# GIT & GITHUB

Antes de qualquer interação com o git, você precisa **informar quem é você** para que ele armazene corretamente os dados do autor de cada uma das alterações no código. Para você fazer isso na sua máquina, caso esteja começando a utilizar o git agora, basta digitar os seguintes comandos (estando na pasta do repositório git):

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

git config --**local** user.email "seu@email.aqui"

**CASO O CAMINHO DA SUA PASTA POSSUA ESPAÇOS**, é preciso colocá-lo entre aspas. Ex.: cd **'**Documents/Curso Git e GitHub**'**.

Sobre o **git status**, seguem algumas definições:

* 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 **VER O HISTÓRICO DE ALTERAÇÕES**, cada mensagem de commits feitos, o andamento do nosso projeto etc., o comando que poderemos utilizar para isto é **git log**, que nos mostrará diversas informações, sendo o primeiro deles um hash do commit, uma identificação única de cada commit, isto é, não existem dois commits com o mesmo hash. Assim, conseguiremos realizar algumas manipulações, que veremos mais adiante.

**Obs.**: alternativas ao **git log** (<https://devhints.io/git-log>).

A informação seguinte se refere ao BRANCH, ou "ramo" em que o commit se encontra. Neste caso, verificamos que há HEAD e main. Isto quer dizer que HEAD é o local onde nos encontramos no nosso código, onde acontecem as alterações que fizermos, e que estamos em um ramo denominado main.

Além disso, temos a autoria do commit, e-mail configurado, data de commit, e mensagem. Mas como é que o Git sabe que este e-mail é o seu? Eu já tinha utilizado o Git algumas vezes neste computador, então algumas configurações já estavam definidas, o que é possível fazermos a partir do comando **git config --local** para cada projeto, ou, para a máquina toda, utilizando o **git config --global**.

Poderemos **VISUALIZAR AS CONFIGURAÇÕES SALVAS** por meio de **git config user.name**, ou **git config user.email**.

Todos os commits executados neste repositório serão atribuídos à pessoa com nome Nome da pessoa:

git config --local user.name "Nome da pessoa"

<https://git-scm.com/book/pt-br/v2/Customizing-Git-Git-Configuration>

**SAIR DA TELA DE SCROLL**

Você deve ter reparado que ao executar **git log -p**, o git nos mostrou uma tela onde é possível rolar para baixo e para cima através das setas. Isso não é algo específico do git, mas sim do próprio terminal do sistema operacional. Quando finalizarmos a visualização do log, basta apertar a tecla **q** para voltar "ao normal" em nossa linha de comando.

**IGNORANDO ARQUIVOS**

Pode acontecer de não querermos que determinado arquivo seja monitorado, como no caso de um arquivo de configurações da IDE. Como poderemos fazer para que o Git o ignore?

Existe um arquivo especial do Git, chamado **.gitignore**, e todas as linhas que estiverem nele serão lidos e ignorados pelo Git. Se temos um arquivo denominado ide-config que queremos que seja ignorado, por exemplo, basta o incluirmos em **.gitignore**, digitando ide-config simplesmente. Da mesma forma, se tivéssemos uma pasta ide, incluiríamos ide/, em uma nova linha.

Porém, antes de conferirmos isto com **git status**, precisaremos adicioná-los, com **git add .gitignore**, por exemplo, e **git commit -m "Adicionando .gitignore"**. Neste momento, poderemos nos perguntar: em que momento criamos um commit? Apenas no fim do projeto? Quando finalizarmos tudo ou a cada linha modificada?

Este é um assunto muito extenso, que gera discussões bem calorosas, mas um consenso geral é que jamais devemos commitar código que não funciona. Isto é, o código deve estar sempre no estado funcional para ser commitado. Isto não significa que ele deva ser commitado apenas ao fim do projeto. A recomendação é que se gere um commit após cada alteração significativa.

Existem pessoas que defendem que o commit deve ser gerado ao fim do expediente, outras que dizem que isto deve ser realizado a cada alteração, não existe uma regra, e sim recomendações. Sempre que uma pequena funcionalidade for implementada, ou um bug for corrigido, é possível realizar um commit, para que no fim do dia, um conjunto de commits gere o sistema como um todo, e não um único commit.

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1) No terminal (ou **Git Bash**, no Windows) navegue até a pasta recém criada (utilize o comando cd para navegar entre pastas);

2) Execute o comando git add index.html para marcar o arquivo para ser salvo (*commitado*);

3) Execute git status e confira que o arquivo agora mudou de estado e está pronto para ser salvo (*commitado*);

4) Após adicionar, execute o comando git commit -m "Criando arquivo index.html com lista de cursos". Sinta-se à vontade para alterar a mensagem de *commit*, se desejar;

5) Altere o arquivo **index.html**. Adicione o acento em "Integração Continua", por exemplo;

6) Adicione o arquivo para ser salvo, com git add .;

7) Execute o comando git commit -m "Acento adicionado no curso de Integração Contínua". Sinta-se à vontade para alterar a mensagem de *commit*, se desejar;

8) Execute o comando git log e analise a sua saída. Execute também git log --oneline, git log -p e outras alternativas que quiser testar;

9) Crie um arquivo vazio com o nome que quiser, por exemplo, ide-config;

10) Crie o arquivo **.gitignore** e adicione uma linha com o nome do arquivo recém-criado (ide-config, no exemplo acima);

11) Execute git status e verifique que o arquivo **ide-config** não está na lista para ser adicionado;

12) Adicione (com git add .gitignore) e realize o commit (com git commit -m "Adicionando .gitignore") o arquivo **.gitignore**.

**IMPLEMENTANDO UM REPOSITÓRIO REMOTO**

Chegamos à parte de implementação de um repositório remoto, **um** **SERVIDOR LOCAL para onde possamos enviar nossas alterações que ficarão acessíveis para outras pessoas**. Na pasta que contém os arquivos com os quais trabalhamos até então ("vinicius"), utilizaremos o comando **cd ..** para nos localizarmos na pasta superior, no caso, "git-e-github", e criaremos a pasta "servidor" por meio do comando **mkdir servidor**.

E então acessaremos esta pasta, com **cd servidor**, dentro da qual rodaremos **git init**. Como este servidor será um repositório do Git que somente armazenará as alterações, ou seja, não o acessaremos para editar arquivos, por exemplo, usaremos **git init --bare**, cujo parâmetro indica que este repositório é puro, que contém apenas as **alterações** dos arquivos, e não uma cópia física de cada um dos arquivos.

Isso nos traz algumas facilidades e permite que adicionemos este repositório remotamente em outro. Após a criação do repositório, o Git nos fornece o caminho para ele, que serve como nosso servidor. Copiaremos o caminho, no caso C:/Users/ALURA/Documents/git-e-github/servidor, e voltaremos à pasta "vinicius", onde se encontra nosso projeto, por meio do comando **cd ../<nome da pasta>**.

Executaremos **git status** para nos certificarmos de que estamos no repositório correto, e em seguida, uma vez que passamos a trabalhar com dois repositórios, queremos fazer com que o servidor reconheça o repositório remoto, este endereço, para que ele consiga enviar os dados para lá futuramente.

Se executarmos o comando **git remote**, teoricamente, nada acontece. Mas na verdade, todos os repositórios remotos que o repositório local conhece são listados, que até o momento é nenhum. Portanto, adicionaremos um, com **git remote add local <endereço do da pasta>** C:/Users/ALURA/Documents/git-e-github/servidor, e para quantos repositórios remotos quisermos, poderemos dar algum nome, no caso, local, também incluiremos um caminho, que poderá ser uma URL de um servidor pela internet, um endereço na rede, inclusive de outro computador, qualquer endereço válido para um repositório Git. Neste caso, será uma pasta no próprio servidor.

Depois que pressionamos "Enter", aparentemente nada acontece, e se usarmos o comando **git remote**, o retorno será local. Se quisermos garantir que o endereço esteja correto, poderemos executar **git remote -v**, que faz com que o endereço de local seja exibido. Além disso, indica que os dados deste caminho serão buscados (**fetch**), e enviados para este mesmo caminho (**push**).

Em situações complexas, de uma infraestrutura de redes mais robusta, poderíamos fazer o envio para um local e a busca viria de outro. Não é nosso caso, portanto não nos preocuparemos com isto no momento. Já criamos um repositório remoto, que adicionamos no repositório local, e agora passaremos a imaginar que a Ana está trabalhando conosco e precisa baixar os dados contidos neste repositório.

Voltaremos à pasta "git-e-github" por meio de cd .., e criaremos uma pasta para a Ana, com mkdir ana. Acessaremos a pasta com cd ana, e ela então precisará **CLONAR O REPOSITÓRIO**, é assim que chamamos quando queremos **trazer todos os dados de um repositório remoto para o nosso repositório local pela primeira vez**.

Sendo assim, executaremos **git clone <endereço do diretório>** /c/Users/ALURA/Documents/git-e-github/servidor, para que sejam trazidos os dados do repositório localizado neste endereço. Isso fará com que dentro da pasta "ana" seja criada uma pasta chamada "servidor". Porém, não é o que queremos; queremos que a pasta seja "projeto", por exemplo, e para isso executaremos **git clone** /c/Users/ALURA/Documents/git-e-github**/servidor projeto**.

Após "Enter", somos informados de que o clone foi realizado, mas há um aviso de que o repositório clonado está vazio. Mas não adicionamos o repositório remoto no repositório do Vinicius? Sim, porém não enviamos os nossos dados para ele! Portanto, a Ana não possui acesso a eles, e é por isto que o repositório dela está vazio. A seguir, entenderemos como enviar dados para um repositório e buscar as suas modificações.

No momento, temos o Vinicius, que agora poderá enviar os dados para o servidor, e temos a Ana, e ambos se conectarão ao mesmo servidor. Estes também são os nomes das nossas pastas, uma para representar cada usuário, além do próprio servidor. Então, agora precisaremos fazer com que o Vinicius envie os seus dados para o servidor.

No Git Bash, digitaremos **cd ../**vinicius/ e, depois**, git remote** para confirmarmos a existência de local — mas *como será que incluímos o repositório nele*? Empurraremos as modificações, portanto usaremos o comando git push, que não é o suficiente por si só, uma vez que não estamos sendo explícitos.

O comando será **git push local main**, e assim, serão enviados todos os dados por todos os códigos e alterações feitas até então para nosso repositório que chamamos de "local", dentro de "servidor". Após pressionarmos "Enter", teremos a mensagem de que uma nova branch (ramo) foi criada em "servidor", chamada main.

Vamos nos logar como Ana, digitando **cd ../**ana**/projeto/**, e executar **ls** para verificar se o arquivo HTML está contido ali, o que não acontece, pois o usuário Vinicius enviou os dados para o servidor, mas a Ana não os trouxe para o seu próprio repositório. Para isso, executaremos o comando **git pull**, mas se digitarmos **git remote**, teremos ***origin***. O que é isso? De onde ele vem?

Iremos renomeá-la de local também, por meio de **git remote rename origin local**. Assim, manteremos a paridade com a nomenclatura do Vinicius. Em seguida, executaremos **git pull local main** para trazermos os dados. Ainda falaremos melhor sobre branches, no entanto sabemos que estamos trabalhando com main por ora. Desta vez, com **ls** teremos index.html listado, como gostaríamos.

Para garantir que o conteúdo está igual, no VS Code adicionaremos uma pasta da Ana no projeto, chamada "projeto". Com isto, passaremos a ter a pasta "vinicius" e "projeto", e o index.html é igual para ambos, isto é, os conteúdos estão sincronizados. Além disso, o "ide-config" que adicionamos em ".gitignore" não foi enviado, pois configuramos para que fosse assim, lembra?

Assim, conseguimos começar a sincronizar os dados do Vinicius e da Ana; se ela atualizar algo em alguma parte do código, uma vez estando logados como Ana, utilizaremos git status, teremos o aviso de que a modificação foi realizada, executaremos git add index.html, seguido por git commit -m "Renomeando curso de Integração Contínua". Será que se logarmos como Vinicius conseguiremos verificar esta alteração? Ainda não, pois não enviamos os dados; faremos isto com git push local main. Nos logaremos como Vinicius e, antes de mais nada, se executarmos git status, teremos que não há nada a ser enviado, mas que teremos o que trazer de volta. Vamos executar git pull local main. É exibido que houve uma única alteração, a remoção de uma linha e adição de outra.

Ao executarmos **git log -p**, veremos as modificações realizadas, e se abrirmos o arquivo HTML no VS Code, teremos a alteração implementada no arquivo da pasta do Vinicius também. Agora, passamos a sincronizar os dados e modificações entre os integrantes da nossa equipe.

**SERVIDORES REMOTOS (GitHub)**

Nele, poderemos criar uma conta, e a partir daí passar a criar repositórios Git de forma muito simples. Feito o login, independentemente do quão familiar você esteja com o site, é possível clicar no símbolo de + localizado no canto superior direito para criar um novo repositório, por meio da opção "New repository".

Na nova página, poderemos definir o **criador do repositório** (**Owner**) e o **seu nome** (**Repository name**), que pode ser qualquer um. Neste caso, será "alura-git". Daremos uma **descrição** (**Description**), "Lista de cursos para controlar no GIT". O repositório pode ser **configurado como público ou privado**, dependendo da conta que tivermos. Normalmente, os repositórios privados só ficam disponíveis para usuários pagantes. Caso você seja usuário de plano grátis, será possível apenas criar repositórios públicos.

Após clicarmos no botão "Create repository" no fim da página, seremos redirecionados a outra, com dicas sobre como poderemos criar um novo repositório por linhas de comando, entre outras. No nosso caso, já temos um repositório local, arquivos commitados, e tudo o mais, então optaremos pelo envio deste repositório, com **git remote add origin git@github.com:**CViniviusSDias/alura-git.git, uma sintaxe talvez não muito familiar, para o qual precisaríamos definir chave de acesso, algo mais seguro, porém complicado.

Na parte superior desta página, onde se lê "**Quick setup** — if you've done this kind of thing before", selecionaremos "**HTTPS**" em vez de "SSH", de forma que, toda vez que precisarmos enviar os dados ou adicionar um repositório durante envio ou quando formos trazê-lo de volta, precisaremos digitar uma senha.

No Git Bash, logaremos como Vinicius e colaremos o comando, feito isso, no site do GitHub é indicado que devemos enviar os dados do repositório com **git push origin main**.

**Obs**.: **git push** **-u origin main** define que, sempre que usarmos git push e estivermos na main, o envio seja feito para origin. Ou seja, a partir de então poderemos executar simplesmente git push. Eu particularmente prefiro não seguir esta abordagem, e sempre digitar qual o repositório e qual branch quero enviar, para manter um controle maior do meu lado. Sendo assim, no meu caso executo **git push origin main**.

Ao executarmos o comando, será aberta uma janela de login para o GitHub, após o qual os dados serão enviados adequadamente. Caso você não esteja utilizando o Windows, a senha será solicitada diretamente via Terminal. Então, quando atualizarmos nossa página no GitHub, teremos os nossos códigos disponíveis, incluindo uma lista de commits, com as alterações feitas em cada um deles, e suas autorias.

Existem formas mais rebuscadas, um pouco mais profissionais de organizar nosso sistema de controle de versão, e começaremos a falar sobre branches, por exemplo, a seguir!

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1) Crie uma pasta nova em seu computador;

2) No terminal (ou **Git Bash**, no Windows) navegue até a pasta recém criada (utilize o comando cd para navegar entre pastas);

3) Execute o comando git init --bare;

OBS: Não se esqueça do parâmetro **--bare**. Caso tenha executado o comando init sem esse parâmetro, execute na sequência o seguinte comando: git config core.bare true.

4) Navegue até a pasta onde se encontra o seu projeto;

5) Execute o comando git remote add local {caminho}. Substitua {caminho} pelo caminho completo da pasta recém criada;

6) Crie uma nova pasta em seu computador, para representar o trabalho de outra pessoa;

7) No terminal (ou **Git Bash**, no Windows) navegue até a pasta recém criada;

8) Execute o comando git clone {caminho} projeto. Substitua {caminho} pelo caminho completo da pasta que criamos no primeiro passo;

9) Observe que o repositório clonado está vazio;

10) Acesse a pasta Projeto e execute o comando 'git remote rename origin local' para renomear o repositório local da outra pessoa de "origin" para "local";

11) Navegue até a pasta onde se encontra o seu projeto original;

12) Execute o comando git push local main para enviar as suas modificações para o seu servidor;

13) Navegue até a pasta criada no passo 6;

14) Execute o comando git pull local main para baixar as modificações;

15) Abra o seu navegador e acesse <http://github.com/>;

16) Crie uma conta;

17) Crie um novo repositório, clicando no símbolo de adição no canto superior direito;

18) No terminal (ou **Git Bash**, no Windows) adicione, ao seu projeto inicial, o repositório remoto recém criado (os comandos são mostrados pelo próprio GitHub);

19) Execute git push origin main para enviar as suas alterações para o repositório no GitHub.

Sobre este trabalho compartilhado, temos dois usuários, Vinicius e Ana, desenvolvendo o mesmo projeto, e normalmente duas pessoas diferentes trabalham em partes diferentes de um projeto. Sabemos, no entanto, que este tal de main está sendo compartilhado entre eles, então, para evitarmos complicações e, enquanto o Vinicius estiver trabalhando no cabeçalho da página, por exemplo, e a Ana na lista de cursos, **seria interessante termos uma maneira de SEPARAR OS RAMOS DE DESENVOLVIMENTO para sabermos exatamente no que cada um está mexendo**, e para que não haja interferências no código compartilhado.

Talvez isto não tenha ficado tão claro, mas consideremos o seguinte: o Vinicius passará a trabalhar em tudo que estiver contido entre as tags <head> do arquivo index.html. Então, informaremos ao nosso controle de versões que, a partir de um determinado commit, um dos usuários alterará apenas um trecho específico, enquanto o outro usuário informará do seu trecho em desenvolvimento, também.

Estas ramificações do trabalho são uma das formas de com que podemos trabalhar, em relação aos *branches* do Git. Por padrão, se executarmos **git branch** no Git Bash, teremos um único Branch (main), e é exatamente isto que o Git Bash nos mostra ao fim da linha. No entanto, poderemos criar outros. No caso de trabalharmos somente no título, por exemplo, utilizaremos o comando **git branch <nome do arquivo>**titulo, que criará este branch, embora tenhamos que mudar para ela manualmente, com **git checkout <nome do arquivo>**titulo.

A partir daí, estaremos trabalhando na linha de desenvolvimento titulo. Para isso ficar um pouco mais claro, utilizaremos uma ferramenta chamada Visualizing Git (<https://git-school.github.io/visualizing-git/>). Do lado esquerdo da página digitaremos os comandos, e o resultado destes serão exibidos do lado direito. Em se tratando do trabalho conjunto de Ana e Vinicius, teremos duas linhas de desenvolvimento distintas e independentes entre si.

Abriremos o VS Code e alteraremos o título, de <title>Cursos da Alura</title> para <title>Cursos de DevOps da Alura</title>. No Git Bash, estamos logados como Vinicius, e em titulo. Executaremos git status, verificaremos que há uma alteração, que adicionaremos com git add index.html, seguido de git commit -m "Alterando título da página".

Desta vez, se utilizarmos git log, dentre as informações que o comando nos traz, estão todos os commits realizados, incluindo o último, que é indicado como sendo o último commit realizado na main. O commit do título alterado só aparece na branch titulo, e se fizermos outra alteração no mesmo título, e refizermos todo o processo de adição, commit e verificação do log, teremos que até a mensagem "Renomeando curso de Integração Contínua" é feito na main.

Assim, somente a branch titulo possui as alterações feitas a partir de "Alterando título da página". Se precisarmos alterar algo no commit de "Renomeando curso de Integração Contínua", que não é influenciado pelo título, basta utilizarmos git checkout main para retornarmos à branch correspondente.

Feito isso, ao executarmos git log, não teremos acesso àqueles commits em titulo. Isso é bem interessante! Usaremos git checkout titulo para voltarmos, e passaremos a lidar com a Ana, que trabalhará com as listas de cursos. Criaremos, portanto, um branch com git branch lista, e depois faremos o checkout para a lista.

Entretanto, existe um **atalho que cria um branch e já passa para ele**: **git checkout -b** **<nome da branch>**lista, que usaremos. Com isso, a Ana está na branch lista, então poderemos abrir o projeto da Ana no VS Code e adicionar um curso em uma nova lista, como <li>Kubernetes</li>, junto aos demais. No Git Bash, digitaremos git status, verificaremos que há uma modificação, adicionaremos todas elas com git add ., e commitaremos com git commit -m "Adicionando curso de kubernetes".

Assim, a Ana e o Vinicius estão trabalhando ao mesmo tempo em branches independentes de um mesmo projeto. Mas sabemos que em nosso repositório chamado local, por enquanto, temos apenas a branch main. Isso nos leva a assumir que esta branch é a nossa linha de desenvolvimento padrão, ou seja, nosso ramo principal, onde os códigos devem estar quando estiverem prontos, certo? Então, como será que fazemos para trazer os dados das branches titulo e lista para a main?

Branches ("ramos") são utilizados para desenvolver funcionalidades isoladas umas das outras. A branch main 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.

Nesta aula, entenderemos melhor como trabalhar com estes ramos, mas é muito importante que você entenda seu propósito.

Em outros treinamentos aqui na Alura, falaremos mais sobre estratégias para organizar suas branches, então não precisa se preocupar tanto com isso agora!

Estamos entendendo como trabalhar com linhas de desenvolvimento diferentes, mas **como é que conseguiremos TRAZER O TRABALHO QUE FIZEMOS EM UMA DELAS PARA OUTRA?** Porque, recapitulando, eu, como Vinicius, tenho duas branches, titulo e main, e trabalhamos na primeira. Porém, no repositório que se encontra na pasta "servidor", só temos a branch main, então sabemos que esta linha é a principal, onde queremos depositar o código que funciona. Iremos trabalhar na titulo, mas em algum momento precisaremos trazê-la para a main. Na ferramenta Visualizing Git criaremos a branch titulo e passaremos a trabalhar nela, com **git checkout -b** **<nome do arquivo>**titulo. Faremos um commit com **git commit -m "**Editando título**"**, e outro, com **git commit -m "**Adicionando lista no título**"**.

Temos um problema: reparem que nosso curso de Docker na listagem de index.html está com este nome, mas deveria estar como "Docker: Criando containers sem dor de cabeça", e precisaremos corrigir isto. Isso, porém, não tem nada a ver com nossas alterações de títulos, que não está finalizada. Então precisaremos retornar à main e, a partir daí, corrigir o bug.

Utilizaremos git checkout main, e depois git commit -m "Corrigindo bug". Agora, sim, poderemos voltar à branch titulo e finalizá-lo. Analisando com calma, porém, entendemos que esta branch já está finalizada. Então, de que forma trazemos este trabalho, os dados desta linha em específico, para a que contém head e main?

Ou seja, queremos unificar estas duas linhas, portanto usaremos o comando **git merge** **<nome do arquivo>**titulo, e isto fará com que o Git automaticamente crie um commit com o branch atual e todo o conteúdo de nossa branch titulo. Na prática, estando logados como Vinicius, o que acontece é que, ao surgimento de um bug, as alterações de titulo não podem influenciar nesta correção de bug.

Sendo assim, retornaremos à main, branch que não contém as alterações referentes a titulo. Após a alteração no projeto, faremos a adição e o commit normalmente, no Git Bash, e por fim executaremos git merge titulo, como visto anteriormente. Quando dermos um "Enter", será criado um commit de merge, ou seja, de junção de duas branches. Poderemos editar a mensagem exibida, mas caso não queiramos, para salvarmos e confirmarmos a mensagem, pressionaremos "**:x + Enter**" no editor Vim.

Feita a junção, passamos a ter, na branch main, os dados do título alterado. Porém, se executarmos git log, não teremos os dois commits separadamente, e sim um referente ao merge. O Git cria isto para nós. Então, como será que poderemos fazer com que, em vez do Git criar este commit, ele pegue os dois commits e os adicione em nossa branch main? Como faremos com que ele mova estas branches e atualize a main apenas com os dois commits, sem criar um de merge? Veremos isto a seguir!

**git checkout main** e **git merge <nome do arquivo>**titulo

Anteriormente, vimos como unir o trabalho de duas branches desenvolvidas separadamente. No entanto, não queremos gerar um commit a mais, de merge, dependendo da estratégia utilizada para gerar os commits, isto pode acabar atrapalhando ou "poluindo" o log. Assim, o que queremos é **ATUALIZAR A BRANCH MAIN COM OS COMMITS DA BRANCH** titulo, **DE MODO A TERMOS CADA COMMIT ESPECÍFICO NA LINHA DE DESENVOLVIMENTO MAIN**.

Na ferramenta Visualizing Git executaremos clear para limparmos a tela, e repetiremos o processo com **git checkout -b** **<nome do arquivo>**titulo para gerarmos dois commits (git commit duas vezes). Na branch main, corrigimos um bug, portanto geraremos outro commit. E então, da branch titulo, queremos trazer os demais commits para antes de main atualizando as duas branches.

Para isto, estando na main, queremos basear esta branch em titulo, assim, executaremos **git rebase** **<nome do arquivo>**titulo, e o Git pegará os commits na branch titulo, atualizando main, que possui todos os commits contidos em titulo, além do commit que havia nela mesma. Deste modo, **geramos uma única linha**, sem confusões.

No Git Bash, executaremos **git log** novamente, e teremos a informação de commit de merge; de que forma conseguiremos visualizar isso de forma mais interessante? Se digitarmos **git log --graph**, serão exibidas linhas específicas representando o desenvolvimento, uma boa alternativa ao Visualizing Git.

Vamos fazer uma alteração na branch titulo, com git checkout titulo, e no VS Code alteraremos a primeira letra de "Cursos" para que fique em maiúscula. Adicionaremos o arquivo e o commitaremos, e depois iremos à branch main para trazermos os commits de titulo para ela, por meio de git rebase titulo. Ao executarmos git log mais uma vez, teremos o commit "Corrigindo nome do curso de Docker" acima de "Cursos com letra maiúscula", porque ele foi adicionado logo antes. Isto é, o commit que fizemos na branch titulo foi adicionado logo antes do commit feito em main, exatamente como vimos no Visualizing Git.

Ou seja, o rebase atualiza a branch, mantendo o trabalho dela como sendo o último, para que não se gere este tipo de confusão. Com isso, temos as correções realizadas tanto no título quanto na lista, e poderemos fazer o **git push local main**, logados como Vinicius. Tudo está atualizado! Podemos, então, nos logar como Ana, usar **git checkout main** e **git pull local main** para atualizar os dados também.

De acordo com o que foi estudado, então qual é a diferença entre os comandos **rebase** e **merge**?

O merge junta os trabalhos e gera um merge commit.

O rebase aplica os commits de outra branch na branch atual.

Com isso, evitamos os commits de merge. Há uma longa discussão sobre o que é "melhor": rebase ou merge. Estude, pesquise, e tire suas próprias conclusões. Aqui tem um artigo (de milhares outros) que cita o assunto: <https://medium.com/datadriveninvestor/git-rebase-vs-merge-cc5199edd77c>.

Mas lembram que a Ana estava trabalhando em lista? Voltaremos para lá com git checkout lista para atualizarmos os dados, no caso, o título do curso de Docker. Commitaremos, faremos git log -p para garantir que a atualização foi feita, faremos um checkout para main. Teremos que houve uma alteração feita pelo Vinicius, e outra feita pela Ana, na mesma linha. O que será que acontecerá se tentarmos juntar o trabalho deles?

Vimos um caso interessante acontecer: o Vinicius corrigiu um bug, isto é, alterou um determinado trecho de código, porém a mesma tarefa foi executada também pela Ana. O que será que irá acontecer se juntarmos estes trabalhos? Dentre merge e rebase optaremos pelo primeiro, embora o resultado deles seja o mesmo.

**RESOLVENDO CONFLITOS**

Logados como Ana, utilizaremos git merge lista, e o Git nos informa que existe um conflito, e que houve falha no merge automático. É recomendado que corrijamos os conflitos primeiro, e depois commitemos o resultado. Ao voltarmos ao arquivo no VS Code, há indicações coloridas referenciando o conflito do Git, mas para o caso do uso de um editor de texto que não as tenha, focaremos somente no texto, ignorando as cores.

Entre as linhas **<<<<<<< HEAD (Current Change)** e **=======**, estão os **dados do commit atual**, na main. E entre as linhas **=======** e **>>>>>>> lista (Incoming Change)**, são os **dados que estamos tentando trazer** da branch lista. Ou seja, **é exibida exatamente a diferença entre ambos**. E tudo que precisamos fazer para corrigir este conflito é remover as informações indesejadas, sem que haja duplicação.

Editaremos e salvaremos o arquivo, retornaremos ao Git Bash e executaremos **git status**, e teremos a informação de que houve uma modificação em dois lugares, na branch atual e aquela que estamos tentando unificar. Feita a correção, simplesmente utilizaremos **git add <nome do arquivo/diretório>**index.html, e então **git commit** para que o commit de merge seja realizado. Desta vez, se executarmos **git log --graph**, teremos a indicação do merge de lista. Em seguida, poderemos usar **git push local main**.

Vamos imaginar que o Vinicius corrija o título do curso de Vagrant para "Vagrant: Gerenciando máquinas virtuais", e nos logar como Vinicius, solicitar status, adicionar e commitar a alteração. Enviaremos as informações, e o que acontece é que enquanto o Vinicius estava trabalhando, a Ana enviou outra informação, o commit de merge.

**É NECESSÁRIO**, então, antes de enviarmos quaisquer dados e alterações, **GARANTIR QUE ESTAMOS TRABALHANDO COM A VERSÃO MAIS RECENTE DO CÓDIGO**. Isso significa que, antes do envio, precisaremos trazer este código de volta (**git pull local main**). Agora, sim, será feito o merge da main que está no "servidor" com esta.

Assim, poderemos confirmar que tudo está como gostaríamos no VS Code, e depois enviar a alteração, com **git push local main**. Sempre que formos iniciar um desenvolvimento novo, sabemos que precisaremos verificar se há alguma alteração lá antes de enviarmos os dados. Antes da Ana continuar e fazer alguma alteração nova, ela sabe que é necessário verificar se não há nenhuma alteração ali, com git pull local main.

As informações são trazidas conforme esperado pelo Git Bash. Deste modo evitamos maiores conflitos, mas se acontecer, já vimos que conseguimos resolvê-los tranquilamente. Entendemos como trabalhar com repositórios remotos, em equipe, com branches independentes, e como uni-las, seja por meio do merge ou do rebase.

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1) Execute o comando git branch e veja que apenas a *branch* main existe no seu repositório;

2) Execute o comando git branch titulo e logo após execute o comando git branch. Veja que uma nova *branch* foi criada;

3) Agora, para começar a trabalhar nesta *branch*, digite git checkout titulo;

4) Execute novamente git branch e confira que agora você está na *branch* chamado titulo;

5) Altere o título da página **index.html** para "Cursos de DevOps da Alura";

6) Adicione as alterações com git add index.html;

7) Faça o *commit*, com git commit -m "Alterando título da página";

8) Execute o comando git log e confira o novo *commit*;

9) Altere o título da página para "Lista de cursos de DevOps da Alura";

10) Repita os passos 6 e 7, para adicionar um novo *commit*, alterando a mensagem;

11) Repita o passo 8 para conferir o novo *commit*;

12) Execute o comando git checkout main para voltar à linha de desenvolvimento main;

13) Execute git log para conferir que os últimos dois *commits* não estão lá. Confira se o conteúdo do seu arquivo também voltou ao seu estado original;

14) Na pasta criada para representar o trabalho de outra pessoa na aula anterior:

* Execute git checkout -b lista para criar uma nova *branch*, chamada lista e passar a trabalhar nela;
* Adicione o curso de "Kubernetes" na lista;
* Repita os passos 6 e 7 para adicionar um novo *commit*, alterando a mensagem;
* Execute o comando git checkout main para voltar à linha de desenvolvimento main;

15) Volte para a pasta que representa o seu próprio trabalho;

16) Altere o nome do curso de Docker para "Docker: Criando containers sem dor de cabeça";

17) Repita os passos 6 e 7 para adicionar um novo *commit*, alterando a mensagem;

18) Execute o comando git merge titulo para trazer o trabalho feito na *branch* titulo para a *branch* main;

19) Execute o comando git log --graph para ver as linhas de desenvolvimento (*branches*);

20) Execute git checkout titulo para trabalhar na *branch* chamada titulo;

21) Altere o título para ter a palavra "Cursos" com letra maiúscula;

22) Repita os passos 6 e 7 para adicionar um novo *commit*, alterando a mensagem;

23) Execute o comando git checkout main para voltar à linha de desenvolvimento main;

24) Execute o comando git rebase titulo;

25) Execute o comando git log e confira que o *commit* foi adicionado antes do *commit* realizado diretamente na *branch* main;

26) Execute o comando git push local main para enviar suas alterações para o repositório remoto que criamos na última aula;

27) Na pasta criada para representar o trabalho de outra pessoa na aula anterior:

* Execute o comando git pull local main para baixar as alterações que você já realizou;
* Execute o comando git checkout lista para continuar trabalhando na lista de cursos;
* Altere o nome do curso de Docker para "Curso de Docker: Criando containers sem dor de cabeça";
* Repita os passos 6 e 7 para adicionar um novo *commit*, alterando a mensagem;
* Execute o comando git checkout main para voltar à linha de desenvolvimento main;
* Tente juntar seu trabalho com git merge lista;
* Veja que há conflitos. Corrija-os, deixando apenas a linha com o nome correto do curso;
* Execute o comando git add index.html para informar que os conflitos neste arquivo foram corrigidos;
* Execute o comando git commit para que o Git finalize o *merge*;
* Execute o comando git push local main para enviar as suas alterações;

28) Volte para a pasta que representa o seu próprio trabalho;

29) Altere o nome do curso de Vagrant para "Vagrant: Gerenciando máquinas virtuais";

30) Repita os passos 6 e 7 para adicionar um novo *commit*, alterando a mensagem;

31) Tente executar o comando git push local main. Veja a falha;

32) Execute o comando git pull local main para trazer as alterações da outra pessoa;

33) Agora sim, execute o comando git push local main para enviar as alterações.

Considerando estes aprendizados, como será que poderemos navegar no histórico do nosso projeto? E desfazer uma alteração?

**DANDO CTRL + Zs**

Conseguimos nos conectar com repositórios remotos, adicionar mais de um deles, conseguimos compartilhar o código com colegas de equipe, organizar nosso versionamento em branches, linhas de desenvolvimento distintos... É muito comum começarmos a desenvolver e fazer testes e termos que desfazer estas alterações. De que forma será que conseguimos desfazê-las com o Git? Será que ele possui alguma espécie de atalho "Ctrl + Z"?

No VS Code, passaremos a trabalhar com o projeto do Vinicius. Na lista de cursos, trocaremos "Ansible" por "Ansible: Infraestrutura como código". Salvaremos o arquivo index.html, visualizaremos a página e acharemos que não ficou tão interessante. Reparem que não fizemos o commit, a adição, nem nada disso, apenas editamos o código. Por se tratar de um único arquivo, a alteração em uma linha poderia ser desfeita com "Ctrl + Z", mas imaginemos um projeto grande, em que fazemos várias alterações, e só então entendemos que não está como queremos. Teríamos que ir desfazendo-as arquivo por arquivo, ou que só percebemos que não gostamos da alteração após ter passado um dia inteiro, impossibilitando o uso do atalho?

No Git Bash, logados como Vinicius, usaremos **git status**, o que nos traz algumas informações. É identificado que houve modificações no arquivo, que ainda não foram commitadas. Para isso, precisaríamos chamar o git add, no entanto, é indicado que, se quiséssemos descartar as alterações, poderemos chamar **git checkout -- <nome do arquivo>** que queremos desfazer.

O git checkout, portanto, serve para navegarmos em estados do repositório, seja por meio de branches ou desfazendo modificações no arquivo. Sendo assim, neste caso é possível executarmos git checkout -- index.html. Se executarmos git status novamente, não teremos nada a ser commitado, e se abrirmos o arquivo no VS Code, verificaremos que o teste foi realmente desfeito.

Porém, e se depois da alteração no título do curso no VS Code fossemos diretamente ao Git Bash e usássemos git add index.html, mas antes do commit, testássemos e víssemos que não ficou como gostaríamos? Queremos desfazer uma alteração que já foi marcada para ser commitada, então usaremos git status para verificar se o próprio Git nos traz alguma ajuda.

É exibido que há mudanças a serem commitadas, e que poderemos utilizar **git reset HEAD <nome do arquivo>** a ser desmarcado como algo que precisa passar pelo commit. Vamos fazer isso! Para este reset, é preciso enviar um estado, e como ele voltará para o HEAD, para o local de trabalho, isto é, o estado em que ainda estaremos trabalhando.

Feito isto, com **git status** confirmaremos que as alterações continuam ali, porém não estão mais marcadas para serem commitadas. Sendo assim, basta utilizarmos **git checkout -- <nome do arquivo>**, o que fará com que não tenhamos mais nada a ser commitado, uma vez que a alteração foi desfeita com sucesso.

Agora, imaginemos o pior dos casos: após fazermos a alteração no título do curso, a adição e o commit do arquivo, acabamos verificando que introduzimos um bug, e que este código não podia ter sido commitado. Como será que desfazemos um commit já realizado? Usando **git log**, teremos o **hash do commit**, certo? Iremos copiá-lo, colar na linha de comando, juntamente com **git revert**. Isso fará com que o commit informado seja desfeito, criando outro. Ao ser rodado, portanto, ele irá gerar um commit cuja mensagem pode ser alterada, usaremos "**:x**" para salvarmos e sairmos da tela. Ao fazermos **git log** mais uma vez, teremos dois commits, um com a alteração do nome do curso, e outro com a reversão deste. No VS Code, teremos a versão inicial, conforme gostaríamos. Desta forma, conseguimos desfazer nossos trabalhos e eventuais descuidos, e permite testes.

Com o git checkout nós desfazemos uma alteração que ainda não foi adicionada ao index ou stage, ou seja, antes do git add. Depois de adicionar com git add, para desfazer uma alteração, precisamos tirá-la deste estado, com git reset. Agora, se já realizamos o commit, o comando git revert pode nos salvar.

Prosseguindo, imaginemos um código que ainda não está pronto para ser commitado por estar em um estágio não funcional, mas que não queremos desfazê-lo. Há uma nova demanda, uma tarefa a ser feita; será que conseguimos salvar o arquivo temporariamente, com o Git? Veremos isto a seguir!

**E se quisermos GUARDAR UMA PARTE DE UMA ALTERAÇÃO PARA DEPOIS, como faremos?** Alguma modificação no código, para voltarmos a trabalhar nela depois, sem que precisemos commitá-la ou desfazê-la? Alteraremos novamente o título do curso de Ansible para "Ansible: Infraestrutura como código", no entanto, ainda precisaremos confirmar se este é o nome exato do curso, com mais calma, depois, pois fomos interrompidos por uma nova tarefa mais urgente. No Git, existe um conceito denominado Stash, e por meio de **git stash** conseguimos salvar todas as alterações, no caso, somente o arquivo index.html, para um local temporário, sem necessidade de um commit ou de se gerar um commit para isto. Se, após git stash executarmos **git stash list**, teremos uma lista de tudo que estiver salvo nestas condições.

Vamos, então, alterar o nome do curso de Kubernetes, para "Kubernetes: Introdução a orquestração de containers". Executaremos, então, git status, seguido de git add index.html, git commit -m "Alterando o nome do curso de Kubernetes". Feito isto, queremos voltar a trabalhar com as alterações no curso de Ansible. No VS Code, teremos que as alterações em relação ao título do Kubernetes já foram realizados, porém os de Ansible, não. Queremos trazer os dados armazenados pelo **git stash** ao diretório de trabalho. Há **duas opções**: executarmos **git stash list**, e em seguida passarmos o número da stash em **git stash apply 0**, aplicaremos estas modificações, porém elas continuarão na stash. Para a remoção, poderemos usar **git stash drop**.

No caso de querermos fazer ambas as ações ao mesmo tempo, ou seja, pegar a última alteração adicionada à stash, e já removê-la de lá, utilizaremos **git stash pop** que, ao ser executado, realiza o merge com as modificações que já temos e aplica aquelas que já estavam salvas lá. Desta vez, ao consultarmos o VS Code, teremos o código atualizado adequadamente, com o trecho alterado e salvo temporariamente sem necessidade de commit.

Vamos alterar mais uma vez o título do curso de Ansible, para "Ansible: Sua infraestrutura como código", e no Git Bash faremos a adição e commit. Feito isso, realizaremos o envio, pois é sempre importante manter o nosso repositório remoto atualizado. Executaremos o comando **git log --oneline**, e perceberemos que temos vários commits, dentre os quais um de merge, Merge branch 'lista'.

Veremos a seguir como fazemos com que o nosso código volte para o estado em que estava no momento em que aplicamos este commit!

Q**uando precisamos pausar o desenvolvimento de alguma funcionalidade** ou correção, **no meio para trabalhar em outra coisa,** antes de finalizar, talvez não seja interessante realizar um **commit**, pois o nosso código pode não estar funcionando ainda. Nesse caso é interessante salvar o trabalho para podermos voltar a ele depois. Portanto, utilizamos o **git stash** para isto.

**VIAJANDO NO TEMPO**

Queremos observar o projeto como um todo no momento em que aplicamos um determinado merge, ou então um pouco antes, em outro commit. Executamos o git revert anteriormente com aquele commit e o hash, mas poderemos executar as manipulações em um commit com os seus primeiros caracteres. O comando **git log --oneline**, por exemplo, nos traz os hashs com apenas os sete primeiros caracteres, o suficiente para identificá-los de forma única.

No caso, queremos navegar ao commit de hash ea539b3. Já conversamos que o comando **git checkout <nome da branch/commit>** muda o estado da aplicação, seja desfazendo alterações, seja navegando entre branches ou commits. Assim, é possível utilizarmos git checkout ea539b3, e com isso a mensagem que se exibe indica que estamos em um estado de cabeça (**HEAD**) desanexado (**detached**) do controle de versões. Isto é, não estamos mais em nenhum branch, e sim em um commit específico. Não estamos em uma linha bem definida de commit, uma linha de trabalho bem definida do Git. Então, poderemos fazer algumas modificações experimentais, mas também descartar qualquer elemento deste branch sem fazer mais nada. Isto quer dizer que se voltarmos à main, tudo que commitarmos aqui será ignorado. Se quisermos manter os commits feitos a partir deste ponto, será necessário criar uma nova branch.

Reabriremos a ferramenta Visualizing Git para executarmos três git commits seguidos. Para verificarmos como estava o projeto durante o segundo commit, usaremos git checkout 54727de. Isto fará com que HEAD se locomova até ali, no lado direito da tela, e o estado do código esteja sendo exibido no segundo commit. Se realizarmos qualquer alteração, incluindo outro git commit, o HEAD se locomoverá para um lugar sem nome, uma branch inexistente. E se fizermos git checkout main nunca mais conseguiremos acessar o commit em que estávamos anteriormente, que fica desanexado das linhas de desenvolvimento. Repetiremos o comando git checkout 54727de e, se quisermos fazer alterações que sejam salvas a partir daqui, será necessário criar uma branch antes, a ser modificado a partir deste commit. Usaremos **git checkout -b <nome da Branch>**novo-branch, de forma a não estarmos mais desassociados da linha de desenvolvimento, o que se confirma se realizarmos um novo commit.

Poderemos fazer o **git checkout main**, mas se em algum momento quisermos voltar a trabalhar em novo-branch, basta usarmos o git checkout. Assim, conseguimos navegar entre os estados da nossa aplicação, de fato, "viajar no tempo" no projeto. Temos bastante conhecimento e poderemos fazer praticamente tudo o que é necessário para um trabalho do dia a dia, com o sistema de gerenciamento de versões.

A descrição do comando **git checkout --help**, em uma tradução livre é: "Atualizar os arquivos na working tree para ficarem na versão especificada. [...]". Basicamente, podemos deixar o nosso código no estado do último commit de uma branch, de um commit específico, ou mesmo tags (que veremos adiante). Portanto, o comando **git checkout** serve para deixar o nosso código em determinado estado.

Nesta aula, aprendemos:

* Que o Git pode nos ajudar a desfazer alterações que não vamos utilizar;
* Que, para desfazer uma alteração **ANTES de adicioná-la para commit** (com git add), podemos utilizar o comando **git checkout -- <arquivos>**;
* Que, para desfazer uma alteração **APÓS adicioná-la para commit**, antes precisamos executar o **git reset HEAD <arquivos>** e depois podemos desfazê-las com **git checkout -- <arquivos>**;
* Que, para revertermos as alterações realizadas em um commit, o comando **git revert** pode ser a solução;
* Que o comando git revert gera um novo commit informando que alterações foram desfeitas;
* Que, para guardar um trabalho para retomá-lo posteriormente, podemos utilizar o **git stash**;
* Que, para visualizar quais alterações estão na stash, podemos utilizar o comando **git stash list**;
* Que, com o comando **git stash apply <numero>**, podemos aplicar uma alteração específica da stash;
* Que o comando **git stash drop <numero>** remove determinado item da stash;
* Que o comando **git stash pop** aplica e remove a última alteração que foi adicionada na stash;
* Que o **git checkout** serve para deixar a cópia do código da nossa aplicação no estado que desejarmos:
  + **git checkout <branch>** deixa o código no estado de uma branch com o nome <branch>;
  + **git checkout <hash>** deixa o código no estado do *commit* com o hash <hash>.

Mas como informamos que temos uma versão pronta do sistema, um "entregável"? Será que o Git nos ajuda a gerar este tipo de projeto pronto para ser lançado?

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1) Na pasta que representa o seu projeto, faça uma alteração qualquer no arquivo **index.html**;

2) Execute o git status e veja que há uma alteração para adicionar;

3) Execute o comando git checkout -- index.html. Confira se sua alteração foi desfeita;

4) Novamente, faça alguma alteração no arquivo **index.html**;

5) Execute o comando git add index.html;

6) Execute o comando git reset HEAD index.html para trazer o arquivo **index.html** de volta para a HEAD do projeto (remover do stage, que é o que será enviado para o commit);

7) Repita o passo 3;

8) Faça mais uma vez alguma alteração no código;

9) Execute o comando git add index.html e o comando git commit -m "Alterando o código" para realizar um commit;

10) Execute o comando git log e copie o hash deste *commit* recém criado;

11) Rode o comando git revert {hash}, substituindo {hash} pelo *hash* que você copiou anteriormente;

12) Confira que suas alterações foram desfeitas;

13) Mude o nome do curso de Ansible para "Ansible: Infraestrutura como código";

14) Execute o comando git stash para salvar estas alterações na stash;

15) Altere o nome do curso de Kubernetes para "Kubernetes: Introdução a orquestração de containers";

16) Execute o comando git add index.html e o comando git commit -m "Alterando o nome do curso de Kubernetes" para realizar um *commit*;

17) Execute o comando git stash pop para trazer a última alteração da *stash*;

18) Execute o comando git add index.html e o comando git commit -m "Alterando o nome do curso de Ansible" para realizar um *commit*;

19) Execute o comando git push local main para enviar todas as suas alterações;

20) Execute o comando git log --oneline para ver os *commits* de forma resumida. Copie o *hash* do *commit* de merge com a *branch* lista;

21) Execute o comando git checkout {hash} substituindo {hash} pelo *hash* que você copiou;

22) Veja que diversas alterações não estão mais presentes;

23) Execute git checkout main para voltar à linha principal de desenvolvimento.

Temos um projeto finalizado, com todas as alterações necessárias no conteúdo entre tags <titulo>, todos os nomes da listagem de cursos estão corretos, já vimos como trabalhar em equipe, com repositórios remotos, branches independentes, corrigindo conflitos, alocando dados, atualizando branches, enfim, vimos bastante conteúdo. Entretanto, no momento de finalizarmos, queremos verificar o que houve em cada commit, para garantir que nenhum bug foi adicionado no projeto, e entender o que de fato cada commit gerou no código. Como conferiremos as diferenças entre commits?

**VENDO AS ALTERAÇÕES**

Logados como Vinicius, já entendemos que, se utilizarmos **git log -p**, conseguiremos ver o que foi alterado em código commit a commit. Existe um comando do Git, bem interessante e poderoso, que é o **git diff**, capaz de exibir estas diferenças. Ao tentarmos executá-lo, porém, nada é exibido, isso porque por enquanto não há nada alterado no nosso código, que não tenha sido salvo. Então, para entendermos as diferenças entre dois commits, precisaremos informar quais são através de suas hashs, no caso, ea539b3 até (..) 6ca12ac. Por meio deste comando, **visualizamos todas as alterações feitas, marcadas em cores diferentes**. Além disso, caso estejamos modificando algo, como acrescentando um novo curso na listagem, no código, e queiramos verificar o que foi alterado, poderemos simplesmente usar o git diff, que nos mostra o que foi alterado e que ainda não foi adicionado para commit. Mas a partir do momento em que adicionamos o arquivo, este comando não nos mostra mais o que existe de diferente.

Traremos o arquivo de volta após **git status** com **git reset HEAD <nome do arquivo>**index.html, e com **git status** conferiremos que ele está pronto para ser adicionado ao commit. Vamos desfazer as alterações com **git checkout --** **<nome do arquivo>**index.html. Desta forma, conseguimos começar a analisar com maior controle todas as alterações que foram adicionadas durante o desenvolvimento de um projeto.

**+ linha adicionada**

**- linha removida**

**- linha modificada (versão antiga)**

**+ linha modificada (nova versão)**

O sinal de subtração (-) antes da linha indica que ela não está mais presente no arquivo. Já o sinal de adição (+) mostra que é uma linha nova. Alterações são representadas por uma remoção e uma adição de linha.

Após termos visto as alterações e garantido que realmente não há bugs, de que forma poderemos gerar uma versão, por exemplo, 0.1? Como será possível definirmos isto com o Git? Vamos conversar sobre isso adiante.

**TAGS E RELEASES**

Tendo finalizado o projeto de fato, queremos gerar uma versão, indicando ao Git que a partir do penúltimo commit — que acessaremos com **git log -n 2** — seja cravada uma marcação, um checkpoint indicando que se trata da versão 0.1, por exemplo. Em vários outros sistemas de controle de conteúdo, existe o conceito de tag, como quando você cria blogs e possui tags para marcar postagens que pertencem a categorias específicas. No Git, é possível utilizar um **conceito** bastante similar, também denominado **tag**, capaz de marcar um ponto na aplicação que não pode ser modificado, fixo. Assim, após ser lançada, a versão 0.1 nunca deixará de ser a versão 0.1, e quaisquer alterações que forem feitas nela, serão incluídas na versão posterior.

Isso não quer dizer que faremos um código que não será mais editável, apenas que criaremos um marco para onde poderemos ir, e que terá um código correspondente àquele estado. E para criarmos uma tag, informaremos isto ao Git, com **git tag -a, seguido do nome que damos a ela, v0.1.0**, que poderia ser qualquer outro. Além disto, poderemos incluir uma mensagem. O comando completo ficaria, então:

**git tag -a <nome da tag> -m "**Lançando a primeira versão (BETA) da aplicação de cursos**"**

Ao darmos "Enter", geramos uma tag, um marco na nossa aplicação. E se executarmos **git tag**, são exibidos todos estes marcos disponíveis, que no caso por enquanto se resume a apenas um. Já sabemos que é possível fazer push de main, ou de qualquer outra branch, como com **git push local main** e depois **git push local <nome da tag>**v0.1.0 para enviarmos a tag ao servidor.

Para nos lembrarmos de um detalhe, vamos executar **git remote -v**. Estamos utilizando o GitHub, e temos um repositório local denominado origin, que faz menção a ele. Então, atualizaremos nosso código no GitHub com **git push origin main**. Também enviaremos a tag, com **git push origin <nome da tag>**v0.1.0. Ou seja, que resultado gera o envio de uma tag para o GitHub? Gera uma **release**, ou seja, conseguimos baixar um arquivo compactado com o nosso código neste ponto. Mas como será que visualizamos tags por meio do GitHub?

Atualizaremos a página do navegador, que nos informa que temos 17 commits, 1 branch (já que não enviamos as demais para lá), e 1 release, que é a versão pronta para ser lançada ou baixada por qualquer pessoa que queira utilizar em seu sistema. Poderemos ver a nossa mensagem, em qual commit a tag foi gerada, e baixar clicando no ícone com "zip". A release poderá inclusive ser compartilhada com outras pessoas, por meio da URL correspondente, possibilitando o acompanhamento das releases do projeto. Esta é uma feature bem interessante do GitHub!

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

1) Execute o comando git log -p para ver, junto a cada *commit*, as alterações nele realizadas;

2) Execute agora o comando git log --oneline;

3) Execute o comando git diff {hash do commit de merge com lista}..{hash do último commit realizado};

4) Execute alguma (pequena) alteração no **index.html**;

5) Execute o comando git diff e veja esta alteração;

6) Desfaça esta última alteração;

7) Execute o comando git tag -a v0.1.0 para criar uma *tag* no seu código;

8) Execute o comando git push origin v0.1.0 para enviar esta *tag* para o GitHub;

9) Abra a página do repositório do GitHub que você criou e confira a aba de ***Releases***.

Nesta aula, aprendemos:

* Que é possível visualizar quais alterações foram realizadas em cada arquivo, com o comando git diff;
* Que, digitando apenas git diff, vemos as alterações em nossos arquivos que não foram adicionadas para commit (com git add);
* Que é possível comparar as alterações entre duas *branches* com git diff <branch1>..<branch2>
* Que é possível comparar as alterações feitas entre um commit e outro, através do comando git diff <commit1>..<commit2>;
* Que o Git nos possibilita salvar marcos da nossa aplicação, como por exemplo, lançamento de versões, através do git tag;
* Que o comando git tag -a é utilizado para gerar uma nova *tag*;
* As ***Releases*** do GitHub, que são geradas para cada *tag* do Git criada em nosso repositório.

Para darmos uma recapitulada, primeiro entendemos o conceito de repositório, o qual inicializamos na pasta do nosso projeto utilizando **git init**. Depois, vimos que com **git status** conseguimos verificar as alterações já foram realizadas em nosso código, se existe algum arquivo para ser adicionado, ou commitado, algum arquivo que ainda não é trackeado ou monitorado, e assim por diante.

Caso haja alguma modificação a ser adicionada, o fazemos com **git add**. Então, vimos que precisamos gerar commits, que funcionam como sendo um ponto de alteração a ser salva. Conversamos um pouco sobre quando commitar, no entanto, o ponto principal é nunca commitar um código que não funciona.

Aprendemos a verificar nosso histórico de alterações e navegar por elas com **git log --oneline**, geramos tags, uma release que, ao fim, após termos entendido um pouco melhor e trabalhado com GitHub, gera um entregável, uma versão pronta para ser baixada. Vimos também como criar um repositório ao nos cadastrarmos no GitHub, e depois disso, como enviar os dados do nosso projeto local para ele.

Percebemos que com estes repositórios remotos, conseguimos trabalhar em conjunto com outras pessoas da nossa equipe. Além disto, vimos como criar um repositório remoto em nosso próprio computador, e com isto atualmente temos dois deles, identificados a partir do comando **git remote -v**, um denominado local, e outro, de origin.

Aprendemos que não é difícil trabalharmos com mais de um usuário em um mesmo projeto, resolvemos conflitos, trabalhamos com branches, e ainda há muito mais conceito por trás deles, com os quais não trabalhamos no decorrer deste curso, mas o mais importante neste primeiro momento é entender que branches são linhas de desenvolvimento distintas, e mais para a frente poderemos, em cursos futuros, trabalhar as estratégias de quais branches criar, como gerenciá-las, e por aí vai.

Vimos como navegar pelo histórico do nosso código, pelos commits, desfazendo alterações, salvando-as de forma temporária para depois as recuperarmos, com **git stash**, enfim, isso tudo em um único arquivo, o index.html, ou seja, nem focamos ou entramos em detalhes em relação ao código, justamente para focarmos em todo o poder que o Git tem a oferecer.

**Obs.:** no ano de 2020 o GitHub anunciou a mudança do termo “*master*” para “*main*”. Isso ocorreu porque “*master*” é um termo não inclusivo; é uma palavra que é utilizada habitualmente para comunicações em eletrônica, por exemplo: onde se tinha o dispositivo “*master*” ou “mestre” que envia os comandos para o “*slave*” ou “escravo” que responde os processos. Tendo em vista isso, juntamente com o caso de George Floyd e o movimento Black Lives Matter, as empresas de tecnologia foram abandonando esses termos não inclusivos. O GitHub foi uma das primeiras organizações a mostrar apoio a essas mudanças ao anunciar a troca de master para main. Caso você esteja na *branch master* querendo mudar para *main*, pode rodar esses comandos no terminal ou Git Bash:

**git branch -m master main**

**git push -u origin main**