**GIT: PRIMEIROS PASSOS**

**O QUE É GIT?**

Git é um sistema de controle de versão de arquivos. Através deles podemos desenvolver projetos na qual diversas pessoas podem contribuir simultaneamente no mesmo, editando e criando novos arquivos e permitindo que os mesmos possam existir sem o risco de suas alterações serem sobrescritas.

Se não houver um sistema de versão, imagine o caos entre duas pessoas abrindo o mesmo arquivo ao mesmo tempo. Uma das aplicações do git é justamente essa, permitir que um arquivo possa ser editado ao mesmo tempo por pessoas diferentes. Por mais complexo que isso seja, ele tenta manter tudo em ordem para evitar problemas para nós desenvolvedores.

Outro fator importante do git (e essa é um dos seus diferenciais em relação ao svn – caso vc o conheça) é a possibilidade de criar, a qualquer momento, vários snapshots do seu projeto, ou como chamamos mais “nerdmenete”, branch. Suponha que o seu projeto seja um site html, e você deseja criar uma nova seção no seu código HTML, mas naquele momento você não deseja que estas alterações estejam disponíveis para mais ninguém, só para você. Isso é, você quer alterar o projeto (incluindo vários arquivos nele), mas ainda não quer que isso seja tratado como “oficial” para outras pessoas, então vc cria um branch (como se fosse uma cópia espelho) e então trabalha apenas nesse branch, até acertar todos os detalhes dele. Após isso, você pode fazer um merge de volta do seu branch até o projeto original. Veja bem, se tudo isso que você leu só ajudou a te confundir mais – respire fundo – e siga em frente. Com exemplos tudo fica melhor.

**O QUE É GIT (UMA SEGUNDA EXPLICAÇÃO)?**

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.

**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.

**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](mailto:seu@email.aqui)”

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.

**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".

**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.

**VENDO 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) 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!

**PARA SABER MAIS: GIT LOG**

**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**.

**REPOSITÓRIOS REMOTOS**

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.

**AAAAAAA**