Curso de

Git e Github: Controle e compartilhe seu código

[Vinicius Dias- Carga Horária 6h](https://cursos.alura.com.br/user/cviniciussdias)

Faça esse curso de Builds e Controle de versão e:

* **O que é Git e Github?**
* **Entenda um sistema de controle de versão**
* **Salve e recupere seu código em diferentes versões**
* **Resolva merges e conflitos**
* **Trabalhe com diferentes branches**

Aulas

* 1 [**O que é Git?**](https://cursos.alura.com.br/course/git-github-controle-de-versao/section/8340/tasks)
* **10 / 10 14min**
  + **Introdução**
  + **Para que serve Git?**
  + **Utilidade de um VCS**
  + **Instalando o Git**
  + **Para saber mais: Instalação**
  + **Repositórios**
  + **Para saber mais: Quem é você**
  + **Primeiros passos**
  + **Consolidando o seu conhecimento**
  + **O que aprendemos?**
* **2 [Iniciando os trabalhos](https://cursos.alura.com.br/course/git-github-controle-de-versao/section/8341/tasks)**
* **[9 / 9 17min](https://cursos.alura.com.br/course/git-github-controle-de-versao/section/8341/tasks)**
  + **Salvando alterações**
  + **Para saber mais: git status**
  + **Vendo o histórico**
  + **Primeira configuração do Git**
  + **Para saber mais: git log**
  + **Ignorando arquivos**
  + **Para saber mais: Quando commitar**
  + **Consolidando o seu conhecimento**
  + **O que aprendemos?**
* **3 [Compartilhando o trabalho](https://cursos.alura.com.br/course/git-github-controle-de-versao/section/8342/tasks)**
* **[9 / 9 22min](https://cursos.alura.com.br/course/git-github-controle-de-versao/section/8342/tasks)**
  + **Repositórios remotos**
  + **Servidor Git**
  + **Trabalhando com repositórios remotos**
  + **Sincronizando os dados**
  + **Compartilhamos as alterações**
  + **GitHub**
  + **Para saber mais: GitHub**
  + **Consolidando o seu conhecimento**
  + **O que aprendemos?**
* **4 [Trabalhando em equipe](https://cursos.alura.com.br/course/git-github-controle-de-versao/section/8343/tasks)**
* **[10 / 1 026min](https://cursos.alura.com.br/course/git-github-controle-de-versao/section/8343/tasks)**
  + **Branches**
  + **Para saber mais: Ramificações**
  + **Unindo o trabalho**
  + **Merge de branches**
  + **Atualizando a branch**
  + **Rebase vs Merge**
  + **Resolvendo conflitos**
  + **Para saber mais: Conflitos com rebase**
  + **Consolidando o seu conhecimento**
  + **O que aprendemos?**
* **5 [Manipulando as versões](https://cursos.alura.com.br/course/git-github-controle-de-versao/section/8344/tasks)**
* **[8 / 8 18min](https://cursos.alura.com.br/course/git-github-controle-de-versao/section/8344/tasks)**
  + **Ctrl + Z no Git**
  + **Desfazendo o trabalho**
  + **Guardando para depois**
  + **Salvando temporariamente**
  + **Viajando no tempo**
  + **Checkout**
  + **Consolidando o seu conhecimento**
  + **O que aprendemos?**
* **6 [Gerando entregas](https://cursos.alura.com.br/course/git-github-controle-de-versao/section/8345/tasks)**
* **[7 / 7 14min](https://cursos.alura.com.br/course/git-github-controle-de-versao/section/8345/tasks)**
  + **Vendo as alterações**
  + **Exibição das mudanças com o diff**
  + **Tags e releases**
  + **Tags no GitHub**
  + **Consolidando o seu conhecimento**
  + **O que aprendemos?**
  + **Conclusão**

# 1.O QUE É GIT

***Introdução(VÍDEO)***

Olá pessoal, sejam muito bem vindos à Alura! Meu nome é Vinicius Dias e guiarei vocês neste curso de Git e Github, em nossos primeiros passos com sistemas de controle de versões.

Inicialmente, conversaremos sobre o que é um sistema de controle de versões, quando faz sentido utilizá-lo, quais problemas ele resolve, utilizando o Git para controlar as versões do nosso código.

Faremos bastante modificações, mesmo em um projeto bem simples, só para podermos manter o foco no Git, veremos o que é o Github, como usá-lo, e até gerar uma *release* no fim do curso, uma versão pronta do Github para baixarmos como um arquivo .ZIP, por exemplo.

Veremos pontos interessantes, como trabalhar em equipe, compartilhando nosso código com mais de uma pessoa, criar repositórios remotos em nossos próprios computadores, na rede, e assim por diante.

Entenderemos algumas boas práticas, como gerenciar nossa linha de desenvolvimento por meio de *branches*. Adianto que neste curso não aprenderemos tudo que existe sobre o Git, mas sim o básico necessário para começarmos a trabalhar controlando o código e as suas versões.

Espero que tenha bastante proveito! Caso haja alguma dúvida, não hesite em abrir uma dúvida no [fórum](https://cursos.alura.com.br/forum/curso-git-github-controle-de-versao/todos/), pessoalmente sempre tento responder, e quando não consigo, a nossa comunidade é bastante solícita.

Te espero adiante!

***Para que serve o GIT(VIDEO)***

Antes de começarmos a trabalhar de fato no Git, vamos conversar sobre pra quê ele serve, o que é este tal de sistema de controle de versões. Imaginemos que você, uma pessoa que desenvolve, tem um projeto em seu computador.

Obviamente, você fará alterações no código, porém, você trabalha em uma equipe, portanto, além de você existem outras pessoas que estão desenvolvendo neste mesmo projeto. Cada um possui uma cópia do projeto nas suas respectivas máquinas locais, para que seja possível fazer alterações, e ver tudo funcionando direitinho antes de enviar para outra pessoa.

No entanto, como todos estão trabalhando no mesmo projeto, essas pessoas precisam entender que modificações estão sendo feitas em paralelo. Então, quando você realizar alguma alteração, por exemplo, é necessário notificar o restante da equipe.

Porém, quando você faz esta alteração e tenta enviar aos outros, seja por meio de um pendrive, por e-mail, salvando no Dropbox, pode acontecer deles já terem feito uma alteração também, ou terem feito outras modificações anteriormente. **De que forma podemos controlar essas versões diferentes de um mesmo código?**

Não é difícil de imaginar que deste modo o trabalho fica bastante confuso, certo? Então, uma das soluções possíveis é separar um servidor específico para o envio das alterações dos arquivos. Todos da equipe terão acesso a este servidor.

Neste servidor, deve haver alguma ferramenta capaz de identificar que a versão enviada não é a mais recente, e portanto não deixe o arquivo ser enviado. Isto é, antes do envio de uma alteração, este colega de trabalho precisará baixar as alterações que já foram enviadas, para que só então consiga enviar a versão atualizada por ele.

Isso é o chamado **controle de versão**, pois se temos diferentes versões do código precisaremos de um sistema que controle essas versões. E é isso que o Git fará para nós. Este não é o único sistema de controle de versão que existe. Outras alternativas são:

* CVS
* SVN
* Mercurial
* GIT

O Git é o mais utilizado entre eles atualmente por conta de algumas características vantajosas, como permitir uma cópia do projeto, um **repositório do projeto** em sua máquina, para que se possa trabalhar em cima dela e então enviá-lo para outro repositório, o que se denomina **repositórios distribuídos**.

Isso permite o trabalho de modo offline, antes da comunicação com outro servidor para que o envio de versões, e assim por diante. Existem várias outras diferenças entre estas alternativas, e você as entenderá melhor no decorrer do curso.

Agora que já entendemos a motivação para utilizar o Git, e em que cenário faz sentido usarmos um controle de versões, vamos instalá-lo e ver como ele funciona?

***UTILIDADE DE UM VSC***

Agora que nós já entendemos para que serve um **VCS** (***V****ersion* ***C****ontrol* ***S****ystem*), ou sistema de controle de versões, podemos dar continuidade com este treinamento, mas só para garantir...

Quais das afirmações a seguir sobre o Git estão corretas?

* Alternativa correta
* Nos deixa organizar o trabalho em equipe, mantendo as alterações nos arquivos em um servidor específico para isso
* Alternativa correta! O Git permite que a gente armazene as modificações feitas em cada arquivo em um servidor próprio para isso. Toda a gestão de alterações é feita pelo Git e nós só precisamos nos preocupar em criar código que funciona, e não em quem alterou o que antes.
* Alternativa correta
* Permite armazenamento e acesso a um histórico de modificações
* Alternativa correta! Cada alteração que você faz fica gravada em um histórico, podendo ser visualizada e restaurada a qualquer instante.

***INSTALANDO O GIT (VÍDEO)***

Vamos instalar o Git para começarmos a controlar as versões dos nossos códigos. Para isso, basta pesquisarmos por "git" na internet, e clicarmos no que provavelmente será o primeiro resultado, o [site oficial](https://git-scm.com/). Nele, temos algumas informações sobre o que é o Git, mas por ora clicaremos no botão de download, neste caso, "Download 2.21.0 for Windows".

Caso você esteja utilizando Linux, algumas distribuições já vêm com o Git instalado, então é só abrir o Terminal e digitar "git" para verificar isto. Se ele não estiver instalado, o gerenciador de pacotes da sua distribuição, por exemplo o APT para Ubuntu ou derivados de Debian, o DNF para Fedora, com certeza possuem uma versão do Git, basta executar a instalação por meio da linha de comando.

Feito o download, executaremos o arquivo, e durante a instalação, existem alguns pontos interessantes que o Git nos traz, sendo o Git Bash um deles, que é uma forma de digitar comandos que passaremos a utilizar no Windows. Não que não seja possível optar pelo Prompt de comando padrão do Windows. A diferença é que o Git Bash fornece comandos com os quais quem desenvolve em Linux já está acostumado a usar, como o ls para mostrar arquivos e pastas existentes no diretório atual.

Manteremos a instalação padrão e clicaremos em "Next". Em "Adjusting your PATH environment", é possível definir se iremos usar apenas o Git Bash, ou então o Git de qualquer outra interface de linha de comando, portanto também deixaremos marcada a opção padrão. Caso você queira ler com maior atenção cada uma das etapas de instalação, não tem problema, mas para o que faremos neste curso, as opções padrões são o suficiente.

Finalizada a instalação, desmarcaremos o box de "View Release Notes" e marcaremos "Launch Git Bash", para que se inicie a execução do Git Bash. A aparência é muito similar à de um Terminal, com a diferença de que no Prompt de comando digitaríamos, por exemplo, dir para que fossem exibidas as pastas existentes, sendo que no Linux e no Mac utilizamos ls.

Com isso, temos um novo Terminal instalado, além do próprio Git, e para garantirmos isso, podemos executar git --version, ao que será retornado git version 2.21.0.windows.1, neste caso. Não se atente à versão neste momento, caso você esteja com uma versão mais recente, não tem problema também.

Poderemos começar a controlar um código, um projeto. Mas o que desenvolveremos durante este curso? Vamos conversar um pouco melhor sobre isso a seguir!

***Para saber mais: Instalação***

Embora tenhamos passado por diversas etapas durante o vídeo, vimos que a instalação do Git no Windows é simples e ainda nos traz algumas ferramentas para que o nosso ambiente de desenvolvimento seja um pouco mais parecido com o de outras plataformas.

Instalação no Linux

A instalação do Git no Linux é muito simples e em algumas distribuições nem é necessária, pois ele já vem instalado. Caso não seja o caso de sua distribuição, confira [aqui](https://git-scm.com/download/linux) o comando necessário para instalá-lo.

Instalação no macOS

A instalação no macOS também é muito simples. Basta seguir as instruções deste link: <https://git-scm.com/download/mac>.

***Repositórios(VÍDEO)***

O projeto que desenvolveremos durante o curso será uma página HTML de cursos na Alura, simples, sem estilos, para que possamos de fato focar no Git, e não precisemos adentrar em detalhes de nenhuma linguagem de programação. Queremos que um repositório do Git seja inicializado, e para tal usamos o comando git init.

Assim, todas as alterações que forem realizadas no arquivo localizado dentro deste repositório poderão ser mostradas pelo Git, com indicações do que foi modificado, quem modificou, quando, e por aí vai. Ainda não entraremos em detalhes, mas reparem que, a partir do momento em que digitamos git init, uma informação foi acrescentada no final do Git Bash ((master)).

Caso você esteja utilizando o Terminal padrão do Linux ou do Mac, e esta opção não aparecer, não tem problema nenhum, não significa que não esteja funcionando, é simplesmente uma informação a mais, trazida pelo Git Bash. Mas como saberemos que o comando git init está "enxergando" a pasta e entendendo as modificações?

Um comando que mostra o estado do nosso repositório, ou seja, quais arquivos foram alterados, ou não, é o git status. Ao ser rodado, neste caso, por exemplo, ele nos informa que está sendo rodado no ramo, ou branch master (On branch master), e que não possui nenhum commit (No commits yet).

Além disso, é indicado que há arquivos não monitorados (Untracked files) em nosso projeto, justamente index.html, que é o único arquivo que temos por enquanto. É indicado que utilizemos o comando git add junto ao nome do arquivo para que possamos inclui-lo no que se quer "commitar".

Antes de entrarmos em maiores detalhes, e entendermos o que é um commit, um branch, já temos um conceito de **repositório**, e informamos ao Git que esta pasta em específico é um repositório do Git, então, tudo que estiver dentro desta pasta, a menos que informemos o contrário, será monitorado e analisado pelo Git e, se for o caso, salvar as alterações, ou não.

Como o Git mesmo nos informa, o arquivo que usaremos ainda não está sendo monitorado, então, poderemos utilizar o comando git add? Veremos tudo isso adiante!

***Para saber mais: Quem é você***

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.

No vídeo eu não fiz isso pois o git já estava configurado na máquina, mas para você fazer isso na sua, 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"COPIAR CÓDIGO

***Primeiros passos***

Já entendemos o motivo para utilizar o Git. Começamos também a entender como o Git funciona. Sabemos que o Git faz a gestão de repositórios, e cada pessoa na equipe pode ter o seu repositório.

Como fazemos para o Git passar a enxergar determinada pasta como um repositório e a observar as mudanças em seus arquivos?

* Alternativa correta
* Através do comando git init
* Alternativa correta! O git init inicializa um repositório no diretório em que o comando for executado. A partir deste comando, o Git poderá gerenciar as modificações realizadas nos arquivos.

***Consolidando o seu conhecimento***

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 (onde preferir) e dentro dela salve o arquivo index.html, com o seguinte conteúdo:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Cursos da Alura</title>

</head>

<body>

<ul>

<li>Vagrant</li>

<li>Docker</li>

<li>Ansible</li>

<li>Integração Continua</li>

</ul>

</body>

</html>

COPIAR CÓDIGO

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

OBS: Caso o caminho da sua pasta possua espaços, é preciso colocá-lo entre aspas. Exemplo: cd 'Documents/Curso Git e GitHub'

3) Na pasta do projeto, execute o comando git init para inicializar um repositório Git;

4) Execute o comando git status para garantir que você está em um repositório Git e que o arquivo index.html é reconhecido. Leia com calma a saída deste comando. Nem tudo estará claro ainda, mas durante o treinamento ficará!

Opinião do instrutor

Continue com os seus estudos, e se houver dúvidas, não hesite em recorrer ao nosso fórum!

***O que aprendemos?***

Nesta aula, aprendemos:

* O que são (e para que servem) **sistemas de controle de versões** e como eles podem ajudar o nosso fluxo de desenvolvimento
  + Nos ajudam a manter um histórico de alterações;
  + Nos ajudam a ter controle sobre cada alteração no código;
  + Nos ajudam para que uma alteração de determinada pessoa não influencie na alteração realizada por outra;
  + Etc.
* O que é o Git e como instalá-lo
* Que com o comando git init nós conseguimos criar um repositório Git;
* Como analisar o estado do nosso repositório através do comando git status.

# 2 Iniciando os trabalhos

* + **Salvando alterações(VÍDEO)**

Já criamos nosso primeiro repositório, então, se executarmos git status dentro da pasta em que trabalhamos anteriormente, veremos que trata-se de um repositório Git, porém, seu arquivo ainda não está sendo monitorado, ou seja, ele não está salvo no histórico do Git. Para salvarmos uma alteração, ou um arquivo nele, precisaremos que ele monitore o arquivo, e suas mudanças.

Como o arquivo index.html ainda não está sendo monitorado, e nunca foi editado e salvo pelo Git, utilizaremos o comando git add index.html. Se tivéssemos vários arquivos, não precisaríamos colocar seus nomes um a um, bastando git add . para que todos os arquivos da pasta atual sejam monitorados.

Com isso, se rodarmos git status, desta vez teremos um retorno diferente, incluindo Changes to committed, isto é, "mudanças a serem commitadas", ou salvas, enviadas. Inclusive, é indicado que poderíamos executar git rm para remover o arquivo e para que o mesmo deixe de ser monitorado, o que não queremos fazer.

Queremos salvar as alterações, e o que poderemos entender como sendo um *check point* para indicar que houve mudança, seria o commit, que precisa ter modificações, que já adicionamos, mas também precisa ter uma mensagem, o que criaremos agora. Por já termos adicionado as modificações a serem enviadas, executaremos simplesmente git commit -m "Criando arquivo index.html com lista de cursos", em que o parâmetro -m serve para passarmos uma mensagem de commit, que será incluído entre aspas.

A boa prática pede para colocarmos mensagens descritivas, evitando que fiquem muito grandes.

Quando dermos "Enter", o Git Bash nos informa que este é o root-commit, commit base dentro de um master, e exibe a mensagem que configuramos. Também é mostrado quais foram as alterações, no caso, apenas 1, com 15 inserções (linhas). Se executarmos git status novamente, teremos que não há nada a ser commitado, entretanto ele não mostra mais que não há commits ainda.

Vamos fazer uma modificação simples, como colocar um acento agudo em "Contínua" de <li>Integração Contínua</li>. Salvaremos e reexecutaremos git status, e obteremos a indicação de que há uma modificação não salva. Para isso, executaremos git add index.html. Com outro git status, teremos a mensagem de que há mudanças a serem commitadas. Usaremos clear para limparmos a tela, e então git commit -m "Acento adicionado no curso de Integração Contínua" e pressionaremos "Enter".

Conseguiremos acessar uma espécie de lista de commits realizados de forma muito simples, por meio de um comando que veremos a seguir.

* + **Para saber mais: 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>.

Acredite, embora pareça confuso agora, durante o treinamento tudo fará muito mais sentido! :-D

* + **Vendo o histórico(VIDEO)**

Anteriormente, ficamos com a dúvida: como poderemos verificar o histórico de alterações, cada mensagem de commits feitos, o andamento do nosso projeto? 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. A informação seguinte se refere ao branch, ou "ramo" em que o commit se encontra. Neste caso, verificamos que há HEAD e master. 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 master.

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.

Por enquanto, modificaremos as configurações para este único projeto, ou seja, as configurações definidas por meio deste comando só serão válidas para este repositório. Como anteriormente só foi exibido meu e-mail, configuraremos o nome, com git config --local user.name "Vinicius Dias", após o qual pressionaremos "Enter".

Poderemos visualizar as configurações salvas por meio de git config user.name, ou git config user.email. Então, os commits que fizermos a partir daqui terão este nome. Mas será que existe alguma alternativa ao git log?

Sim, existem várias! Uma das mais comuns nos permite visualizar todos os commits, sendo que cada uma ocupa uma única linha: git log --oneline. E se em vez de menos informações quisermos mais, como as alterações do commit, usaremos git log -p. O formato em que elas são exibidas conta com a versão anterior em vermelho, e a versão modificada logo abaixo, em verde.

Existe uma infinidade de formatos que podemos usar como filtros para mostrar nosso histórico, e em [git log cheatsheet](http://devhints.io/git-log) há vários delas. Como exemplo, testaremos git log --pretty="format:%H", comando que nos traz apenas o hash. O comando git log --pretty="format:%h %s", por sua vez, traz o hash resumido seguido pela mensagem do commit. Assim, podemos gerar o histórico da nossa aplicação em formatos personalizados.

Aqui no curso estamos usando o VS Code — é possível utilizar qualquer editor de sua preferência —, mas imaginemos que estejamos usando uma IDE que cria uma pasta contendo configurações, os quais não queremos que o Git fique monitorando. De que forma podemos informá-lo disto? Veremos a seguir!

* + **Primeira configuração do Git**

Para que o Git saiba quem está realizando as alterações, ele precisa de algumas configurações. Na primeira vez que você tentar realizar um *commit* em uma máquina, ele pedirá que você o configure.

Como podemos definir o nome da pessoa que executa *commits* no repositório local atual?

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

**Alternativa correta! Assim todos os *commits* executados neste repositório serão atribuídos à pessoa com nome Nome da pessoa. Para mais detalhes e outras configurações possíveis (até algumas mais avançadas), você pode conferir este link:** [**https://git-scm.com/book/en/v2/Customizing-Git-Git-Configuration**](https://git-scm.com/book/en/v2/Customizing-Git-Git-Configuration)**.**

* + **Para saber mais: git log**

Conforme vimos no último vídeo, podemos visualizar o histórico de alterações em nosso projeto de forma muito fácil, através do comando git log.

Apesar de ser fácil, este comando é muito poderoso. Execute git log --help e veja algumas das opções possíveis. Para alguns exemplos mais fáceis de entender, você pode pesquisar sobre git log ou dar uma olhada neste link: <https://devhints.io/git-log>.

* + **Ignorando arquivos(VIDEO)**

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.

Já entendemos o que é um repositório e como funciona seu conceito, inclusive transformamos nossa pasta em um repositório Git. Além disso, aprendemos a visualizar o seu status, como adicionar e salvar arquivos nele, visualizar as alterações feitas no projeto, e deixar de monitorar determinados arquivos ou pastas.

Conseguimos começar a trabalhar de forma bem interessante com o controle de versões. Mas como será que passamos a trabalhar em equipe, compartilhando o projeto usando um repositório Git?

* + **Para saber mais: 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

* + **Consolidando o seu conhecimento**

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**.

* + **O que aprendemos?**

Nesta aula, aprendemos:

* Que um commit é a forma de salvar um estado ou versão do nosso código;
* Como adicionar arquivos para serem *commitados* com git add;
* Como *commitar* arquivos, utilizando o comando git commit;
* Como verificar o histórico de *commits*, através do git log e algumas de suas opções:
  + git log --oneline
  + git log -p
  + git log --pretty="parametros de formatação"
* Como fazer o Git não monitorar arquivos, através do .gitignore
* Que não devemos realizar commit, ou seja, salvar um estado, da nossa aplicação que não esteja funcionando.

# 3 Compartilhando o trabalho

* + **Repositórios remotos(VÍDEO)**

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 ../vinicius.

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 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, é indicado 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 /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.

* + **Servidor Git**

No último vídeo, nós trabalhamos bastante. Nossa primeira tarefa foi criar um novo repositório, que será utilizado como o nosso "servidor" Git, ou seja, todos os membros da equipe o acessarão para compartilhar suas mudanças.

Como fizemos para definir um repositório Git neste caso?

* Alternativa correta
* git init --bare
* Alternativa correta! Com este comando nós criamos um repositório que não terá a *working tree*, ou seja, não conterá uma cópia dos nossos arquivos. Como o repositório servirá apenas como servidor, para que outros membros da equipe sincronizem seus trabalhos, poupamos espaço de armazenamento desta forma.
  + **Trabalhando com repositórios remotos**

Antes de sincronizar as nossas mudanças no código com algum repositório remoto, precisamos adicioná-lo ao nosso repositório local.

Como adicionamos esta ligação entre os repositórios?

git remote add nome-repositorio caminho/para/o/repositorio

Alternativa correta! Desta forma teremos um link do nosso repositório local com o repositório remoto, que chamamos de nome-repositorio, que está armazenado em caminho/para/o/repositorio.

**Sincronizando os dados(VIDEO)**

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 master, 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 master.

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 master para trazermos os dados. Ainda falaremos melhor sobre branches, no entanto sabemos que estamos trabalhando com master 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 master. 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 master. 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.

E se não quisermos criar um servidor, ou se não pudermos criar um servidor local, muito menos compartilhar uma pasta no computador? E se quisermos colocar o conteúdo em algum servidor online? Será que existe um serviço que nos permita um repositório Git online?

* + **Compartilhamos as alterações**

Além de adicionar repositórios remotos para sincronizar os dados, vimos que o git clone traz um repositório remoto para o nosso computador, criando um repositório local.

Ao alterar os códigos em nosso repositório local, como enviar as alterações para o repositório remoto?

* Alternativa correta
* git push [repositorio] master
* Alternativa correta! Desta forma, nós enviamos as alterações em nosso *branch* master (falaremos mais sobre *branches* já já) para o repositório remoto. Basta substituir [repositorio] pelo nome que demos ao repositório ao adicioná-lo. Já para trazer os dados que estiverem no repositório remoto, podemos utilizar o git pull [repositorio] master.
  + **GitHub(VIDEO)**

Anteriormente, fizemos a sincronização do conteúdo dos arquivos da Ana e do Vinicius, no entanto, surgiu um questionamento: precisaremos realmente ter um servidor na nossa rede, ou uma pasta compartilhada com nossos arquivos? Será que existem alternativas para criarmos servidores remoto gratuitamente, compartilhável pela internet?

Se você já sabe onde quero chegar, você provavelmente já ligou um ponto a outro; existem vários serviços do tipo, mas aqui, trataremos do **GitHub** que, dentre outras características, é um serviço que fornece a possibilidade de se criar repositórios Git. Acessaremos o [site oficial](http://github.com/) que, diga-se de passagem, é da Microsoft.

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 -u origin master, cujo -u define que, sempre que usarmos git push e estivermos na master, o envio seja feito para origin. Ou seja, a partir de então poderemos executar simplesmente git push.

**Atenção:** 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 master.

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.

Lidamos, assim, com uma interface bem interessante para o gerenciamento do nosso projeto. O GitHub é uma plataforma muito poderosa, e faz muito mais do que simplesmente disponibilizar repositórios remotos: conseguimos configurar colaboradores no projeto, para que outros usuários de GitHub possam fazer commits diretamente, entre outras vantagens. Neste curso não entraremos em detalhes, continuaremos utilizando nosso repositório local, mas já entendemos como enviar um dado para o GitHub.

Continuando, 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!

* + **Para saber mais: GitHub**

No último vídeo, nós conhecemos (bem por alto) um dos serviços mais famosos e utilizados por pessoas que desenvolvem software: o GitHub.

Com o GitHub, podemos ter repositórios remotos públicos e privados gratuitos para armazenar e compartilhar o código dos nossos projetos.

O GitHub fornece muitas outras funcionalidades bem legais. Explore-as, brinque com elas, e em outros cursos aqui na Alura nós vamos falar mais sobre esse assunto.

* + **Consolidando o seu conhecimento**

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.

**O QUE APRENDEMOS?**

Nesta aula, aprendemos:

* O que são repositórios remotos;
* Como criar um repositório Git sem uma cópia dos arquivos (com --bare) para ser utilizado como servidor;
* Como adicionar links para os repositórios remotos, com o comando git remote add;
* Como baixar um repositório pela primeira vez, clonando-o com o comando git clone;
* Como enviar as nossas alterações para um repositório remoto, com git push;
* Como atualizar o nosso repositório com os dados no repositório remoto, utilizando git pull;
* O que é e para que serve o **GitHub**;
* Como criar um repositório no **GitHub**;
* Como adicionar um repositório do **GitHub** como repositório remoto.

# 4 Trabalhando em equipe

* + **Branches(VIDEO)**

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 master 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, master, 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 titulo, que criará este branch, embora tenhamos que mudar para ela manualmente, com git checkout 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 master. 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 master.

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 master 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 passar para ele: git checkout -b 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 master. 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 master?

* + **Para saber mais: 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.

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! ;-)

* + **Unindo o trabalho(VIDEO)**

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 master, e trabalhamos na primeira. Porém, no repositório que se encontra na pasta "servidor", só temos a branch master, 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 master. Na ferramenta [Visualizing Git](https://git-school.github.io/visualizing-git/) criaremos a branch titulo e passaremos a trabalhar nela, com git checkout -b 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 master e, a partir daí, corrigir o bug.

Utilizaremos git checkout master, 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 master?

Ou seja, queremos unificar estas duas linhas, portanto usaremos o comando git merge 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 master, 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 master, 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 master?

Como faremos com que ele mova estas branches e atualize a master apenas com os dois commits, sem criar um de *merge*? Veremos isto a seguir!

* + **Merge de branches**

Agora que entendemos como separar o desenvolvimento em linhas ("ramos") diferentes, é hora de trazer estas modificações para a master, que é a nossa *branch* "padrão".

Como podemos fazer o merge da *branch* titulo para a *branch* master?

git checkout master e git merge titulo

Alternativa correta! Desta forma colocaremos o HEAD na *branch* master, ou seja, faremos com que o nosso código esteja no estado que o deixamos com o último commit na master. Depois, uniremos o trabalho da *branch* titulo com a *branch* atual (master).

* + **Atualizando a branch(VIDEO)**

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 master com os commits da branch titulo, de modo a termos cada commit específico na linha de desenvolvimento master.

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

Para isto, estando na master, queremos basear esta branch em titulo, assim, executaremos git rebase titulo, e o Git pegará os commits na branch titulo, atualizando master, 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 master 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 master, 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 master, logados como Vinicius. Tudo está atualizado! Podemos, então, nos logar como Ana, usar git checkout master e git pull local master para atualizar os dados também.

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 master. 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?

* + **Rebase vs Merge**

Já sabemos como trazer o trabalho de outra *branch* e unir com a *branch* atual. Conhecemos duas formas de fazer isso: merge e rebase.

Neste cenário, qual 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.

Alternativa correta! 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>.

* + **Resolvendo conflitos(VIDEO)**

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.

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 master. 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 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 master.

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 master). Agora, sim, será feito o *merge* da master 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 master. 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 master.

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.

Existem estratégias bem específicas de quando e como criar uma branch, e podem surgir dúvidas quanto à criação de uma branch para cada funcionalidade ou feature nova. Sem entrar em detalhes — por não ser o foco deste curso — sabemos que branches são linhas de desenvolvimento, e aprendemos a lidar com elas.

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

* + **Para saber mais: 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.

Boa sorte! ;-)

* + **Consolidando o seu conhecimento**

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* master 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 master para voltar à linha de desenvolvimento master;

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 master para voltar à linha de desenvolvimento master;

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* master;

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 master para voltar à linha de desenvolvimento master;

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* master;

26) Execute o comando git push local master 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 master 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 master para voltar à linha de desenvolvimento master;
* 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 master 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 master. Veja a falha;

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

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

* + **O que aprendemos?**

Nesta aula, aprendemos:

* Que uma *branch* (ou ramo) é uma linha de *commits* separada, e que pode ser utilizada para desenvolver funcionalidades independentes;
* Que com *branches* separados, podemos evitar que o código de uma funcionalidade interfira em outra;
* Como trazer o trabalho realizado em uma *branch* para outra *branch*, como por exemplo, o master, através do comando git merge;
* Que o git merge gera um novo *commit*, informando que houve uma mescla entre duas *branches*;
* Como trazer os *commits* de uma *branch* para outra, com o git rebase
* Que o git rebase não gera um *commit* de merge, simplificando o nosso *log*;
* Como os conflitos são apresentados pelo Git;
* Como resolver os conflitos e manter apenas as alterações desejadas com o Git.

# 5 Manipulando as versões

* + **Ctrl + Z no Git(VIDEO)**

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, e antes de continuarmos com este aprendizado — no repositório da Ana não fizemos aquela configuração para definir que o nome de quem faz o commit é o dela, para mantermos o histórico correto.

Vamos criar a configuração com git config --local user.name "Ana", a partir do qual todos os commits que forem feitos neste repositório irão pertencer a ela. Continuando, 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 -- seguido do 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 seguido do 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 -- index.html, 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.

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!

* + **Desfazendo o trabalho**

No último vídeo, nós aprendemos a desfazer alterações das quais não vamos precisar mais.

Quais os comandos, respectivamente, desfazem alterações antes de adicioná-las (1); depois de adicioná-las, mas antes de *commitá-las* (2); e após realizar o *commit* (3)?

1 - git checkout

2 - git reset

3 - git revert

Alternativa correta! 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.

* + **Guardando para depois(VIDEO)**

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!

* + **Salvando temporariamente**

Vimos como podemos utilizar git stash para armazenar temporariamente algumas de nossas alterações.

Em que momento o stash parece útil?

Quando precisamos parar o desenvolvimento de algo no meio para trabalhar em outra coisa

Alternativa correta! Quando precisamos pausar o desenvolvimento de alguma funcionalidade, ou correção, 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.

* + **Viajando no tempo(VIDEO)**

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 muda o estado da aplicação, seja desfazendo alterações, 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 master, 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 commit 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 master 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 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 master, 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.

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?

* + **Checkout**

Já utilizamos em mais de uma ocasião o comando git checkout.

Resumidamente, para que serve o comando git checkout?

* Alternativa correta
* Para deixar o nosso código em determinado estado
* Alternativa correta! 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).
  + **Consolidando o seu conhecimento**

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 master 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 master para voltar à linha principal de desenvolvimento.

* + **O que aprendemos?**

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>.

# 6 Gerando entregas

* + **Vendo as alterações(VIDEO)**

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?

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, 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 acresentando 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 index.html, e com git status conferiremos que ele está pronto para ser adicionado ao commit. Vamos desfazer as alterações com git checkout -- index.html. Desta forma, conseguimos começar a analisar com maior controle todas as alterações que foram adicionadas durante o desenvolvimento de um projeto.

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.

* + **Exibição das mudanças com o diff**

Com o comando git diff, nós vimos que é possível visualizarmos as mudanças realizadas em determinado código. Podemos ver as diferenças entre *commits*, *branches*, etc.

Como o git diff exibe as mudanças no código?

* Alternativa correta

+ linha adicionada

- linha removida

- linha modificada (versão antiga)

* + linha modificada (nova versão)
* Alternativa correta! 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.
  + **Tags e releases(VIDEO)**

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 trata-se 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 v0.1.0 -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 master, ou de qualquer outra branch, como com git push local master e depois git push local 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 master. Também enviaremos a tag, com git push origin v0.1.0. 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!

* + **Tags no GitHub**

Vimos no último vídeo como gerar versões, ou marcos interessantes em nosso sistema de controle de versões.

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

Alternativa correta! O GitHub nos dá a possibilidade de baixar um arquivo compactado contendo o código no estado em que a *tag* foi gerada.

* + **Consolidando o seu conhecimento**

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***.

* + **O que aprendemos?**

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.
  + **Conclusão(VIDEO)**

Boas vindas ao final do curso de Git e GitHub, uma introdução a tudo que a ferramenta de controle de versões pode nos oferecer, ainda há muito o que aprender.

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.

Mais uma vez, isso não é tudo, outros cursos virão para nos aprofundarmos um pouco mais, lembrando que este conteúdo é o suficiente para nosso dia a dia. Espero que tenham tirado bastante proveito, caso tenha ficado alguma dúvida, não deixe de abrir um tópico no [fórum](https://cursos.alura.com.br/forum/curso-git-github-controle-de-versao/todos/). Espero te ver em cursos futuros aqui na Alura! Forte abraço e bons estudos!