**Git e Github**

O git é um sistema que permite que várias pessoas façam alterações num código e envie para um armazenamento remoto, não permitindo que outras pessoas que também tenham feito alterações possam mandar essas novas modificações antes de baixar as que a outra pessoa fez. Isso facilita a comunicação entre os desenvolvedores já que não tem a necessidade de colocar o código num pendrive e levar até os outros para que eles possam ver as modificações feitas e iniciar a partir daí. Todos podem fazer alterações ao mesmo tempo e todos ficam sabendo na hora o que foi alterado.

Além do Git existem outras plataformas que fazem isso, como a CVS, SVN. Mercurial...

As branchs servem para separar o que cada indivíduo está trabalhando evitando que hajam confusões no código compartilhado entre a equipe. Por exemplo, a pessoa que está trabalhando no cabeçalho da página web cria uma branch e irá mexer somente nessa parte, a pessoa que do main a mesma coisa, mas para o main, e a do footer, igual. Esses são exemplos de divisões do código compartilhado em branchs evitando que o código acabe quebrando em determinada parte por várias pessoas estarem mexendo na mesma brunch, que é a master. Ao final da implementação da feature que cada um estava fazendo tiver sido finalizada, faz-se o merge dessa brunch com o main/master/branch principal do código, implementando de fato o que foi desenvolvido.

Para saber mais do git status:

Ao executar o comando git status, recebemos algumas informações que talvez não estejam tão claras, principalmente quando nos deparamos com termos como HEAD, working tree, index, etc.

Apenas para esclarecer um pouco, visto que entenderemos melhor o funcionamento do Git durante o treinamento, seguem algumas definições interessantes:

* HEAD: Estado atual do nosso código, ou seja, onde o Git os colocou
* Working tree: Local onde os arquivos realmente estão sendo armazenados e editados
* index: Local onde o Git armazena o que será *commitado*, ou seja, o local entre a *working tree* e o repositório Git em si.

Além disso, os possíveis estados dos nossos arquivos são explicados com detalhes neste link: <<https://git-scm.com/book/pt-br/v2/Fundamentos-de-Git-Gravando-Altera%C3%A7%C3%B5es-em-Seu-Reposit%C3%B3rio>.

Para saber mais de quando commitar:

Devemos gerar um *commit* sempre que a nossa base de código está em um estado do qual gostaríamos de nos lembrar. Nunca devemos ter *commits* de códigos que não funcionam, mas também não é interessante deixar para *commitar* apenas no final de uma *feature*.

Essa pode ser uma discussão sem fim e cada empresa ou equipe pode seguir uma estratégia diferente. Estude sobre o assunto, entenda o que faz mais sentido para você e sua equipe e seja feliz! :-D

Para saber mais de ramificações:

*Branches* ("ramos") são utilizados para desenvolver funcionalidades isoladas umas das outras. A *branch* master é a *branch* "padrão" quando você cria um repositório.

É interessante separar o desenvolvimento de funcionalidades em *branches* diferentes, para que as mudanças no código para uma não influencie no funcionamento de outra.

Para saber mais de Conflitos com rebase:

Vimos como é simples resolver conflitos identificados pelo Git ao tentar realizar o merge.

Agora, gere um conflito e, ao invés de utilizar o merge, utilize o rebase para atualizar o master:

* Vá para a *branch* titulo
* *Commite* algo
* Vá para a *branch* master, *commite* uma alteração na mesma linha
* Execute git rebase titulo

Veja a saída do Git e utilize as informações que ela te der para, após corrigir o conflito, continuar o rebase.

**Comandos no bash do git ou no terminal:**

- *git init*: inicia o repositório na pasta em que está.

--bare: indica que é um repositório puro e spó contem as alterações dos arquivos e não uma cópia física de cada arquivo que está sendo alterado. Utilizado para criar o servidor do projeto que estamos trabalhando com um time.

\* O ideal é criarmos uma pasta separada de “servidor” para fazer isso

- *git status*: exibe algumas informações da pasta que estamos, como os arquivos que não estão sendo rastreados por mudanças, diz quantos commits esse repositório já tem...

- *git add <file>*: adiciona o arquivo no commit para começar a ser rastreado.

- *git config --local user.name "Seu nome aqui"*

- *git config --local user.email “*[*seu@email.aqui*](mailto:seu@email.aqui)*”*: Esses dois comandos servem para identificar quem que está fazendo as alterações nessa pasta no computador, deixando registrado sempre. Sempre fazer quando instala o git. Deve ser feito na pasta em que iniciamos o git.

Se alterarmos o --local por --global ele fará as alterações/definições de nome e e-mail de usuário para todos os projetos.

Em --local ele muda/define apenas para aquele repositório específico que estamos trabalhando.

Utilizando o comando sem o --local e os dados em “ “, ele apenas exibe qual o nome de usuário ou o e-mail que está configurado.

- *git rm –cached <file>*: remove o arquivo que adicionamos com git add e está aguardando o commit para ser rastreado.

- *git commit -m “mensagem”*: faz o commit das novas alterações que foram feitas no repositório. O -m é para podermos digitar a mensagem a seguir. Pode ser algo simples, não devemos fazer uma documentação toda nessa mensagem. O git monitora linha a linha de tudo o que estamos fazendo o commit por conta disso, ao executar o comando ele diz até quantas linhas o arquivo que estamos commitando tem.

Temos que fazer todo o processo de adicionar e commitar com a mensagem sempre que fizermos uma modificação. Na dúvida, utilize o git status para verificar como estão as coisas.

- *git log*: exibe o histórico de commits que fizemos nesse repositório. Ele mostra o head em primeiro lugar, sendo onde estamos atualmente na linha do tempo dos commits. Ele mostra o autor do commit, data e hora e a descrição dele.

--oneline: na frente ele exibirá cada commit de forma resumida em uma única linha.

-p: mostra todas as alterações detalhadamente, inclusive o que foi acrescentado e/ou tirado do código.

Existe uma infinidade de formas que podemos visualizar o histórico: <http://devhints.io/git-log>

--graph: exibe linhas antes das informações dos logs, mostrando em qual linha de desenvolvimento/branch cada commit foi feito.

- *.gitignore*: arquivo que criamos e tudo o que colocamos dentro são os arquivos que o git não irá monitorar, ou seja, criamos esse arquivo e escrevemos nele o nome das pastas/ e arquivos que desejamos que o git não veja as mudanças e afins que fizermos neles. A partir do momento que criamos esse arquivo só precisamos commitar ele e tudo o que estiver dentro dele será ignorado.

- *git remote*: exibe a lista de repositórios remotos que temos.

-v : mostra os repositórios remotos que temos e seus caminhos.

- *git remote add <nome> <caminho>*: adiciona a pasta em que estamos com o nome que colocamos ao repositório remoto que criamos com o git init –bare no caminho que colocarmos. Esse caminho não precisa ser um repositório remoto que nós criamos, mas também links de sites e afins.

Esse servidor remoto serve para quando temos 2 usuários na mesma máquina e que farão alterações no código em momentos diferente, deixando assim tudo bem documentado e organizado.

- *git remote add origin* [*https://linkDoRepositórioGitHub.com.br*](https://linkDoRepositórioGitHub.com.br): faz com que conectemos o repositório que estamos trabalhando a um servidor online do github, desse modo, 2 ou mais pessoas podem clonar esse repositório e fazer as modificações, pulls e pushs de suas próprias máquinas em suas casas.

- *git remote rename*: renomeia o nome do computador que está usando.

- *git clone <caminho> <nome pasta>*: clona um repositório do endereço passado para dentro da pasta com o nome que acabamos de colocar.

- *git push <local> <master>*: envia as modificações que fizemos para o servidor.

Git push origin master: após conectar com o github, isso fará com que faça o upload de todos os arquivos desse repositório para o github.

- *git pull <local> <master>*: pega as alterações feitas do servidor e trás para o local.

- *git branch <nome>*: cria uma nova ramificação/branch no repositório. Se não colocar o nome apenas será listado as branchs existentes. Se tiver mais de uma branch, a que você estiver utilizando ficará com um ‘\*’ e em coloração verde.

- *git checkout ‘branch’*: altera a branch que está conectado para trabalhar nessa nova linha de desenvolvimento.

-b: coloca antes da ‘branch’ desse modo ele já cria e muda para a branch nova automático. É um atalho bem útil.

- -- <file>: desfaz todas as alterações de um arquivo antes de fazer o commit.

- Se colocarmos o hash de algum commit voltamos para ele, não importa qual é a ordem na linha do tempo. Tudo o que for feito aqui será descartado caso voltemos para a branch master, já que ao fazer isso somos desanexados da linha de desenvolvimento. Se quisermos manter o que for feito nessa linha, precisamos criar uma nova branch, fazer as alterações e posteriormente anexar a master. Como se estivéssemos criando uma nova branch de desenvolvimento padrão mesmo.

- *git merge <branch>*: ao estar na branch master e utilizar esse comando colocando o nome de outra branch, fazemos o merge, a união de ambas as branchs, trazendo todas as modificações de ambas as branchs para uma única – a master.

- *git rebase <branch>*: ao estar na branch master e utilizar esse comando colocando o nome de outra branch, fazemos um merge de tudo, mas sem gerar um novo commit, apenas trazendo todos os commits da master e da outra branch para uma única linha, evitando poluição de log com vários merges.

- *git restore –stage <file>*: Após adicionar o arquivo e antes de fazer o commit, podemos utilizar esse comando para retirar ele do adicionamento, caso necessário fazer um um checkout --.

- *git revert <hash>*: Desfaz o commit que fizemos acidentalmente, removendo todas as alterações.

- *git stash*: salva uma mudança temporariamente, sem gerar nenhum tipo de log ou informação sobre isso. Muito utilizado caso esteja fazendo alguma alteração e surgiu uma tarefa mais urgente. Para não perder e garantir que o que foi feito fique salvo, utilizamos esse comando.

Colocar um list na frente exibe tudo o que está salvo no stash.

Colocar um apply na frente e o número da stash que deseja, ele irá trazer aquelas alterações de volta.

Colocar o drop remove esse salvamento do stash.

Colocar o pop faz com que remova do stash e ainda traga essas alterações. É a junção dos dois acima.

- *git diff <hash1>..<hash2>*: mostra a diferença que teve entre as duas versões dos hashs. Os ‘..’ indicam do hash1 ‘até’ o hash 2.

Caso já tenhamos começado uma alteração no código, mas ainda não adicionamos ela para fazer o commit, digitar somente git diff exibe o que foi feito no código desde o último commit.

- *git tag -a <name> -m “mensagem”*: Cria uma tag, um espécie de commit que não pode ser modificado. Basicamente um marco como quando lançamos as versões de aplicações. Também utilizamos para poder navegar entre as versões, caso já estejamos na v3.1.2 e queiramos voltar a versão BETA v0.1.0 no seu exato estado de lançamento, basta voltar para essa tag.

Para visualizar todas as tags que já criamos para poder retornar a elas caso necessário, basta digitar git tag e serão exibidas todas elas.

A mensagem não é obrigatória.

Podemos fazer o push dessa versão para o servidor também.

Ao fazer o push dessa versão/tag para o github, ela se torna uma release, onde qualquer pessoa pode baixar essa versão do nosso projeto e rodar na máquina dela.