***Curso de Git e Github***

Criando um repositório:

Mkdir mais o nome a sua escolha.

Depois abra a pasta cd + o nome da pasta.

Dentro da pasta use o comando git init. Para inicializar o repositório.

Para ver as pastas ocultas use o comando ls -la.

Para ver o conteúdo da pasta .git:

cd .git

para listar o conteúdo da pasta ou repositório.

ls.

Para apagar o repositório e so apagar o repositório e pronto.

Para criar um arquivo:

Usa-se o comando touch mais o nome do arquivo e sua extensão.

Git status mostra o status dos arquivos que estão no repositório que ainda não está sobre o controle de versão do git. Por isso eles irão aparecer em vermelho.

Para adicionar um arquivo para o controle de arquivos usa-se o seguinte comando:

Git add e o nome do arquivo junto com a extensão.

***Tipos de git log***

Git log -p mostra todas as mudanças feitas nos arquivos.

Git log -p -2 mostra as 2 ultimas mudanças feitas nos arquivos.

Git log --pretty=oneline, seve para trazer todos os commits em uma line só.

Git log --pretty=format:”%h - %an, %ar : %s”, serve para trazer um relatório de quando foram feitos commits e quanto tempo se passou.

Git log --since=2.days, serve para trazer os commits de dois dias atrás.

Git log --stat , mostra todas as statisticas para todos os commits.

Git log --since=2.weeks serve para duas semanas atrás.

Para usar estes comandos deve estar na pasta onde se encontra o arquivo.

Rever o vídeo de ignorando um arquivo no .gitignore

***Brincando com o fluxo de commits***

Git add . Serve para adicionar todos os documentos para serem commitados.

Git reset HEAD . Serve para desfazer a adicionamento de todos os arquivos.

Git reset HEAD nome do arquivo Serve para retirar somente um arquivo por vez.

***Voltando versões***

Para voltar á uma versão, use o comando git chekout mais o numero do commit em que quer voltar.

Agora para ir voltando um a um ou dois a dois enfim use git reset HEAD~1 ou 2 ou 3 enfim.

Existe agora dois tipos de git reset HEAD~1 –soft remove uma quantidade de commits desejado sem remover as alterações.

Já o outro comando git reset HEAD~1 –hard ele retira definitivamente um commit anterior juntos com os arquivos.

***Criando primeiro branch***

Para criar um branch use o comando git checkout -b mais o nome do branch que vc deseja criar.

E recomendado fazer um branch para cada funcionalidade do sistema que estiver sendo desenvolvido, depois colado no branch mestre com merge ou rebase.

***Fazendo um merge***

Merge significa mesclar branch e ao ser executado gera um novo commit para o branch máster referenciando o branch a ser colado no branch máster.

Git merge nome do branch a ser colado no principal. Isto não é recomendado.

Já o rebase evita fazer outro commit no máster, mas porem isto e feito localmente em sua maquina.

Ele também organiza os commits na ordem que eles foram criados no mesmo branch que no caso e no máster.

Git reset head~1 –hard serve para voltar o branch principal a uma versão anterior a inserção do merge.

GitHub

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.

Falaremos mais sobre push em outros módulos, para explicar melhor este comando.

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 reposotório 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. No próximo módulo, daremos continuidade.

Criem seus repositórios, para praticarem.

Primeiro push

Neste módulo vocês aprenderão a realizar o primeiro push.

Antes de realizarmos o procedimento e falarmos dos comandos, explicaremos o que é o 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.

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)

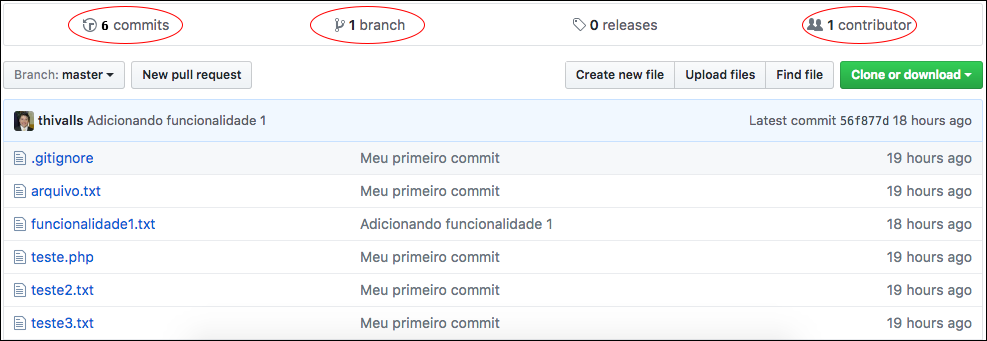
remote: Resolving deltas: 100% (1/1), done.

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.



Observem que todas as informações de commits e branchs 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.

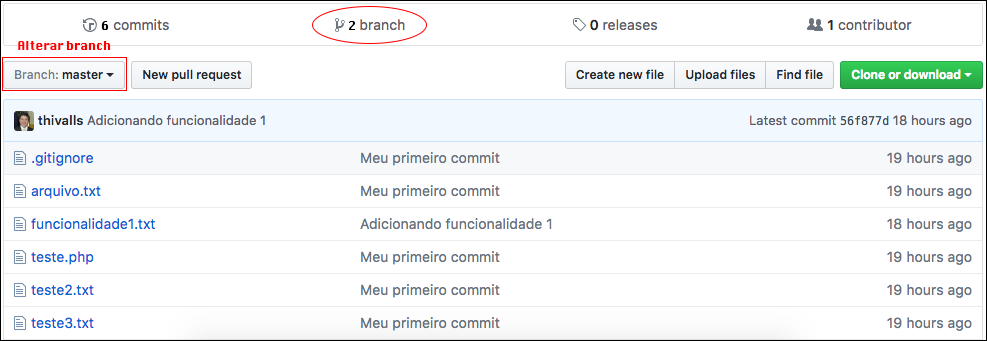
Total 0 (delta 0), reused 0 (delta 0)

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 branchs criados e onde vocês podem alterar o branch online.

Vocês já sabem como enviar seus branchs para o repositório remoto. Daremos continuidade ao conteúdo.

Fazendo clone

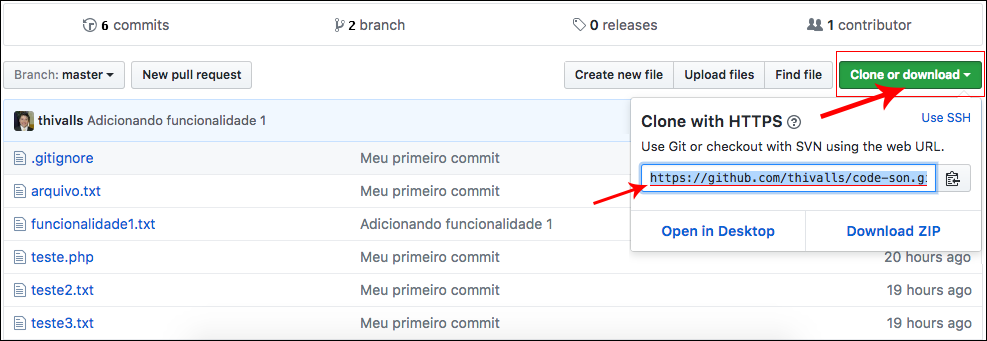
Neste módulo, mostraremos como fazer um clone de um repositório e para que ele serve.

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 branchs que estiverem, localmente. Mas, vocês podem ver os branchs remotos, também. Basta acrescentarem o parâmetro -a no comando. Neste caso, vocês terão os branchs locais e remotos.

$ git branch -a

\* master

remotes/origin/HEAD -> origin/master

remotes/origin/funcionalidade1

remotes/origin/master

Observem que ele mostra os branchs 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 branchs locais e mais dois branchs 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 branchs 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.

Agora vocês já conseguem clonar, completamente, qualquer repositório do Github.

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 módulo 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.

$ 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 branchs 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 branchs.

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

Pratiquem bastante, para que possam dominar estas **idas e vindas** de repositórios locais para repositórios remotos, e vice versa.

Github - Dicas gerais

Apresentaremos algumas dicas, gerais, do Github para que vocês possam trabalhar da melhor forma, possível, com a ferramenta.

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.

# Versionamento semântico

Já que falamos de versões, no módulo anterior, precisamos passar para vocês, um conceito de 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.

Este módulo serviu para que pudéssemos passar este conceito, que é simples, porém muito importante. Agora, vocês sabem que os desenvolvedores não saem criando versões, sem nenhum método específico e sem responsabilidade. Vocês tem que saber que muitos, podem estar utilizando o seu código. Quando disponibilizado para muitos, a responsabilidade sempre aumenta.

Resolvendo conflitos

Este módulo foi criado para ajudá-los com as dicas mais variadas, possíveis, do Git. Se estivermos falando de várias coisas, ao mesmo tempo, não fiquem preocupados, este módulo é misto, para que possamos abordar alguns pontos que vocês poderão se deparar, no dia a dia.

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 branchs estavam no mesmo commit, percebemos que os branchs não estavam sincronizados. Rodamos o comando abaixo, para sincronizá-los.

$ git merge novobranch

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

Alteraremos o arquivo **arquivo-novobranch.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, novobranch 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 branchs, 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.

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 branchs 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 branchs e em equipe, pode ser que vocês se deparem com esta situação. Nosso objetivo, com este módulo, 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.

Simulem mais um conflito como este, no projeto, para que resolvam sozinhos e pratiquem esta resolução de conflitos do Git.

Git bare

Ensinaremos a trabalharem de forma remota, mas localmente, criando um repositório remoto, em nossa máquina.

Quando vocês rodam um git init, estão criando um novo repositório git, em que conseguem trabalhar remotamente e localmente.

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 pushs 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, que criamos no módulo anterior, 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. Obervem 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, consequentemente, em nossa pasta meusite.com.br, que acabamos de configurar. Depois que o repositório bare receber o push, ele executará o push 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

Gostaríamos de agradecer o interesse de todos.

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.