontrar os arquivos:

- id_rsa
- id_rsa.pub

Observem que o arquivo que tem a extensão **.pub** se trata da chave pública que iremos compartilhar com os serviços. Para pegar o conteúdo deste arquivo, rodem o seguinte comando:

```
$ cat id_rsa.pub
```

E seguida, copiem o código para adicionarem ao Github. Vocês podem abrir o arquivo em algum bloco de notas, se preferirem. O importante é copiarem o conteúdo deste arquivo.

Para cadastrarem no Github, acessem **Account Settings** e depois cliquem em **SSH Keys**. Neste local, observem que existe a possibilidade de cadastrarem diversas chaves. Criem uma nova chave. Vocês podem adicionar uma descrição para cada chave, para saberem a qual máquina ela pertence.

No campo da chave, vocês colarão o conteúdo copiado e clicarão em **add key**. Depois disso, já teremos uma nova chave configurada e, a partir deste momento, todas as operações que forem efetuadas, do computador para o repositório online, não precisarão mais de autenticação por login e senha.

Criando um repositório

Mostraremos como criar um novo repositório no Github, apesar de ser muito fácil.

Agora que vocês já possuem uma conta no Github e já configuraram a chave de segurança, basta começar a criarem repositórios e a desenvolver.

Para criarem um novo repositório, basta acessarem o profile e depois clicar em **Repositories**. Neste local, vocês clicam em **New**.

Na próxima página, vocês preencherão os dados do repositório como: nome, descrição, público ou privado e tem a opção de iniciar o repositório com um arquivo README.

O repositório privado só é possível, contratando o serviço do Github. Os repositórios públicos, são gratuitos e ilimitados.

Todo repositório estará no endereço padrão do Github, que será o endereço do Github + seu usuário + nome do repositório. Este é um padrão de URL que representa cada repositório criado.

Quando vocês escolherem entre, iniciar com o arquivo README ou não, existe uma particularidade. Quando não selecionamos nada, nosso repositório é iniciado, totalmente, vazio, inclusive, sem nenhum branch criado, nem o branch master. Neste caso, quando fizermos o primeiro commit, teremos que passar os parâmetros para que este branch seja criado.

Caso vocês selecionem a opção de iniciar com o arquivo readme, o branch já será criado, automaticamente.

Se vocês escolherem a opção de criar com um arquivo, precisarão executar uma clonagem deste repositório, em suas máquinas e, somente, depois, adicionarem os arquivos do projeto para depois, comitá-los.

No primeiro caso, vocês poderão trabalhar da mesma maneira que estamos trabalhando até agora. Criando uma pasta, iniciando o repositório Git e adicionando os arquivos. A única diferença é que,

vocês deverão configurar o repositório remoto, adicionando o endereço/url, do repositório criado.

Vejam o procedimento, que o próprio Github orienta fazer, inicialmente, para deixar o repositório totalmente configurado e linkado.

```
$ touch README.md
$ git init
$ git add README.md
$ git commit -m "first commit"
$ git remote add origin https://github.com/schoolofnetcom/git-code-education.git
$ git push -u origin master
```

Observem que, até a quarta linha, já passamos a vocês, que é o procedimento de criação, adição e commit natural. O segredo de linkar o repositório local com o remoto, está no quinto comando:

```
$ git remote add origin https://github.com/schoolofnetcom/git-code-education.git
```

E o primeiro push, que cria o nosso branch master:

```
$ git push -u origin master
```

Depois destes procedimentos, o repositório já está criado.

Primeiro push

O push, em inglês, é “empurrar”. Para o nosso cenário ele significa, praticamente, isso mesmo. Quando vocês tem o repositório local e querem enviar os commits para um repositório online, para centralizar o conteúdo, vocês podem subir, seus arquivos, para um repositório online, por segurança, ou até mesmo, para compartilharem o código com outros desenvolvedores.

Então, para que vocês consigam trabalhar com o **git push**, precisam adicionar um repositório remoto. Vocês adicionam esta configuração, informando o endereço do repositório online, para que, toda vez que efetuarem um push, o Git saiba para onde enviar os arquivos.

Para criarem este link e configurarem o repositório local, temos o seguinte comando:

```
$ git remote add origin https://github.com/schoolofnetcom/git-code-education.git
```

Vocês podem ver que o comando é bem intuitivo, a única particularidade é o **origin**. Muitos pensam que esta palavra faz parte do comando, mas, trata-se, apenas, de um apelido que vocês darão para o repositório remoto. Sabendo disso, vocês poderiam utilizar qualquer outro nome, mas não é muito recomendada a alteração, uma vez que é uma convenção e a maioria utiliza o origin.

Após rodarem o comando, com o endereço do repositório online, vocês já podem enviar os

arquivos para o repositório online, com o comando push

arquivos para o repositório online, com o comando push.

Para saberem onde o Git guarda estas configurações, informaremos o local de armazenamento.

Abram o arquivo chamado config, que está dentro da pasta oculta do Git, no repositório local.

```
$ vim .git/config
```

```
[core]
  repositoryformatversion = 0
  filemode = true
  bare = false
  logallrefupdates = true
  ignorecase = true
  precomposeunicode = true
```

```
[remote "origin"]
```

```
url = https://github.com/schoolofnetcom/git-code-education.git fetch =
+refs/heads/*:refs/remotes/origin/*
```

Vejam que neste arquivo ele guarda as informações necessárias para o repositório remoto. Todas as configurações, que vocês fizerem no repositório, ficará neste arquivo de configuração.

Agora que já configuramos, enviaremos os nossos arquivos para o repositório online.

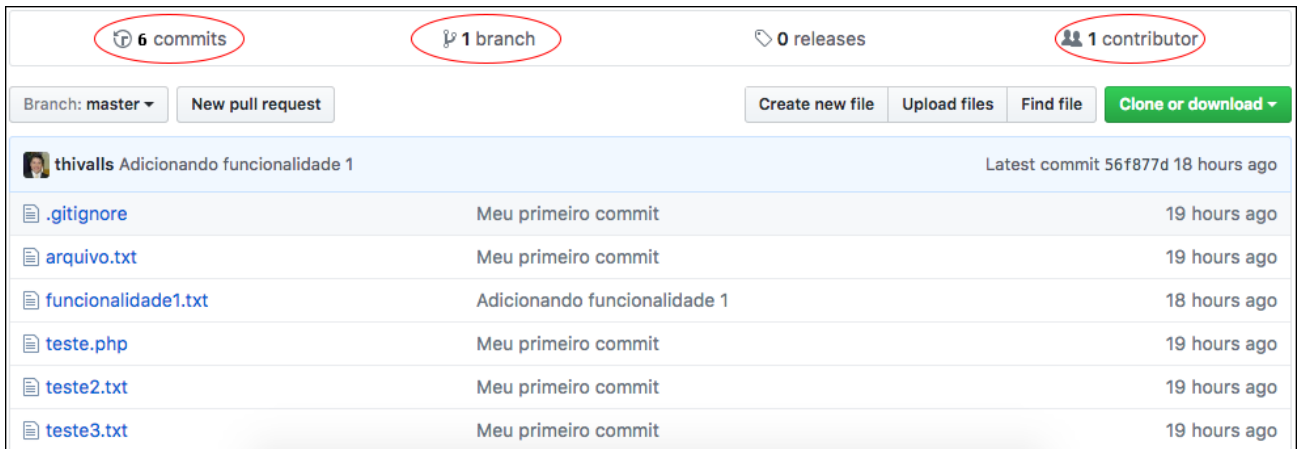
```
$ git push origin master
```

O comando diz: “empurre o meu branch master para o repositório origin”. Isso significa que estamos subindo todos os arquivos e commits para o repositório online.

```
Counting objects: 7, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (7/7), 560 bytes | 0 bytes/s, done.
Total 7 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/schoolofnetcom/git-code-education.git
 * [new branch]      master -> master
```

Vejam que no resultado do comando, o Git informa que foi criado um novo branch, chamado **master**, no repositório online. Em seguida, ele informa que está enviando do branch master local para o branch master online.

Depois de enviar, acessem o repositório online, para verificarem os arquivos e informações.



Commit	Branch	Releases	Contributors
6 commits	1 branch	0 releases	1 contributor

Branch: master New pull request Create new file Upload files Find file Clone or download

thivalls Adicionando funcionalidade 1 Latest commit 56f877d 18 hours ago

File	Commit	Time
.gitignore	Meu primeiro commit	19 hours ago
arquivo.txt	Meu primeiro commit	19 hours ago
funcionalidade1.txt	Adicionando funcionalidade 1	18 hours ago
teste.php	Meu primeiro commit	19 hours ago
teste2.txt	Meu primeiro commit	19 hours ago
teste3.txt	Meu primeiro commit	19 hours ago

Observem que todas as informações de commits e branches foram enviadas, juntamente, com o nosso push. Significa que temos uma cópia do nosso repositório local em um repositório remoto, centralizando nosso projeto.

Dando push em outro branch

Agora que já fizemos o push normal, enviando o branch master para o repositório remoto, ensinaremos como enviar um novo branch.

Temos, em nosso exemplo, um branch chamado **funcionalidade1**. O procedimento é acessar este branch e, em seguida, dar um push deste repositório, para o Github.

```
$ git checkout funcionalidade1
```

Depois de acessarem, rodem o push.

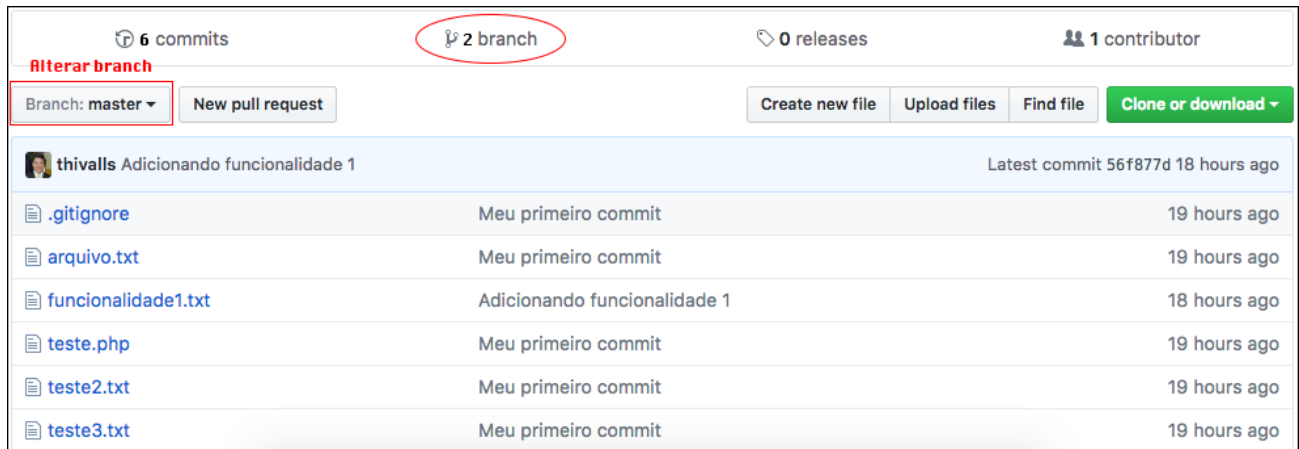
```
$ git push origin funcionalidade1
```

Após rodarem o push, teremos a seguinte mensagem do Git.

```
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/schoolofnetcom/git-code-education.git
 * [new branch]      funcionalidade1 -> funcionalidade1
```

Vejam que ele informou que foi enviado de funcionalidade1 local para funcionalidade1 remoto. Quando não existe o branch, ele cria, automaticamente. Vocês poderiam colocar outro nome no repositório remoto, caso quisessem, mas não é muito indicado, para não haver confusão.

Depois de terem enviado, vamos conferir se, realmente, o branch foi criado, remotamente.



Na imagem, mostramos os dois branches criados e onde vocês podem alterar o branch online.

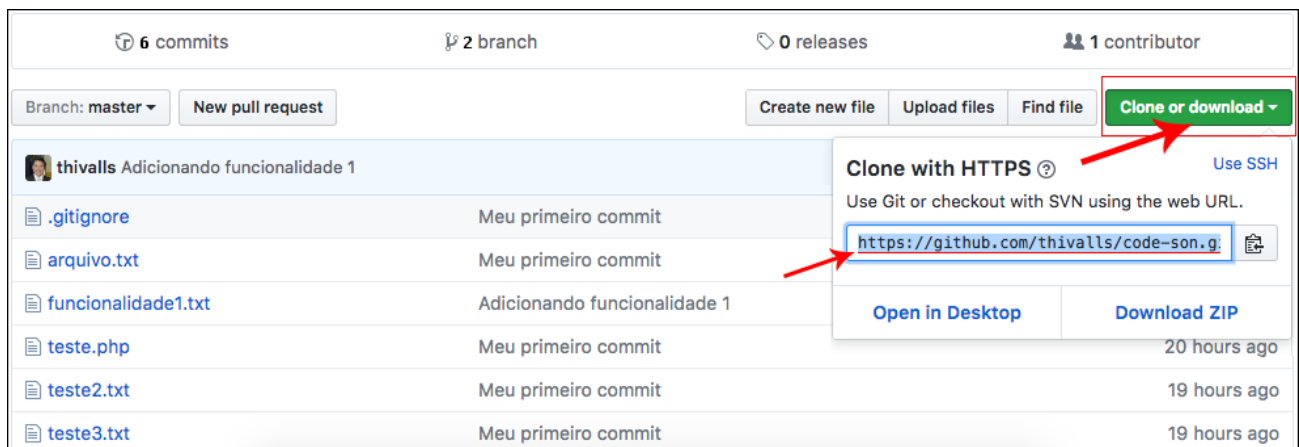
Fazendo clone

Vamos supor que vocês, por engano, apagaram o repositório local, ou estão em outro computador e precisam daquele repositório. Para exemplificar esta situação, apagaremos a pasta **aulagit**, que criamos como exemplo.

```
$ rm -rf aulagit
```

Desta forma, não existe mais nenhum arquivo em nosso repositório na nossa máquina. Para termos, novamente, precisaremos baixar do repositório remoto. Para isto que serve o comando **clone**.

Para conseguirmos rodar o comando, precisaremos do link de clonagem, que é fornecido pelo próprio Github.



Todo repositório terá um link para clonagem e vocês podem encontrá-lo na imagem acima.

Existem dois modos para fazermos a clonagem:

1. Utilizando https
2. Utilizando ssh

No repositório remoto vocês terão o link, para ambos. Com estes dados, vocês podem escolher qual formato preferem. Na maioria das vezes, o mais utilizado é o protocolo **https**, porque o ssh nem sempre está configurado, ou às vezes, pode estar bloqueado pelo proxy.

```
$ git clone https://github.com/schoolofnetcom/git-code-education.git ou $ git clone https://github.com/schoolofnetcom/git-code-education.git aulagit
```

Observem que, colocamos dois exemplos: um com parâmetro e outro sem parâmetro, depois da URL. Este parâmetro é para dar um nome à pasta da clonagem, caso não passem nada, vocês terão o mesmo nome do repositório.

Para clonarmos o repositório, utilizamos o primeiro comando, sem parâmetro, e obtivemos a seguinte mensagem:

```
Cloning into 'git-code-education'...
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 7 (delta 1), reused 7 (delta 1), pack-reused 0
Unpacking objects: 100% (7/7), done.
Checking connectivity... done.
```

Clonar um repositório é relativamente fácil, mas se rodarem o comando **git branch**, verão que, na clonagem só veio o branch master. Existe um outro branch, que deveria fazer parte do clone, agora mostraremos o detalhe.

Quando vocês rodam o comando **git branch**, sempre virão, somente, os branches que estiverem, localmente. Mas, vocês podem ver os branches remotos, também. Basta acrescentarem o parâmetro **-a** no comando. Neste caso, vocês terão os branches locais e remotos.

```
$ git branch -a
```

```
* master
remotes/origin/HEAD -> origin/master
remotes/origin/funcionalidade1
remotes/origin/master
```

Observem que ele mostra os branches locais e os remotos. Sabendo destes dados, podemos fazer a ligação de um branch local com um branch remoto.

Primeiro, vocês devem criar o branch, normalmente, no repositório local, como já aprenderam, só que o comando deve receber um parâmetro a mais, que é o branch que vocês desejam ligar com o repositório remoto. Vejam o comando:

```
$ git checkout -b funcionalidade1 origin/funcionalidade1
```

Vejam que o comando retorna a mensagem, informando que o branch foi criado de acordo com o branch remoto.

```
Branch funcionalidade1 set up to track remote branch funcionalidade1 from origin.  
Switched to a new branch 'funcionalidade1'
```

Se vocês rodarem um `$ ls` já listará os arquivos que estiverem dentro do branch `funcionalidade1`, mas não, necessariamente, os arquivos do branch remoto, `funcionalidade1`, estarão dentro desta pasta. Temos como garantir que todos os arquivos do branch remoto estejam no branch local, rodando o seguinte comando:

```
$ git pull
```

Este comando verifica se todos os arquivos já estão sincronizados, se não estiverem, ele atualizará.

Temos dois branches locais e mais dois branches remotos. Para terem certeza destas quantidades e informações, rodaremos, novamente, o comando **git branch -a**.

```
$ git branch -a
```

E teremos o seguinte resultado:

```
* funcionalidade1  
master  
remotes/origin/HEAD -> origin/master  
remotes/origin/funcionalidade1  
remotes/origin/master
```

Podemos concluir que, existem os mesmos branches locais, no repositório remoto e, o comando ainda traz a informação de que o branch remoto principal, é o `master`, porque ele mostra que o **HEAD** está apontando para o **origin/master**.

Push e Pull com novo branch

Para finalizarmos o conceito de push e pull, faremos uma simulação para facilitar o entendimento.

Suponham que somos uma equipe de colaboradores e temos o repositório **git-code-education**.

Para isso, devemos ter, pelo menos, dois clones de nosso repositório, para simular dois

colaboradores

Criem uma pasta chamada gitCode e movam o repositório clonado, no capítulo anterior, para dentro desta pasta. Em seguida, façam outro clone com o nome de **aulagit**.

```
$ git clone https://github.com/schoolofnetcom/git-code-education.git aulagit
mkdir gitCode
mv aulagit/ gitCode/
mv git-code-education/ gitCode/
```

Nos comandos acima, fizemos todas estas configurações. Temos uma pasta com dois repositórios, idênticos, e conectados ao mesmo repositório, remoto. Suponham que, cada repositório seja um colaborador diferente, com máquinas diferentes e que, cada um, fará uma alteração diferente. Faremos isso para que entendam o processo e como o Github trabalha.

Agora que temos os dois ambientes, faremos uma alteração qualquer, em aulagit, e subiremos para o repositório, remoto. Em seguida, faremos um alteração no git-code-education e tentaremos subir para o repositório remoto, também. Vocês verão que o primeiro push será aceito, mas o segundo não conseguiremos realizar porque, primeiro, teremos que efetuar um comando **pull**, para atualizarmos a alteração feita em aulagit, e somente depois, conseguiremos subir a alteração de git-code-education. Sempre que tiverem alguma alteração no repositório remoto, deveremos atualizar, para depois subirmos qualquer outra alteração, local.

Alteramos o arquivo teste.php, da pasta aulagit, depois comitamos e damos um push, para o repositório remoto. Isso quer dizer que o desenvolvedor da pasta git-code-education não possui, ainda, esta alteração. Para terem certeza disso, abram a pasta git-code-education e o arquivo que vocês alteraram em aulagit. Vocês verão que a alteração feita em aulagit, ainda não existe.

Para fazerem a sincronização, vocês deverão fazer um **pull**, que é o inverso do comando push. Dentro da pasta git-code-education, rodem o comando abaixo:

```
$ git pull origin master
```

Depois de rodarem este comando, vocês podem abrir, novamente, o arquivo e já poderão ver as alterações que fizeram, porque os dois repositórios estão sincronizados.

Quando trabalhamos em equipe, sempre, devemos verificar se o repositório local, está sincronizado com o repositório remoto e, depois vocês podem enviar suas alterações. Caso queiram, somente, atualizar o repositório, antes de começarem a trabalhar, basta rodarem o **git pull**.

Criaremos um novo arquivo, na pasta git-code-education, subiremos para o repositório remoto. Em seguida, entraremos na pasta aulagit e faremos um pull para pegar este novo arquivo. Faremos isso, utilizando um novo branch

UTILIZANDO UM NOVO BRANCH.

```
$ git checkout -b novobranch
$ touch arquivo-novobranch.php
$ git add arquivo-novobranch.php
$ git commit -m "Arquivo - exemplo de novo branch"
$ git push origin novobranch
```

Executamos os comandos acima, em sequência, para conseguirmos exemplificar o processo de criação de um arquivo, em um novo branch. Primeiro, criamos o branch, depois o arquivo, adicionamos, comitamos e, por último, subimos para o repositório remoto, criando um novo branch.

Observem que, antes de darmos o **push**, para o repositório remoto, o branch só existia localmente, depois ele foi criado remotamente, junto com o comando.

Agora, saiam do repositório git-code-education e acessem o aulagit. Vocês poderão ver que não existe o arquivo **arquivo-novobranch.php**, nem o branch **novobranch**.

Como já falamos, quando queremos sincronizar com o repositório remoto, temos que rodar o comando **pull**. Portanto, rodem os comandos abaixo:

```
$ git pull
$ git branch -a
$ git checkout -b novobranch origin/novobranch
```

O comando pull, sincronizou os repositórios. O **branch -a** rodamos para enxergar os novos branches e para sincronizá-los. E, por último, criamos o branch local, com o mesmo nome e linkamos com o branch, online.

Desta forma temos tudo sincronizado, tanto repositório, quanto branches.

Antes do pull

```
* master
remotes/origin/HEAD -> origin/master
remotes/origin/funcionalidade1
remotes/origin/master
```

Depois do pull

```
* master
remotes/origin/HEAD -> origin/master
remotes/origin/funcionalidade1
remotes/origin/master
remotes/origin/novobranch
```

Github – Dicas gerais

Será comum, quando se trabalha em equipe, que alguém peça para fazerem um **Pull Request**.

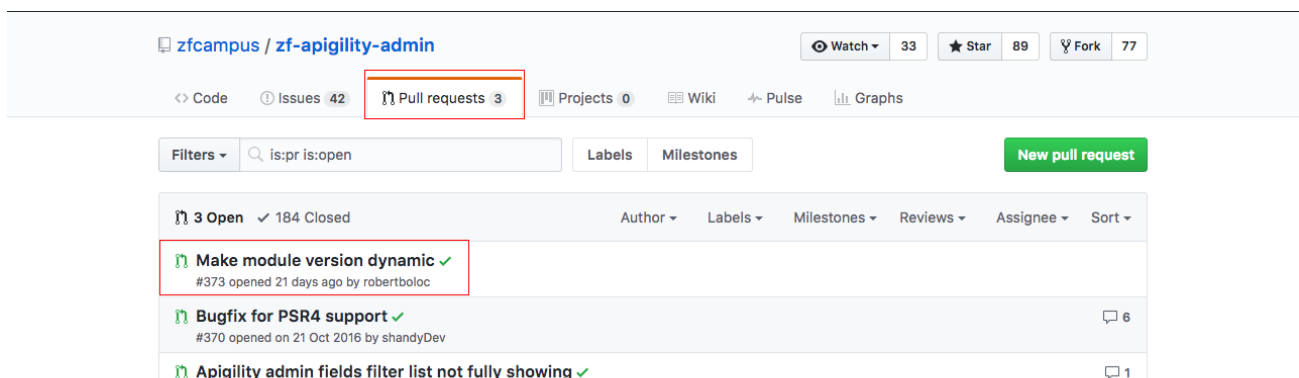
Suponham que o nosso repositório de exemplo seja um projeto e que disponibilizamos online, como um projeto open source. Assim que disponibilizamos, tiveram outros desenvolvedores interessados no projeto e quiseram contribuir com a melhoria do mesmo.

Este desenvolvedor pode contribuir com o projeto, mas a princípio, ele não terá como fazer alterações, diretamente, no projeto, a não ser que vocês deem permissão a ele. Não aconselhamos que isso seja feito. Para esta situação, existe o pull request.

Para começarmos a contribuir com um projeto no Github, em primeiro lugar, precisamos dar um **Fork**. Quando damos um Fork, estamos criando uma cópia do repositório, que nos interessamos, para a nossa conta do Github.

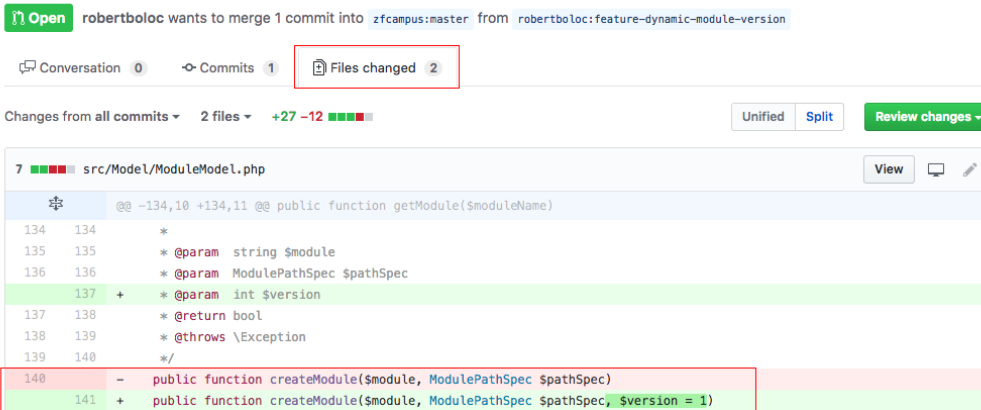
A partir do fork, podemos fazer qualquer alteração que quisermos, porque este é o nosso repositório. E, após as alterações, podemos enviar um pull request, para o repositório original do projeto que demos o fork.

O dono do repositório, comparará o código original dele com suas alterações e se ele achar que foi válido, ele aceitará seu pull request, dando um **merge**, no repositório dele. A partir daí, suas implementações farão parte do repositório original, e você já contribuiu com sua parte.



#352 opened on 8 Aug 2016 by developer-devPHP

Make module version dynamic #373

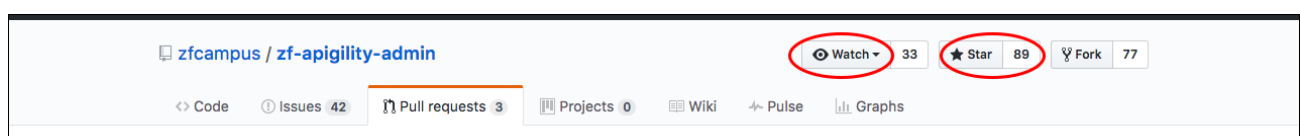


Na primeira imagem, mostramos um exemplo, de um repositório qualquer, que possuem 3 pull requests. Isso quer dizer que são 3 colaboradores querendo aplicar modificações, ou melhorias, no projeto.

Na segunda imagem, mostramos uma parte das alterações que foram feitas. Seriam estas diferenças que o dono do repositório iria avaliar e, caso sejam válidas, ele aprovaria e faria o merge com seu repositório principal.

Esta é uma forma de permitir que outras pessoas melhorem seus projetos e de uma forma segura. Caso vocês identifiquem qualquer código malicioso, vocês só precisam recusar o pull request. Caso seja uma melhoria sadia, vocês podem aprovar, sem problema algum.

Existe uma maneira de acompanharem todas as alterações e evoluções de um projeto no Github. Basta selecionarem o **watch** e o **star**. Com isso, vocês estão informando para o Github que se interessam por aquele repositório e que cada alteração que ele tiver, vocês querem ser informados.



Estas são as dicas, básicas, para que vocês consigam trabalhar com o Github, não só como um repositório para seus projetos, mas como uma comunidade que pode se ajudar e melhorar os códigos, uns dos outros.

Trabalhando com tags

Tags, são marcadores e funcionam como releases do nosso projeto, ou seja, versões. A cada nova

versão, podemos gerar uma tag e, desta forma, as pessoas conseguem ter acesso às versões anteriores, do projeto.

Tags podem ser vistas como ponteiros, que apontam para um determinado commit. Quer dizer que, uma determinada tag aponta para um determinado commit e sua atual situação e arquivos.

Gerando a primeira tag

Acessem o branch master do projeto e rodem o seguinte comando.

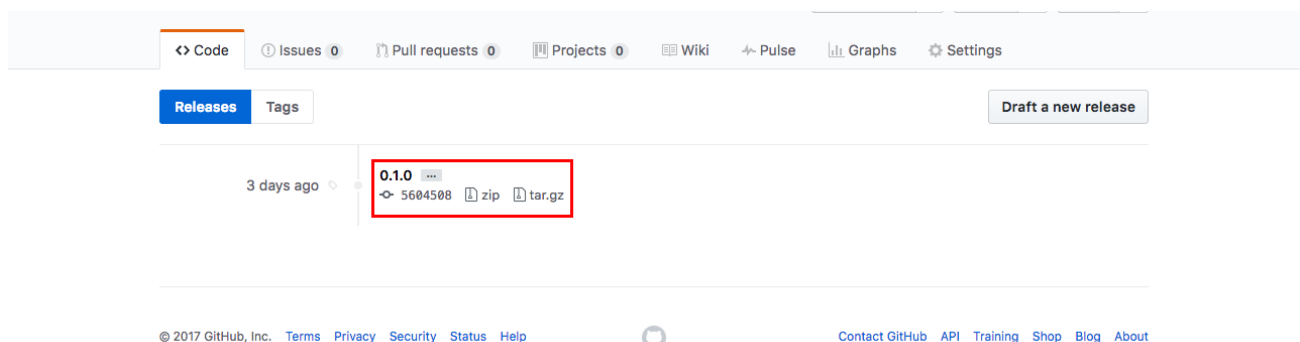
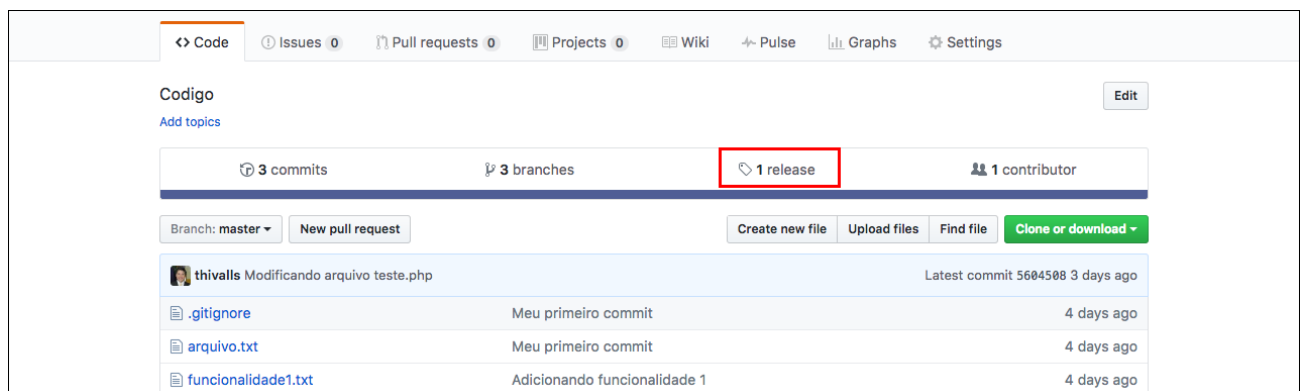
```
$ git tag 0.1.0  
$ git tag -l
```

O primeiro comando gera a tag e o segundo, lista todas as tags que existirem, localmente.

Subindo a tag para geração do release/versão:

```
$ git push origin master --tags
```

Desta forma, estamos subindo todas as tags, locais, para o repositório remoto.



Vocês podem ver na imagem, que existe a opção de baixar a versão no formato zip e tar.gz e isso vale para todas as demais versões que criarem.

Versionamento semântico

Versionamento semântico é uma forma, organizada, de versionar o software. Este assunto não está

relacionado com Git e Github, mas acabam interligados, de uma maneira ou de outra.

Existem 3 itens principais, como vocês podem ver no exemplo abaixo:

0.1.0

O primeiro item representa a versão principal, do sistema. Conhecido como **Major version**.

O segundo item, representa uma subversão, ou uma versão dentro da versão principal. Conhecido como **Minor version**. Este item, geralmente, representa as funcionalidades dentro da versão principal, ou seja, uma segunda funcionalidade criada, poderíamos criar a versão **0.2.0**.

Precisamos estar atentos quando chegar a hora de atualizar um sistema. Fiquem atentos a esta questão de versionamento, porque da versão 0.1.0 para 0.2.0 pode ser que tenhamos algumas quebras de funcionamento e que precisarão ser corrigidas.

O terceiro item é conhecido como **PATCH** e é utilizado para correções e melhorias, mas que não quebram a compatibilidade. Imaginem que foi encontrado um bug na versão 0.2.0 e foi corrigido. Atualizaremos para 0.2.1, desde que, esta correção ou melhoria, não quebre a compatibilidade. Se, por acaso, tenha esta quebra de compatibilidade, já seria o caso de criarmos mais uma minor version e criar a versão 0.3.0.

Caso venhamos a criar a versão 1.0.0, alterando a Major Version, não temos obrigação nenhuma de manter a compatibilidade com as demais versões.

A obrigatoriedade de manter a compatibilidade, vem da direita para esquerda, ou seja, qualquer alteração no patch, não pode afetar a compatibilidade da minor version e, do mesmo modo, qualquer alteração na minor version não pode afetar a compatibilidade da major version.

Um exemplo real, que teve esta mudança, é o Zend Framework, que tinha a versão 1 e veio com a versão 2, quebrando toda compatibilidade, por ser uma versão muito diferente da primeira e com muitas outras funcionalidades. Mas isso não causou nenhum estranhamento, porque eles mudaram a Major Version do projeto.

Quando forem criar o versionamento dos seus projetos, sempre levem este conceito em consideração, para quem for utilizar saber se pode ou não atualizar o projeto, sem que quebre, totalmente, o código.

Resolvendo conflitos

Infelizmente, quando trabalhamos com repositórios, estamos sujeitos a nos deparar com conflitos, mesmo trabalhando sozinhos, em um único repositório.

Para falarmos sobre os conflitos possíveis, utilizaremos nosso próprio exemplo. Estamos dentro do repositório **aulagit**. Verificamos se os branches estavam no mesmo commit, percebemos que os branches não estavam sincronizados. Rodamos o comando abaixo, para sincronizá-los.

```
$ git merge novobranh
```

Pronto, agora temos os dois branches apontando para o mesmo commit final, ou seja, estão sincronizados.

Alteraremos o arquivo **arquivo-novobranh.php**, colocando o seguinte conteúdo:

```
<?php
echo "novo branch -1";
?>
```

Depois de alterarem, comitem esta alteração.

```
$ git commit -a -m "Mudando o novo branch -1"
```

Em seguida, acessem o branch, novobranh e, dentro dele, editem o mesmo arquivo, colocando o seguinte conteúdo.

```
<?php
echo "novo branch -2";
?>
```

Depois comitaremos.

```
$ git commit -a -m "Mudando o novo branch -2"
```

Com estas alterações, podemos concluir que temos dois branches, com dois diferentes commits, no mesmo arquivo. Neste caso, não existe uma mágica que resolva nossa situação, porque, nem matematicamente falando, o Git saberia o que fazer nesta situação. Tentem imaginar como ele decidiria qual commit assumir como sendo, o correto ou incorreto, para descartar.

Para mostrar este conflito rodaremos o comando abaixo:

```
$ git merge master
```

Teremos o seguinte conflito sendo mostrado pelo Git

teremos o seguinte commit, sendo mostrado pelo Git:

```
Auto-merging arquivo-novobranch.php
CONFLICT (content): Merge conflict in arquivo-novobranch.php
Automatic merge failed; fix conflicts and then commit the result.
```

Isso ocorre porque o Git, realmente, não sabe qual dos commits tomar como sendo o correto. Neste caso, temos que abrir o arquivo e analisar o código, para ver qual seria o correto. O Git nos ajuda neste momento. Para visualizar esta ajuda, abram o arquivo para editar e verão que o Git criou uma marcação.

```
<<<<<< HEAD
echo "novo branch -2";
=====
echo "novo branch -1";
?>
>>>>>> master
```

Observem que ele nos mostrou o que temos no arquivo do branch atual, que ele marca como HEAD, e nos mostra o que temos no branch master. Agora, chegou a hora de tomarmos uma decisão, porque o Git não conseguirá tomar esta decisão, sozinho.

Antes de resolvermos o conflito, rodaremos o comando **git status**, para que consigam ver o que o Git nos informa.

```
On branch novobranch
Your branch is ahead of 'origin/novobranch' by 1 commit.
  (use "git push" to publish your local commits)

You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

   both modified:   arquivo-novobranch.php
```

Notem que o Git nos mostrou uma terceira categoria, que ainda não tínhamos visto. Seria **Unmerged paths**, com a informação de **both modified**. Isso quer dizer que, existem dois arquivos e ambos estão modificados, em branches diferentes.

Resolvendo o conflito antes de rodar o merge

Observem que, de acordo com o erro acima, temos que resolver o conflito e depois adicionar o arquivo para confirmar a solução do conflito. Em primeiro lugar, solucionaremos e, para fazer isso, basta apagarmos o que não queremos e deixarmos o correto.

```
<?php  
echo "novo branch -2";  
?>
```

Ficaremos com o conteúdo acima, como sendo o correto. Após, teremos que adicionar o arquivo, para confirmarmos a resolução do conflito, como informamos acima.

```
$ git add arquivo-novobranch.php
```

Depois de adicionarem, deem outro **git status**, para verem a mensagem.

```
On branch novobranch  
Your branch is ahead of 'origin/novobranch' by 1 commit.  
  (use "git push" to publish your local commits)  
All conflicts fixed but you are still merging.  
  (use "git commit" to conclude merge)
```

Temos a mensagem de que todos os conflitos foram resolvidos, mas que precisamos rodar o commit, para que a ação de merge seja concluída.

```
$ git commit -m "Resolvendo conflito"
```

Temos a nossa ação completa e sem conflito algum. Isso gera um commit a mais em nosso log, para informar que está tudo correto e sem conflitos.

Esta situação não é muito comum, quando se trabalha sozinho em um projeto, mas quando se trabalha com vários branches e em equipe, pode ser que vocês se deparem com esta situação.

Nosso objetivo foi ensinar a lidar com esta situação. Quando se trabalha em equipe, provavelmente, vocês podem ter que decidir o conflito, juntamente com outro desenvolvedor, que alterou o mesmo arquivo que você.

Git bare

O Git possui um tipo de repositório que é utilizado, apenas, para servir, ou seja, ele fica preparado, somente, para receber os commits e os pushes que vocês executarem e, também, libera os pulls, quando vocês solicitarem.

Vamos mostrar na prática.

Saiam da pasta atual, do nosso exemplo, e criem uma nova pasta chamada, **aulagit.git**. Depois, acessem esta pasta **\$ cd aulagit.git/**.

Agora, estando dentro da pasta criada, rodem o seguinte comando:


```
$ git init --bare
```

O único objetivo deste repositório, é servir. Não trabalharemos dentro dele, pois ele cria a mesma estrutura de pastas que o comando `git init`, mas não de forma oculta.

Para visualizarem, rodem `$ ls`, dentro da pasta e vocês poderão visualizar a estrutura das pastas.

Voltaremos ao repositório `aulagit`, que faz parte dos nossos exemplos, e linkaremos este repositório, ao nosso repositório **bare**, rodando o seguinte comando:

```
$ git remote add local ssh://localhost/Users/mac/www/aulagit.git
```

Observem que, como é um repositório remoto, mas está localmente, não o chamamos de **origin**, o nomeamos como **local** e passamos o caminho da pasta da nossa máquina, de forma remota.

Depois de linkar, acessem o branch master:

```
$ git checkout master
```

Em seguida, falta darem um push, no repositório local, para verem se está funcionando.

```
$ git push local master
```

Teremos o seguinte resultado:

```
Counting objects: 20, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (20/20), 1.63 KiB | 0 bytes/s, done.
Total 20 (delta 7), reused 0 (delta 0)
To file:///Users/mac/www/aulagit.git/
 * [new branch]      master -> master
```

Pronto! Já temos um repositório local, trabalhando como se fosse um repositório remoto.

Git hook

Agora que vocês já sabem que existe um tipo de repositório que só tem a finalidade de servir e para trabalhar como repositório central. Mostraremos como podemos fazer o deploy, automatizado.

Suponham que estejam no seu repositório Git, trabalhando, normalmente, em suas máquinas, e toda vez que vocês realizarem um push, este repositório central, copiará estes arquivos para uma outra pasta, onde estará hospedado o site de vocês, por exemplo. Isso quer dizer que, vocês trabalharão, localmente e, quando derem um push no site, será atualizado junto com sua versão local. Esta funcionalidade é, realmente, muito interessante, o que faz vocês ganharem tempo.

Para fazerem esta configuração, acessem o repositório `aulagit.git` do tipo **bare**, depois acessem a pasta **hooks** e criem um arquivo chamado **post-receive**. Os hooks são ganchos que, quando determinada ação acontece, dispara outras ações, pré-configuradas. Este arquivo, `post-receive`, nada mais é do que um outro gancho, que será executado, toda vez que dispararmos um push.

Dentro deste arquivo adicionem o seguinte código:

```
#!/bin/sh
GIT_WORK_TREE=/Users/wesley/gitCode/meusite.com.br git checkout -f
```

A primeira linha indica que estamos trabalhando com comando do shell e a segunda, é a configuração que indica para onde os arquivos serão copiados, quando o push for executado. Observe que, passamos o parâmetro **-f** para que isso aconteça.

Depois disso, deem permissão de execução para este arquivo, rodando o comando **\$ chmod +x post-receive**. Pronto, já temos o nosso hook configurado.

Devemos criar a pasta **meusite.com.br**, que configuramos acima, para que no momento do push, a mágica aconteça e os arquivos sejam copiados, automaticamente, para este endereço. Criaremos, dentro da pasta **gitCode**.

```
mkdir meusite.com.br
```

Caso queiram confirmar, abram esta pasta, antes de dar qualquer push, para vocês verem que ela está vazia e, depois do push ela terá os arquivos copiados.

Acessem o repositório `aulagit` e alterem o arquivo **arquivo.txt**, adicionem e comitem este arquivo e realizem o push. Notem que, temos duas opções de push:

1. `git push origin master`
2. `git push local master`

O primeiro, enviamos para o repositório remoto do Github e o segundo realiza o push em nosso repositório remoto local e, conseqüentemente, em nossa pasta `meusite.com.br`, que acabamos de configurar. Depois que o repositório bare receber o push, ele executará o `post-receive` e enviará os arquivos para a pasta que configuramos.

O Git pedirá a senha do usuário da máquina para concluir a operação.

Depois que os dois push forem realizados, vocês podem acessar, novamente, a pasta `meusite.com.br` e verão que os arquivos já estarão lá dentro, assim como esperado. Desta forma, resolvemos nosso problema e ganhamos muito tempo de deploy, onde precisaríamos subir por FTP ou acessando o servidor de alguma outra forma.

Suponham que o repositório do tipo bare, esteja no servidor de hospedagem do site e ele utilize o hook post-receive para direcionar os arquivos para a pasta public_html, por exemplo. É desta forma, que conseguimos fazer o deploy online, utilizando o Git.

Esta, não é a melhor forma do mundo de configurar o deploy, porque se vocês alterarem arquivos de senha, por exemplo, vocês terão problemas, mas de forma geral, é uma opção muito interessante e que, em muitos casos, pode facilitar.

Estamos fazendo estas configurações, utilizando o ambiente Linux/Unix. Procurem trabalhar com o mesmo ambiente para que não tenham nenhum problema.

Conclusão

Esperamos ter cumprido com o objetivo do conteúdo, que era dar uma boa base de trabalho com Git e Github.

Não deixem de estudar e praticar, inclusive com projetos reais, para que o conteúdo esteja cada vez mais fixado e o mais natural possível, para vocês.

Até o próximo tema.