**Git e Github: controle e compartilhe seu código**

**Introdução**

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 Git?**

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?

**INSTALANDO O GIT**

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!

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**

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!

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"

**CONSOLIDANDO CONHECIMENTOS**

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!

**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.

**02.INICIANDO OS TRABALHOS**

**SALVANDO ALTERAÇÕES**

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:**

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**

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" \t "_blank) 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!

## Mais opções

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>.

## 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. :-D

**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.

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 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.

**03.COMPARTILHANDO O TRABALHO**