GIT & GITHUB

Do Básico ao Avançado

Domine Git e GitHub com este material completo. Com ele você aprenderá a gerenciar fluxos de trabalho de maneira simples e dinâmica.

Git Stash

Casualmente haverá momentos em que você vai trabalhar em cima de uma nova funcionalidade, ou em alguma alteração no seu projeto.

E acontecer de você parar o seu desenvolvimento para resolver outra coisa mais urgente, como a correção de um **"bug"**, ou uma **funcionalidade que precisa ser implementada para ontem**.

Considerando que a funcionalidade que você está trabalhando está com apenas 60% de conclusão, e que você precisa parar o desenvolvimento dela para resolver um bug que te pediram.

Qual o procedimento que você poderia fazer para salvar suas alterações de modo a deixá-las de lado, ou seja, fazer uma cópia daquilo que você fez, para que posteriormente você consiga voltar para elas?

Bem, existam duas formas de você fazer isso, a primeira é por meio de branchs, e a segunda você pode fazer isso com o comando Git Stash.

Fazendo uma cópia com Branchs

Para testarmos esse procedimento por meio das branchs, vamos iniciar criando uma pasta chamada de **‘stash’**, e dentro dela, criaremos um arquivo de texto chamado de **‘BoasVindas.txt’**, com o seguinte conteúdo:

1	Olá Aluno, seja muito bem-vindo aos treinamentos intensivos de Marketing Digital da Olyng.
2	Para acessar as aulas, faça o login em:
3	https://academy.olyng.com/entrar-na-academy

Após isso, vamos dar sequência aos procedimentos básicos do git, abrindo o terminal do git **(“git bash here”)** na pasta do projeto, seguido dos comandos:

```
git init
git add .
git commit -m ‘Configuração Inicial’
```

(Vamos considerar que você está trabalhando diretamente na branch Master/Main nesse projeto, ok?)

Após a configuração inicial, nosso gestor, nos pediu para criar um novo documento de texto, onde vai conter as **regras da comunidade**.

Para isso, precisamos criar um arquivo de texto com o nome **‘RegrasDaComunidade.txt’** com o seguinte conteúdo:

1	Regra Nº 1 – Não tratar mal qualquer membro da comunidade, incluindo os alunos.
2	Regras Nº 2 – Não devo compartilhar meus dados pessoais de acesso com mais ninguém.

Como nosso gestor nos disse anteriormente, a nossa comunidade é composta por 10 regras, mas no momento ele só conseguiu nos passar duas delas (N° 1 e N° 2), pois as outras ainda estavam sendo criadas pelos executivos da empresa.

Nesse meio tempo, nosso gestor entrou em contato conosco, nós pedindo urgentemente para corrigir o link contido no documento de boas-vindas.

Segundo ele, o link que foi passado estava errado, e que muitos alunos não estavam conseguindo entrar na plataforma.

Nesse momento, nós vamos precisar para o fluxo de trabalho do arquivo **'RegrasDaComunidade.txt'** para podermos voltar nele mais tarde.

Para isso, salve esse arquivo, e adicione-o na **STAGE**:

```
git add .
```

Em seguida, vamos criar uma nova branch, chamada de **'wip'**, que é sigla para **"Work in Progress"**, ou na sua tradução **"trabalho em progresso"**.

```
git checkout -b wip
```

Em seguida vamos fazer o commit das alterações que nós fizemos dentro dessa branch:

```
git commit -m "wip"
```

Dessa forma, você não vai perder as alterações que você fez no arquivo **'RegrasDaComunidade.txt'**.

Agora, chegou a hora de voltarmos para a nossa branch **master/main**:

```
git checkout master
```

Note que nesse momento, o arquivo **'RegrasDaComunidade.txt'** não existe mais na branch principal.

Dentro da **master/main**, vamos precisar abrir o arquivo de boas-vindas, e fazer uma alteração no link, veja como ficou:

1	Olá Aluno, seja muito bem-vindo aos treinamentos intensivos de Marketing Digital da Olyng.
2	Para acessar as aulas, faça o login em:
3	https://academy.olyng.com/login

Salve esse arquivo, e siga o fluxo normalmente para adicionar ao histórico:

```
git commit -am "Alteração no arquivo de boas-vindas"
```

Com o problema do link já resolvido, podemos voltar para a branch que criamos:

```
git checkout wip
```

Em seguida precisamos fazer um Reset do commit com a descrição de **"wip"** que realizamos nessa branch, pois tecnicamente ele não deveria existir:

Nesse caso, vamos executar um reset do tipo soft, com uma versão anterior a que a **HEAD** está apontando:

```
git reset head^
```

Lembre-se que o **^** faz o reset para uma versão anterior.

Com isso voltamos ao estado que apenas criamos o arquivo **'RegrasDaComunidade.txt'**, ao mesmo tempo que não temos ele na Stage e que também não está comitado.

Após isso, podemos voltar para a branch principal:

```
git checkout master
```

Note que o arquivo **'RegrasDaComunidade.txt'**, voltou a aparecer na branch principal.

Agora basta que você adicione na **STAGE**:

```
git add .
```

E continue esperando o executivo terminar de criar as regras para fazer o commit na branch principal.

O que é o Git Stash?

O comando **Git Stash**, é um comando responsável por arquivar alterações que você está fazendo no seu repositório, para que você possa trabalhar em alguma outra coisa no projeto, depois voltar e continuar trabalhando na alteração anterior.

Usamos este comando, pois muitas vezes somos obrigados a corrigir algum bug, ou implementar algo urgentemente, ao mesmo tempo que estamos trabalhando em uma determinada alteração, que não podemos fazer o commit, pois ela ainda não se encontra finalizada.

O **stashing** é útil quando você precisa alternar com rapidez o contexto e trabalhar em outra coisa, mas está no meio da alteração de código e não está pronto para fazer commit.

```
git stash
```

Com esse comando salvamos todas as nossas alterações que fizemos no projeto. Tanto aquelas que estão na Stage, como aquelas que ainda não foram adicionadas a Stage.

É importante ressaltar que o comando **git stash**, salva as alterações em uma pilha chamada **WIP**, para uso posterior, e as reverte da cópia de trabalho.

Tanto que após a execução deste comando, se executarmos o **git status**, receberemos a mensagem:

```
MINGW64:/c/Users/SeuUser/Desktop

SeuUser@DESKTOP-UHDBDBV MINGW64 ~/Desktop/Meu Projeto (master)
$ git status
On branch main
nothing to commit, working tree clean
```

Indicando que a branch atual não conta com nenhuma modificação/alteração aparente, até porque tudo que fizemos foi arquivado pelo stash.

Nesse ponto, você está livre para fazer alterações, criar novos commits, alternar ramificações e executar quaisquer outras operações do Git.

É importante ressaltar que o stash é feito localmente no seu repositório Git, e que eles não são transferidos para o servidor quando você os envia por push.

Recuperando Cópias

Já fez as alterações que o gestor te pediu? Já arrumou aquele bug urgente que passou despercebido no projeto? Então chegou a hora de voltarmos ao que estávamos fazendo antes!

Atualmente o comando Stash conta com dois comandos responsáveis por reaplicar as alterações presentes na stash, são elas:

```
git stash pop
```

Com este comando nos estamos recuperando as alterações que estão no stash, ao mesmo tempo que estamos deletando a cópia dessas mesmas alterações que estão no stash.

```
git stash apply
```

Com este comando nos estamos recuperando as alterações que estão no stash, sem deletar a cópia dessas mesmas alterações que estão no stash.

Enquanto o **stash apply** está para **Control + C**, e **Control + V** (copiar e colar). O **stash pop** está para **Control + X**, **Control + V** (recortar e colar).

É importante ressaltar que por padrão, o comando git stash realiza uma cópia somente nas:

+ Alterações que já foram adicionadas a Stage Area.

+ Alterações feitas em arquivos que estão sendo rastreados pelo Git.

Entretanto, o stashing por padrão, não se aplica a:

+ Arquivos que não estão sendo rastreados, e que também não estão na Stage Area.

+ Arquivos que foram ignorados pelo .gitignore.

```
git stash -u
```

Com este comando dizemos ao git para realizar o stashing de arquivos que não foram rastreados. (pode trocar a flag **-u** para **--include-untracked**)

```
git stash -a
```

Com este comando dizemos ao git para adicionar ao stash também os arquivos ignorados. (pode trocar a flag **-a para --all**)

Múltiplos Stashes

Você não está limitado a um único Stash, sabia disso?

Sempre quando executamos o comando git stash, uma nova pilha de stashing é criada em **WIP**.

```
git stash list
```

Com este comando podemos visualizar todos os stashs que estão armazenados no projeto.

Tanto é que se usarmos o comando **git stash list**, veremos uma lista das cópias armazenadas:

```
MINGW64:/c/Users/SeuUser/Desktop

SeuUser@DESKTOP-UHDBDBV MINGW64 ~/Desktop/Meu Projeto (master)
$ git stash list
stash@{0}: WIP on main: 5002d47 ....
stash@{1}: WIP on main: 5002d47 ....
stash@{2}: WIP on main: 5002d47 ....
```

O único problema é que depois de algum tempo, pode ser difícil nos lembrarmos do que se trata cada um dos stashs.

Seria ótimo se pudéssemos adicionar uma descrição a cada stash que salvamos, não é verdade?

```
git stash save “mensagem”
```

Com este comando adicionamos os arquivos ao stash, atribuíndo uma mensagem a ele.

Com o comando acima, nós podemos, por exemplo, atribuir a mensagem **“funcionalidade da homepage”** àquela cópia.

E quando executarmos o **git stash list**, por exemplo, seria mais fácil identificar do que se trata aquele stash:

```
MINGW64:/c/Users/SeuUser/Desktop

SeuUser@DESKTOP-UHDBDBV MINGW64 ~/Desktop/Meu Projeto (master)
$ git stash list
stash@{0}: On main: Funcionalidade da HomePage
stash@{1}: WIP on main: 5002d47 ....
stash@{2}: WIP on main: 5002d47 ....
```

Observação: Todos os stashs que contam com mensagens, ficam salvos diretamente na main, nesse caso, eles não estão relacionados com a WIP.

Para recuperarmos um determinado stash, podemos usar o comando abaixo:

```
git stash pop stash@{2}
```

No caso do comando acima, estamos recuperando o stash de número 2, ou seja, a segunda cópia que fizemos.

No lugar do 2, podemos colocar o número do stash que queremos selecionar..

Lembrando que no lugar de **pop**, podemos usar também o **apply**, caso preferir.

Quando usamos os comandos: pop, apply ou drop sem informar o número do stash, o git entende que queremos selecionar o stash que está no topo, que é sempre o primeiro.

É como se fizéssemos **git stash pop stash@{1}**.

--Keep-Index

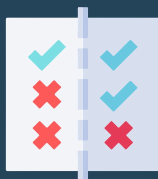
O comando stash conta com uma flag chamada **--keep-index**, que pode ser usada da seguinte forma:

```
git stash --keep-index
```

Com ela, todas as alterações já adicionadas ao índice permanecem intactas, ou seja, mantém as alterações que já estão na área de STAGE, e remove do stash todas as alterações marcadas como unTracked.

Em outras palavras, todos os arquivos que você adicionou ao git também permanecerão fora do stash quando você o criar.

Considerando que normalmente os arquivos recém-adicionados seriam removidos após o stashing.



Comparando Stash

Para fazer a comparação entre os stashs, você pode usar o comando:

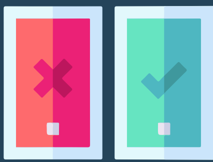
```
git stash show
```

```
MINGW64:/c/Users/SeuUser/Desktop

SeuUser@DESKTOP-UHDBDBV MINGW64 ~/Desktop/Meu Projeto (master)
$ git stash show
 index.html | 1 +
 style.css | 3 +++
 2 files changed, 4 insertions(+)
```

Ou passar a opção **-p** (ou **--patch**) para visualizar a comparação completa do stash:

```
git stash show -p
```



Stashs Parciais

É possível escolher fazer o stashing em um único arquivo, uma coleção de arquivos, ou mudanças individuais de dentro de arquivos.

Se você passar a opção **-p** (ou **--patch**) ao git stash, ele vai percorrer cada "fragmento" alterado na cópia de trabalho, e perguntar se você quer fazer o stashing:

```
git stash -p
```

Aqui temos acesso a diversos comandos que podem nos ajudar, são eles:

/ = realiza a pesquisa de um fragmento por regex.

? = pede ajuda ao git.

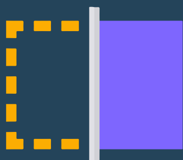
n = não faz stash neste fragmento.

q = sai do processo de stashing (o stash vai ser feito em quaisquer fragmentos que já foram selecionados).

s = faz a divisão deste fragmento em fragmentos menores.

y = realiza o stash neste fragmento.

CTRL+C = Aborta o processo de stashing.



Branch com Stash

É possível criar uma branch (ramificação) que contenha as alterações de um determinado stash.

Isso é de grande valia, pois se as alterações na ramificação atual, são diferentes das alterações que estão no stash, você vai poder entrar em conflito ao fazer o popping (**pop**), ou se você tentar aplicar (**apply**) o stash.

```
git stash branch minha-branch stash@{1}
```

Com este comando estamos criando uma nova branch chamada de **"minha-branch"** que contem as modificações do primeiro stash que realizamos.

Limpando Stashs

E para deletar as cópias feitas no stash? Como isso é feito?

Simples, vejamos:

```
git stash drop stash@{1}
```

Com este comando nós estamos deletando somente o primeiro stash que está na pilha.

```
git stash clear
```

Com este comando estamos deletando todos os seus stashes.

Lembrando que entre as chaves, colocamos o número referente ao stash que devemos apagar.

Fazendo o uso do Git Stash

Anteriormente neste material, nós vimos que o exercício anterior, nos ensinou a fazer a cópia das modificações atuais em uma nova branch, para mais tarde continuarmos com as modificações.

Vejamos, como fazer isso usando tudo aquilo que aprendemos sobre git stash.

1) Crie uma nova pasta chamada **'stash'**.

2) Crie um arquivo chamado **'BoasVindas.txt'** que contenha o mesmo texto do exercício anterior.

3) Realize os comandos inicias do git:

```
git init
git add .
git commit -m ‘Configuração Inicial’
```

4) Crie um novo arquivo chamado **‘RegrasDaComunidade.txt’**, com o conteúdo do exercício anterior.

Imaginando que o gestor novamente te pediu para arrumar o link no arquivo **'BoasVindas.txt',** ao mesmo tempo que já estamos com uma modificação em andamento.

Chegou o momento de usarmos o **git stash**! Primeiro, você precisa adicionar os arquivos que modificamos na **STAGE**:

```
git add .
```

E por fim, basta executarmos o comando:

```
git stash
```

A partir desse momento, todas as modificações foram copiadas a **WIP**, nos dando a oportunidade de corrigir o arquivo **‘BoasVindas.txt’**.

Após a correção do arquivo, realize o **commit** deste arquivo:

```
git commit -am “Correções em boas-vindas”
```

Por fim, basta trazer novamente o projeto que estávamos trabalhando, por meio do **apply**:

```
git stash apply
```

Considerando que o gestor nos passou todas as outras regras, basta salvar o arquivo que estamos modificando, e fazer o processo final de commit:

```
git add .
git commit -m “Regras da Comunidade”
```

Stash com Conflito

É importante ressaltar que o Git não traz de volta os arquivos automaticamente quando ele se depara conflitos, ou seja, quando ele vê que arquivos da área de stash também receberam alterações, ele nos pede que resolvamos os conflitos antes de trazer a cópia de volta.

Em casos como esses, assim que tentarmos executar o comando **git stash apply** ou **git stash pop**, olha o que acontece:

```
git stash pop
```

```
git stash apply
```

```
MINGW64:/c/Users/SeuUser/Desktop

SeuUser@DESKTOP-UHDBDBV MINGW64 ~/Desktop/Meu Projeto (master)
$ git stash pop
error: Your local changes to the following files would be overwritten by merge:
        YourFile.txt
Please commit your chances or stash them before you merge.
Aborting
```

Existem dois métodos que podemos seguir para resolver esses conflitos:

Método 1: Caso você descarte na mão as alterações que você fez em arquivos anteriores, você poderá trazer normalmente os arquivos que estão em stash.

Método 2: (Recomendável) Outra forma de resolver os conflitos, é:

1) Faça um commit das alterações feitas fora de stash:

```
git commit -am “alterações”
```

2) Traga todas as alterações que estão na área de stash:

```
git stash apply
```

Como você já tinha um commit anterior, agora você já consegue resolver os conflitos normalmente como se você estivesse realizando um merge:

```
  Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
1 <<<<<<< HEAD (Current Change)
2
3 •••••
4 =======
5
6 •••••
7
8 >>>>>>> master (Incoming Change)
9
```

Após isso, não esqueça de salvar os arquivos para continuar com o processo normalmente:

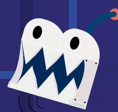
```
git add .
git commit -m “Alterações feitas...”
```

END (y/n)

Gostou desse material?

Então não deixe de acessar a nossa seção de Git & GitHub do básico ao avançado.

ACESSAR



MICILINI.COM

<Seu Portal de CODE>