# Controle de versão, Git, Github: o que são?

## Introdução ao controle de versões com Git

Quando iniciamos um novo projeto, uma das primeiras perguntas que surge é: onde os arquivos desse projeto ficarão armazenados?. É necessário, principalmente quando trabalhamos em conjunto com outros desenvolvedores, que o projeto esteja armazenado em um local que esteja acessível a todos os membros da equipe, com as respectivas permissões para que os mesmos sejam alterados, para que novos arquivos sejam criados e que arquivos existentes possam ser excluídos.

Tendo em vista esse cenário, podemos definir que qualquer computador, numa rede onde todos os membros da equipe tenham acesso, é candidato a "local do projeto". Mas, e se ocorrer algum problema com esse computador como, por exemplo, alguma falha em seu hardware?

Então podemos pensar em armazenar os arquivos do projeto em um servidor, com redundância de dispositivos de armazenamento e backups constantes. Uma pasta compartilhada na rede em uma máquina mais segura do que uma estação de trabalho torna-se uma opção melhor. Agora temos um local propício para que o projeto seja armazenado com segurança e acesso para toda a equipe, inclusive com backups constantes que permitem que recuperemos arquivos excluídos acidentalmente. Podemos chamar esse local de **repositório de código**.

Apesar dessa segurança adicional, ficamos um pouco restritos quanto a recuperar arquivos excluídos, não temos como desfazer alterações no nosso projeto seletivamente e também não temos informações suficientes sobre o projeto para saber exatamente qual versão desse arquivo eu devo recuperar do backup?. Podemos criar um histórico (log) textual e adicioná-lo como parte do projeto. Agora perceba que ficamos com um ambiente com requisitos demais para um simples "local de armazenamento de projeto".

Pensando nesse problema foram criadas ferramentas de **controle de versão** e esse curso apresentará uma das mais utilizadas hoje em dia, o **Git**. O Git é utilizado em muitos projetos de código aberto no mundo todo. Um dos maiores e mais importantes é o kernel do Linux. É possível, porém, utilizá-lo para projetos privados com todas as restrições e permissões de acesso necessárias.

Uma grande vantagem do Git é a existência de ferramentas comerciais de hospedagem de código na Web. A maior provedora de hospedagem de repositórios Git, abertos ou privados, hoje é o Github (<http://www.github.com/>). Ele permite criar gratuitamente repositórios abertos.

Para que seja possível utilizar o Git teremos que instalá-lo. O Git é uma ferramenta baseada em linha de comando, ou seja, realizamos as operações de controle dos arquivos pelo prompt de comando. Se você é adepto de ferramentas gráficas não se preocupe, existem algumas opções para facilitar seu trabalho e vamos conhecê-las no decorrer do curso.

Iniciaremos o curso utilizando o Git no prompt de comando, para que seja possível demonstrar suas funcionalidades que, de certa maneira, são mascaradas em botões e opções de menus nas ferramentas gráficas. O uso do Git no prompt de comando será muito importante para detalhar suas capacidades.

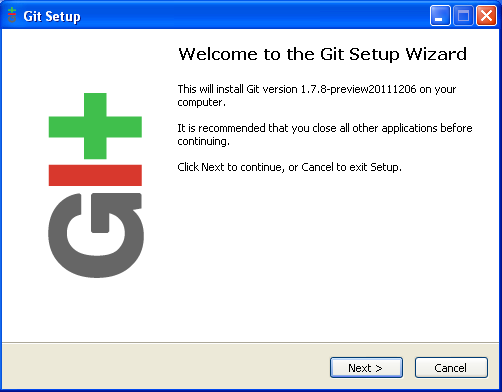
### Download e instalação do Git

A instalação do Git é diferente para cada sistema operacional. A seguir, mostraremos como todo o processo é feito em cada um deles. [Caso você já tenha o Git instalado, clique aqui para ir direto ao uso do Git.](https://cursos.alura.com.br/course/git/task/444#clone)

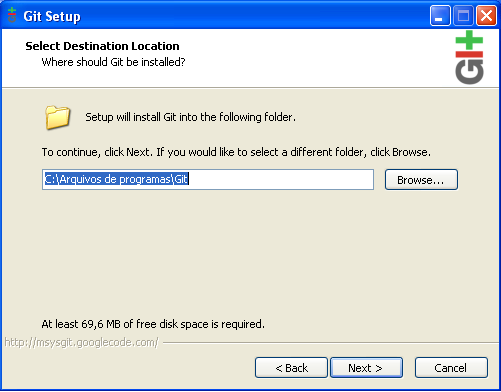
#### Windows

Para instalar o Git, iniciaremos acessando a página <http://msysgit.github.io/> ou <https://git-scm.com/download/win> e baixando o instalador do msysgit. O instalador da versão mais recente, no momento da escrita, é o Git-1.9.5-preview20141217.exe. Uma vez baixado, basta executar o programa e seguir as instruções (caso haja uma tela de instalação que não esteja aqui, pode aceitar com as opções padrão).

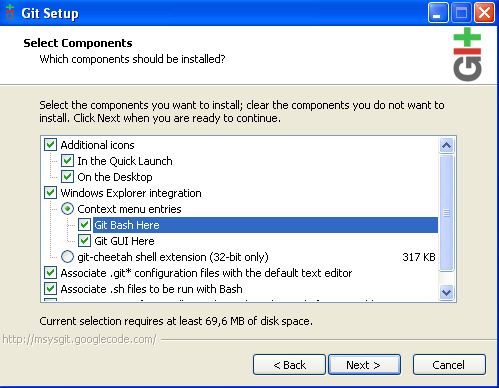
Na primeira tela de boas vindas, apenas clique em Next:



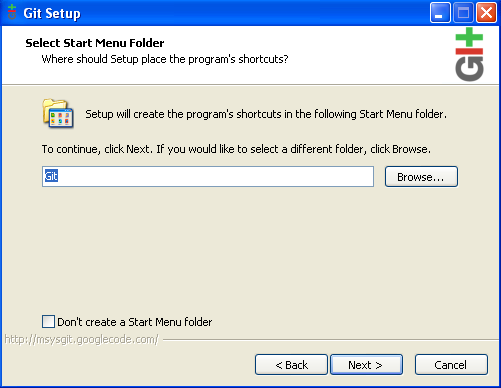
Na próxima tela, indique o diretório onde deseja instalar o Git em seu computador:



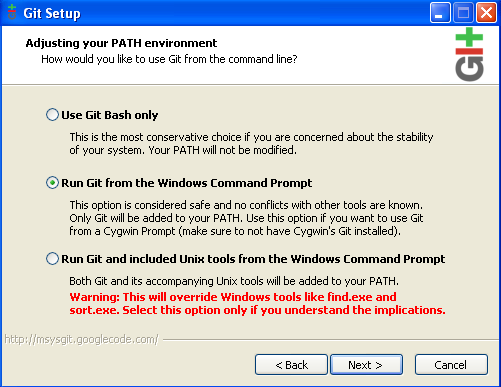
Em seguida, indique quais utilitários serão instalados junto do Git. Marque todas as opções, exceto "git-cheetah shell extension", como mostrado na imagem a seguir:



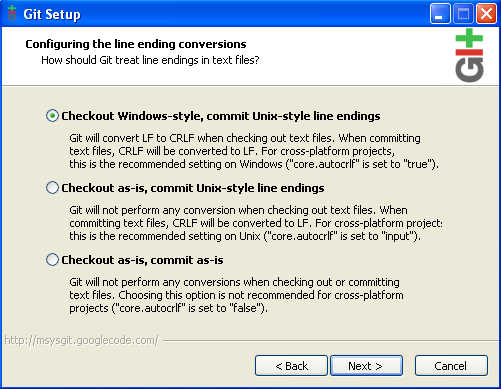
A seguir, defina o nome do menu do Git para o menu Iniciar do Windows:



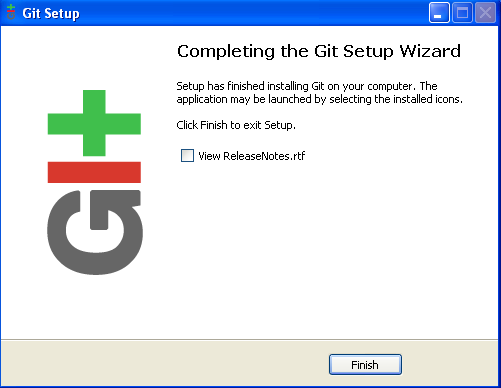
O msysgit instala por padrão em seu computador um programa próprio chamado Git Bash, onde é possível manipular os repositórios de arquivos que trabalharemos no futuro. Caso você prefira, é possível usar o próprio prompt do Windows. Basta, na próxima janela, marcar a opção "Run Git from the Windows Command Prompt", como indicado na imagem a seguir:



Na janela a seguir, indique que queremos que as quebras de linha sejam consideradas da mesma maneira, independente do sistema operacional trabalhado. Para isso, marque a primeira opção:



Nesse instante, a cópia dos arquivos será feita para o computador, e em seguida a mensagem de que a instalação foi finalizada aparecerá:



#### MacOS

Para instalar o Git no Mac OS X (Snow Leopard ou superior), há um instalador pronto. Basta baixá-lo em <http://code.google.com/p/git-osx-installer/downloads/list?can=3>. Na listagem, selecione o arquivo **git-1.7.7.3-intel-universal-snow-leopard.dmg**.

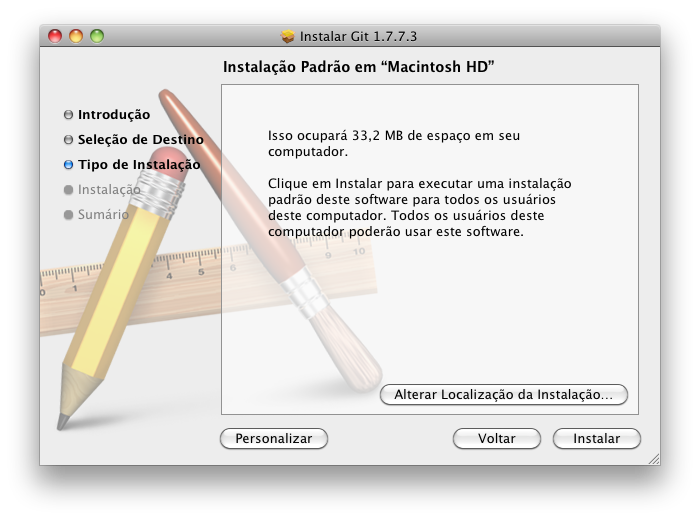
Após baixá-lo, basta clicar duas vezes para ter acesso ao pacote de instalação. Agora é só clicar duas vezes no arquivo "git-1.7.7.3-intel-universal-snow-leopard.pkg" (de acordo com a versão encontrada no momento da escrita) para iniciar o processo de instalação:

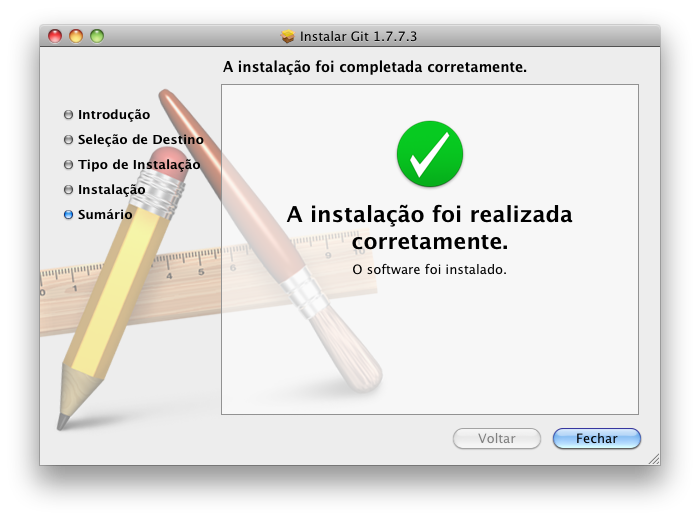


Na tela seguinte, selecione a opção "Instalar para todos os usuários desse computador" para continuar:

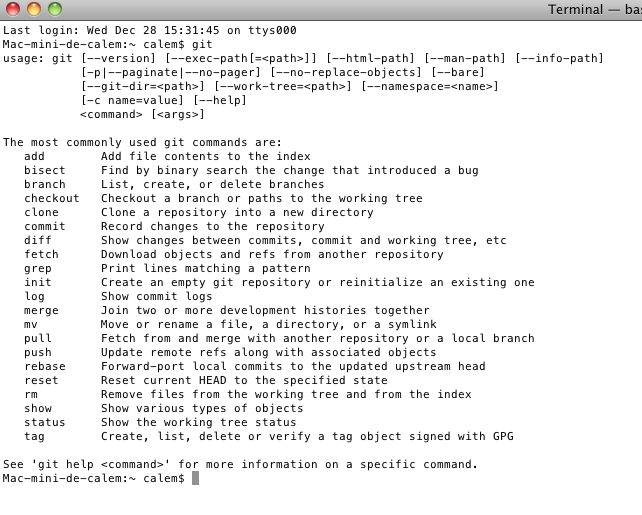


Depois é só prosseguir com a instalação padrão.





Para testar a instalação, abra a aplicação "Terminal" e digite o comando git. A saída deve ser similar à imagem:



Opcionalmente, é possível configurar o Terminal para completar os comandos do Git ao pressionarmos a tecla "tab", além de mostrar na linha de comando se a pasta atual está sendo rastreada pelo Git. Para isso é necessário adicionar as seguintes linhas ao arquivo de perfil do usuário para o prompt de comando, habitualmente encontrado na pasta home do usuário com o nome de .bash\_profile (ou .bashrc).

Adicione as seguintes linhas ao fim do arquivo:

if [ -f /usr/local/git/contrib/completion/git-completion.bash ]; then

. /usr/local/git/contrib/completion/git-completion.bash

fi

GIT\_PS1\_SHOWDIRTYSTATE=true

PS1='\u@\h:\w $(\_\_git\_ps1 "(%s)")\$ '

Ao iniciar uma nova janela ou aba do Terminal, as alterações estarão aplicadas. Caso queira aplicar as alterações imediatamente no Terminal digite source ~/.bash\_profile.

Obs.: Caso tenha efetuado a instalação do GIT via Homebrew (<http://brew.sh/>), alterar o endereço do arquivo **git-completion.bash** para **/usr/local/Cellar/git/\*/etc/bash\_completion.d/git-completion.bash**.

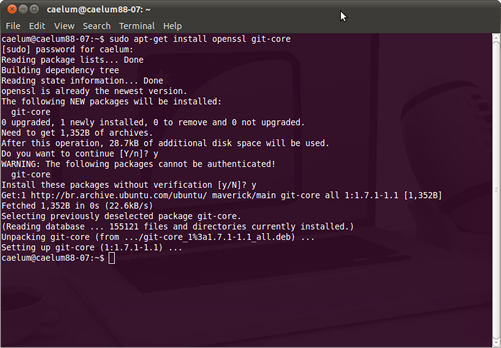
#### Linux

Em qualquer sistema Linux, podemos utilizar o gerenciador de pacotes da respectiva distribuição para instalar o Git. No Ubuntu e no Debian, por exemplo, basta instalar o pacote git-core para ter o Git instalado. No Fedora, o pacote git. E no Gentoo, o pacote dev-util/git.

Portanto, por exemplo, para fazer a instalação do Git no Ubuntu, basta executar o comando:

sudo apt-get install openssl git-core

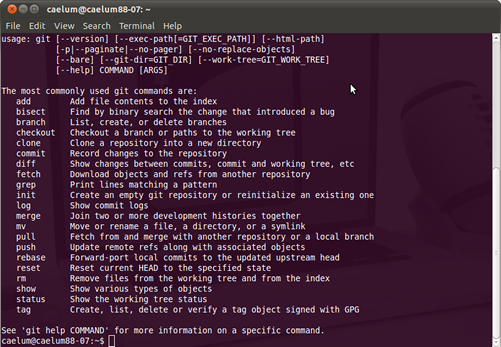
Siga as instruções do prompt de comando, primeiro confirmando a instalação dos pacotes e suas dependências, depois confirmando a instalação do pacote git-core, como demonstrado na imagem:



É possível testar se o Git foi corretamente instalado rodando o comando git no prompt de comando:

git

A saída deve ser uma mensagem de ajuda, similar à demonstrada na imagem:



Ao instalar o Git, um script de ajuda que completa os comandos do Git no terminal é instalado, juntamente com um script que demonstra na linha de comando se estamos em um diretório que é um projeto rastreado pelo Git. Para ativá-los, é necessário modificar o arquivo de perfil do prompt de comando. Normalmente esse arquivo encontra-se na pasta home do usuário e chama-se .bash\_profile (em alguns casos .bashrc ou .zshrc).

Caso nenhum dos arquivos citados exista, crie-o; caso exista, adicione as seguintes linhas ao final dele:

if [ -f /etc/bash\_completion.d/git ]; then

. /etc/bash\_completion.d/git

fi

GIT\_PS1\_SHOWDIRTYSTATE=true

PS1='\u@\h:\w $(\_\_git\_ps1 "(%s)")\$ '

Salve o arquivo. A partir de agora, caso estivermos em uma pasta rastreada pelo Git no prompt de comando, seremos informados. Para ativar as alterações imediatamente, execute o comando source informando o arquivo de perfil que alteramos:

source ~/.bash\_profile

### Criando conta no Github

Agora que temos o Git instalado, podemos utilizar os serviços do Github. Primeiro precisamos criar uma conta. Para isso acessamos a página <https://github.com/plans>, escolhemos o tipo de conta que queremos criar e prosseguimos preenchendo os formulários pedidos.

Depois de criada nossa conta, precisamos gerar uma senha (chave de segurança) que será responsável por identificar nossa máquina quando fizermos as interações entre nosso projeto e o serviço do Github. Caso não exista essa configuração entre nossa máquina e a conta do Github, o acesso ao serviço pelos comandos do Git será negado.

O processo de criação da chave de segurança é similar em todos os sistemas operacionais: basta abrir o prompt de comando (no caso do Windows, é necessário abrir o Git Bash) e inserir o seguinte comando:

ssh-keygen -t rsa -C "seu\_email@provedor.com"

Lembre-se de substituir seu\_email@provedor.com pelo seu endereço real de email. A resposta do terminal vai perguntar em qual local do seu disco você quer salvar sua chave de segurança. Para evitar problemas, mantenha a opção padrão. Em algumas versões, pode ser necessário incluir também a opção -b 2048 ou -b 4096 para indicar o tamanho da chave a ser gerada.

A seguir, será solicitada a entrada de uma senha para a chave de segurança. Caso o computador seja público ou compartilhado, é recomendado que sua chave esteja protegida por uma senha. Caso contrário, recomendamos que a senha seja ignorada, bastando pressionar Enter ao ser solicitada a senha e a confirmação de senha.

Ao término do processo você deve ver uma mensagem similar à seguinte:

Your identification has been saved in /Users/seu\_usuario/.ssh/id\_rsa.

Your public key has been saved in /Users/seu\_usuario/.ssh/id\_rsa.pub.

The key fingerprint is:

01:0f:f4:3b:ca:85:d6:17:a1:7d:f0:68:9d:f0:a2:db seu\_email@provedor.com

The key's randomart image is:

+--[ RSA 2048]----+

| .+ + |

| = o O . |

| = \* \* |

| o = + |

| o S . |

| o o = |

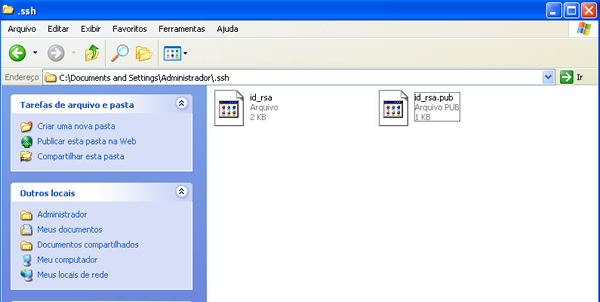
| o . E |

| |

| |

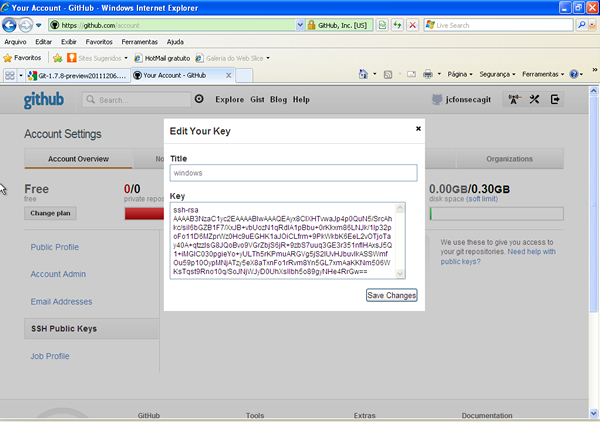
+-----------------+

Com isso, dois arquivos serão gerados: a sua chave privada e a sua chave pública, conforme pode ser visto na imagem a seguir:



Agora será necessário realizar o login no [http://github.com](http://github.com/) e seguir para as configurações do seu perfil. Siga "Settings", "SSH and GPG keys" e "New SSH key".

Insira na caixa de texto o conteúdo do seu arquivo id\_rsa.pub. O local exato do arquivo foi informado na saída do processo de criação da chave. Cole-o exatamente como ele está, sem adição de espaços ou quebras de linha. Para isso, abra-o no editor de textos simples do seu sistema operacional (TextEditor, Bloco de Notas, GEdit), como pode ser visto na figura a seguir:



Agora você tem o Git instalado em sua máquina, uma conta no Github e as configurações necessárias para utlizar o serviço com segurança.

### Copiando o projeto para sua máquina: o comando clone

Iniciaremos o curso trabalhando em um repositório já existente. Para podermos fazer isso, nós temos que copiá-lo para nossa máquina. O repositório que vamos copiar está hospedado no Github. Para garantir que vamos adquirir todos os arquivos necessários, utilizaremos o comando git cloneque, além de copiar os arquivos do projeto, traz todos os arquivos necessários para que o controle de versão dos arquivos continue sendo feito.

Abra o prompt de comando do computador e navegue até uma pasta que será a pasta "pai" do seu repositório local, por exemplo:

cd Documentos/Projetos

Lembre-se de que você pode escolher o local mais conveniente para você em seu computador. Agora vamos executar o comando que copiará o repositório para essa pasta:

git clone https://github.com/jcfonsecagit/repositorio.git

Esse comando **clone** criará uma pasta com o mesmo nome padrão do repositório (que no caso é "repositorio") e copiará para esse diretório todos os arquivos que estavam disponíveis nele.

Nosso projeto de exemplo contém páginas HTML. Para visualizar o projeto basta abrir o arquivo **index.html** em qualquer navegador (Chrome, Firefox, Internet Explorer, entre outros).

Você pode ver os arquivos do projeto navegando para a pasta "repositorio" e listando os arquivos no prompt de comando:

cd repositorio

ls

### Controle geral de versões: o comando tag

Uma situação comum no desenvolvimento de um projeto de software é que, conforme ele vai evoluindo, vai ganhando novos releases e, consequentemente, novas versões. Com isso o software passa a ter a versão 1.0, 1.1, 2.0, 3.0 e assim por diante.

O Git nos permite, sempre que desejarmos, adicionarmos um marcador no repositório. Então, em determinado momento do projeto, é possível marcar o estado atual do repositório como a versão 1.0. Esse processo de adicionar um marcador, que habitualmente é o número da versão, é chamado de tag. É possível criar tags com qualquer nome, para que não fiquemos presos a um determinado padrão de nomenclatura de versões. Isso nos permite inclusive marcarmos o repositório como "v0.1a", "v0.1b" com a finalidade de definirmos versões preliminares, de acordo com a necessidade do projeto atual.

Mas como é possível visualizar as tags existentes num projeto? No Git, é possível listar todas as Tags existentes em um projeto através do comando tag:

git tag

Feito isso, como podemos fazer para voltar o repositório para alguma dessas versões? Por exemplo, queremos visualizar como o nosso projeto, junto com todos os seus arquivos html, estava na versão v0.1. Para isso, basta dizer ao git que queremos buscar a versão v0.1. Chamamos esse processo de **checkout** e podemos executá-lo com o comando a seguir:

git checkout v0.1

Abrindo a página html novamente, podemos verificar como ela estava na versão v0.1.

#### Como saber o que mudou nos arquivos?

E se entre a versão v0.2 e a v0.1 surgiu um bug? Como descobrir o que mudou entre elas para isolar onde pode estar o bug? As ferramentas de controle de versão possuem tarefas específicas para mostrar a diferença (diff) entre duas versões de qualquer elemento do projeto, podendo ser um arquivo, um diretorio etc. No nosso caso, queremos ver as diferenças entre a versão v0.2 e a v0.1, o que conseguimos com o Git através do comando git diff, onde indicamos entre quais versões desejamos visualizar as diferenças.

git diff v0.1 v0.2

Com isso, serão mostradas as linhas existentes numa versão que não aparecem na outra versão. Também serão mostradas as alterações efetuadas entre as versões selecionadas.

#### Como descobrir quem realizou as alterações em um arquivo linha a linha?

Dentro do projeto, é comum encontrar linhas de código cuja compreensão não é imediata. Como descobrir o motivo pelo qual uma linha foi alterada ou o motivo para que ela tenha sido implementada de uma determinada maneira? Se analisando apenas o contexto não é possível compreender o motivo, que tal falar com o responsável por aquela modificação?

O Git nos fornece uma funcionalidade onde podemos consultar quem foi o autor de cada linha de um arquivo através do comando git blame:

git blame index.html

Para sair do blame, basta apertar a tecla q

#### Conclusão

Como vimos através dessa apresentação simples de apenas alguns comandos disponíveis no Git, temos uma ferramenta que permite muito mais segurança no controle de alterações de arquivos de um projeto. A utilidade dessa ferramenta é indiscutível: com ela, é possível realizar com facilidade alguns controles que antes eram muito trabalhosos ou até mesmo impossíveis de se fazer quando trabalhamos simplesmente com pastas compartilhadas em uma máquina ou servidor.

Durante este curso, aprenderemos outros comandos que o Git nos fornece para solucionar diversos problemas desde o trabalho básico de criação de um repositório, commits, branches, conflitos, como trabalhar no repositório sem conexão com a internet, trabalhar em diferentes implementações simultaneamente num mesmo projeto, modificação de um arquivo por mais de uma pessoa simultaneamente no projeto e tratar os conflitos, entre outros.

### Mais sobre o Git

Agora que você aprendeu um pouco sobre o que é o *Git*, selecione quais das alternativas a seguir descrevem suas funções e características.

Top of Form



Só permite que o usuário trabalhe quando está conectado à internet.



Manter um histórico das alterações feitas no projeto.

Cada alteração que você faz fica gravada em um histórico, podendo ser visualizada e restaurada a qualquer instante.



Marcar diferentes versões para o seu projeto.

O *Git* permite que você marque um dado estado do seu projeto com uma etiqueta, que pode ser usada para determinar uma versão dele.



*Git* é a ferramenta de controle de versão mais antiga do mercado.



Organizar o sistema de trabalho em equipe, centralizando os arquivos em um servidor.

Uma das funções do *Git* é centralizar os arquivos de trabalho, impedindo que haja confusão sobre o que foi e não foi modificado por cada um.

Selecione as alternativas que descrevem as funções e características do *Github*, como visto em aula.

Top of Form



O *Github* é gratuito e você pode ter quantos repositórios quiser, entre públicos e privados.



Já inclui o controle de versão do *Git*.

Do mesmo modo que aquele servidor que vimos em aula, o *Github* mantém a integridade do código quando há diversos desenvolvedores trabalhando ao mesmo tempo.



É uma plataforma online na qual você pode hospedar o seu código fonte.

Do mesmo modo que aquele servidor que vimos em aula, o *Github* hospeda o código fonte do seu projeto.

#### Instalando o Git

Para prosseguirmos com esse curso, precisaremos instalar o **Git**!

Para isso, escolha seu sistema operacional na página de [Downloads](http://git-scm.com/downloads) do projeto e siga as instruções de instalação do programa.

##### Windows

Para instalarmos o **git**, basta fazermos o download no link acima e seguirmos as instruções de instalação do próprio instalador.

##### Linux

Para instalarmos o **Git** no Linux utilizaremos apt-get.

Abra o terminal do Linux e digite o comando abaixo:

sudo apt-get install git

Agora é só colocar sua senha e aguardar a instalação.

##### MacOS

Para instalarmos o **Git** no MacOS utilizaremos o gerenciador de downloads **Brew**.

Caso você não tenha o **Brew** instalado, rode o comando abaixo:

/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

Aguarde a instalação do Brew e prossiga executando o comando abaixo:

brew install git

##### Instalou?

Rode o comando git no seu terminal para checar se a instalação foi um sucesso!

##### Clonando o projeto

Para clonar um repositório do *Github*, basta usar o comando git clone URL, substituindo **URL** pelo endereço de clonagem do *Github*. Como vimos em aula, o endereço de clonagem pode ser encontrado na página do repositório, ou adicionando-se o final .git ao endereço da página do repositório.

Por exemplo: o projeto **repositório** é visualizável pela página <https://github.com/jcfonsecagit/repositorio>. Basta adicionar .git ao final para obter a **URL** para clonagem.

Faça o teste! Clone o projeto com o comando a seguir e depois veja que arquivos foram copiados para o seu computador!

git clone https://github.com/jcfonsecagit/repositorio.git

Vimos no video um comando que pode mostrar as versões do nosso projeto.

Utilize e descubra: quantas tags de versões existem no projeto repositorio?

Top of Form



4



2

Isso! Para visualizar as tags de um projeto, basta entrar no diretório do repositório e digitar o comando git tag. Neste projeto há duas versões: **v0.1** e **v0.2**. A saída do comando vai a seguir:

$ git tag

v0.1

v0.2

$



0



1



3

Para trocar de versão para a **v0.1**, qual o comando que devemos utilizar?

Top of Form



$ git clone v0.1



$ git checkout versao-0.1



$ git clone https://github.com/jcfonsecagit/repositorio.git v0.1



$ git checkout v0.1

Isso mesmo! A opção checkout altera seu repositório local fazendo com que seus arquivos fiquem no mesmo estado em que estavam quando a tag v0.1 foram criados!

##### 8. Diferença entre versões

Para ver a diferença entre dois pontos do projeto, no caso duas versões, usamos a opção diff do *git*. Use o comando e escolha a opção que representa o que foi alterado da versão **v0.1** pra **v0.2**.

Top of Form



Não há diferença.



A frase <h3>Versão 0.1</h3> foi removida.

Isso mesmo! Ao rodar o comando git diff v0.1 v0.2 vemos as alterações entre uma versão e a outra. No caso, a frase está com um símbolo - na frente, indicando que esta foi removida da versão **0.1** pra **0.2**.



A frase <h3>Versão 0.1</h3> foi adicionada.

Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form

## O ciclo básico do Git

O Git é uma ferramenta de controle de versão baseada no sistema de arquivos, ou seja, podemos fazer a associação de uma pasta diretamente a um repositório. Então, vamos criar um novo diretório que conterá os arquivos do nosso projeto:

mkdir curso-git

cd curso-git

Mas, como essa é uma pasta como qualquer outra em nosso computador, será que o git já sabe que ela conterá os arquivos do nosso projeto? Como indicar que essa pasta será o nosso futuro repositório? Isso é feito a partir do seguinte comando:

git init

Será exibida uma mensagem similar a: Initialized empty Git repository in /diretorio/repositorio/do/git E pronto. Já temos um repositório.

### Adicionando arquivos ao repositório

Agora começaremos o nosso projeto em si. Faremos um projeto que conterá páginas HTML. Então, vamos criar o primeiro arquivo, uma simples página HTML vazia chamado index.html:

<html>

<head>

</head>

<body>

</body>

</html>

Já temos o nosso primeiro arquivo em nosso projeto. Mas, será que o git já sabe que o arquivo criado pertence ao repositório? Para tal, podemos verificar quais os arquivos que pertencem ao nosso repositório. Isso pode ser feito digitando o comando:

git ls-files

E aí? Retornou algo? Nada? Onde está o nosso arquivo? Como fazemos para adiciona-lo no repositório? Para confirmar, podemos verificar o estado dos arquivos do nosso projeto:

git status

E teremos como saída:

# On branch master

#

# Initial commit

#

# Untracked files:

# (use "git add <file>..." to include in what will be committed)

#

# index.html

Com isso, podemos verificar que o nosso arquivo está na condição de Untracked files, isto é, ele não está na lista de arquivos cujas alterações serão rastreadas, ou controladas. Isso acontece porque o Git não sabe que deve controlar as alterações deste arquivo, ou seja, que ele deva fazer o track. Então, como dizemos ao Git que ele deve realizar o track? Isso é feito a partir do comando git addpassando o nome do arquivo do qual o Git deve fazer o track. No nosso caso, queremos adicionar o arquivo index.html ao repositório:

git add index.html

Vamos verificar novamente o estado dos arquivos do nosso repositório com o comando git status:

# On branch master

#

# Initial commit

#

# Changes to be committed:

# (use "git rm --cached <file>..." to unstage)

#

# new file: index.html

#

Vemos que o nosso arquivo passou para condição de Changes to be committed, isto é, na lista de arquivos que estão prontos para o commit. Mas o que é commit? Toda vez que terminamos de realizar as alterações nos arquivos de um projeto, precisamos "entregar" essas alterações, isto é, realizar um commit. E somente as alterações que estiverem sob a condição de "Changes to be commited" é que serão entregues.

Agora, podemos realizar o primeiro commit do nosso projeto. Mas como é que o Git sabe quem é o responsável pelo commit? Na seção anterior, vimos o comando git blame, o qual mostrava os responsáveis por cada linha de código. Nós precisamos informar ao Git o nosso nome e e-mail. Isso é feito com os seguintes comandos:

git config user.name "João Carlos Fonseca"

git config user.email "jcfonsecagit@gmail.com"

Com isso, o usuário João Carlos Fonseca será o responsável pelas alterações no repositório atual. Porém, se quisermos fazer isso para outros repositórios, temos que dar o mesmo comando toda vez? Felizmente, o Git nos fornece a opção de definir um nome de usuário e e-mail para todo o sistema, isto é, deixando esta configuração global:

git config --global user.name "João Carlos Fonseca"

git config --global user.email "jcfonsecagit@gmail.com"

Caso essa configuração não seja efetuada, o Git vai determinar o nome de usuário atual do terminal como o autor das alterações. É importante que essa configuração seja feita para que seja mais fácil encontrar suas próprias alterações e facilitar a comunicação entre os membros do time caso haja alguma dúvida sobre um código feito por outra pessoa.

Agora sim, estamos prontos para executar o nosso primeiro commit do projeto. Para que isso aconteça, devemos executar:

git commit -m "Início do projeto"

Com isso, realizamos o primeiro commit do sistema. A flag "-m", indica que o conteúdo a seguir é a mensagem que será utilizada para descrever o que está sendo feito no commit. Verificando os estados dos arquivos do nosso sistema com o comando git status, verificamos que não há nenhuma alteração em nosso projeto.

# On branch master

nothing to commit (working directory clean)

### Alterando o projeto

Vamos continuar o nosso projeto modificando o arquivo index.html:

<html>

<head>

</head>

<body>

<h1>Git</h1>

<h2>Trabalhando em Equipe com Controle e Segurança</h2>

<p>Um curso que explora os benefícios de utilizar o Git como ferramenta de controle de versão para projetos em qualquer linguagem, em qualquer plataforma.</p>

<h3>Principais benefícios:</h3>

<ul>

<li>Funciona de maneira distribuída</li>

<li>Permite a edição concorrente de arquivos do projeto</li>

<li>Não depende de uma conexão ativa com um servidor</li>

</ul>

</body>

</html>

Verificando o estado dos arquivos novamente com o comando git status, percebemos que o Git já reconhece que temos arquivos que foram modificados no nosso projeto desde o nosso último commit:

# On branch master

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: index.html

#

no changes added to commit (use "git add" and/or "git commit -a")

Contudo, vemos que essas alterações ainda não fazem parte das alterações que serão adicionadas no próximo commit. Para adicionar esses arquivos para o próximo commit, precisamos rodar novamente o comando git add com o nome do arquivo.

Verificando o estado novamente, vemos que o nosso arquivo index.html está sob a condição "Changes to be committed".

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: index.html

#

Agora as alterações estão prontas para o commit. Ao rodar o comando git commit -m "Conteúdo da página index.html", temos o segundo commit do nosso projeto.

### Recapitulação do processo

Durante o ciclo básico que demonstramos anteriormente, nós interagimos com 3 estágios diferentes do repositório. O primeiro deles enquanto nós criamos o repositório, mas não indicamos nenhum arquivo para ser rastreado.

Nesse estágio, estamos interagindo com um estado do projeto que chamamos de "Working Directory", ou seja, é o nosso sistema de arquivos atual. Nele estão as alterações que estamos realizando no momento.

O Working Directory pode estar "limpo", quando não há diferença entre os arquivos armazenados no repositório e como estão atualmente. Quando há diferença (por exemplo, alteramos determinado arquivo, mesmo que uma alteração mínima), o Working Directory fica marcado como "sujo".

Em nosso caso o arquivo não existia para se ter uma comparação, pois tínhamos um repositório novo. Após modificarmos nossos arquivos a ponto de definirmos que um "passo" foi concluído, criamos uma visão desse passo, um ponto de controle preliminar com o comando "git add".

Esse comando cria um novo estágio do repositório, o que chamamos de "Index" ou "Staging Area". Esse estágio é transitório e pode ser alterado ainda antes de se tornar um passo do projeto: podemos adicionar novos arquivos, removê-los ou mesmo alterá-los.

Quando satisfeitos com o conteúdo do "Index", utilizamos o comando "git commit" para persistí-lo, gravar esse passo com todos os arquivos novos e alterações efetuadas. O comando "commit" criou um terceiro estágio do repositório que conhecemos como "HEAD". O HEAD é o último estado que o Git usa como referência, é a visão do último passo do projeto que foi concluído e entregue.

### Conclusão

Uma ferramenta como o Git nos permite fazer alterações em nosso projeto com mais segurança pois sabemos que podemos controlar cada alteração, às vezes de maneira bem detalhada, de cada um dos nossos arquivos.

Apesar de poderosa, o Git não é uma ferramenta totalmente automatizada (automatizar esse processo seria impossível, pois o gerenciamento das versões varia muito de acordo com o ambiente do projeto e da equipe). Precisamos interagir com o Git constantemente para podermos extrair o melhor dos benefícios que ele oferece.

Apesar de demonstrarmos o uso do Git desde o início do projeto, podemos utilizar os mesmos passos em um projeto existente para que ele possa, a partir de um momento, ser controlado pelo Git.

Esse capítulo demonstrou o ciclo básico do trabalho de controle de versões de um projeto do ponto de vista de um único desenvolvedor. Com o Git podemos trabalhar em equipe, conforme veremos nos próximos capítulos. Apesar de podermos trabalhar em equipe, não precisamos, por enquanto, nos conectar a um servidor central; fizemos o controle local do nosso projeto. Isso permite que possamos trabalhar em um projeto mesmo sem ter acesso à rede ou à internet pois essas alterações podem ser aplicadas mais tarde ao repositório "central".

### Criando um novo repositório

Qual comando utilizamos para informar ao Git que a pasta atual no prompt de comando deve ser tratada como um repositório?

Top of Form



git .



git start



git add



git init

O comando git init inicia um novo repositório! É isso aí!



git track

### O que é mesmo o Git?

"O Git é uma ferramenta de controle de versão de arquivos que permite o rastreio de alterações em diversos arquivos em um mesmo repositório, não importando a localização dos mesmos no sistema de arquivos."

A afirmação acima é:

Top of Form



Falsa

Cada repositório deve ser associado diretamente com uma pasta no sistema de arquivos e, portanto, a afirmação é falsa.



Verdadeira

Bottom of Form

O Git permite que realizemos o controle de versão de um repositório, porém um repositório deve ser associado diretamente com uma pasta no sistema de arquivos. Portanto, não podemos adicionar arquivos localizados em pastas diferentes que não estejam associadas ao mesmo repositório.

### Quais os estados possíveis?

Quais são os três estados de um repositório com os quais interagimos ao realizar o fluxo de controle com o Git?

Top of Form



commit, add e clone



unsaved, saved e controlled



working directory, index e HEAD

working directory representa o estado atual dos arquivos no repositório. Podemos utilizar o comando git add para adicioná-los ao index, ou staging area, que representa uma visão preliminar das modificações que queremos definir para nosso projeto, comparando essas ao HEAD, o último passo completo do nosso projeto, que serve de referência para comparação do trabalho com o working directory.



init, add e commit



untracked, tracked e modified

Bottom of Form

Bottom of Form

### Verificando as alterações

Como podemos verificar quais arquivos foram alterados, em comparação com o último passo (commit) do repositório?

Top of Form



O comando git status lista o estado atual de cada arquivo, caso tenha sido modificado

O comando git status exibe um resumo dos arquivos modificados, adicionados ou excluídos de seu working directory, em comparação com o HEAD.



O comando git ls traz uma listagem dos arquivos do repositório, marcando com um \*aqueles que foram modificados



O ícone do arquivo na interface do explorador de arquivos fica marcado em vermelho

Bottom of Form

### Como rastrear um arquivo?

Qual o comando utilizado para informarmos ao Git que desejamos rastrear determinado arquivo, incluindo-o no controle de versão?

Top of Form



git status



git add

Correto! O comando git add, seguido pelo caminho relativo ou completo do arquivo, registra que o mesmo deve ser adicionado ao controle de versão!



git blame



git commit

Bottom of Form

### Adicionando múltiplos arquivos

Vimos que o comando git add pode receber como parâmetro o nome de um arquivo que deve ter suas alterações rastreadas. Assim, o mesmo será adicionado ao index do repositório. De que outras maneiras podemos utilizar o comando git add? Como podemos adicionar mais de um arquivo ao index ao mesmo tempo? Escolha as alternativas que representem corretamente estes outros modos.

Top of Form



git add arquivo1 arquivo2

Você pode adicionar arquivos específicos passando cada um como um parâmetro, separando cada arquivo por um espaço!



git add caminhoDeUmDiretorio

Você pode adicionar um diretório inteiro passando como parâmetro do comando git add o caminho do diretório!



git add arquivo1, arquivo2



git add all



git add .

O caminho git add . funciona do mesmo modo que passar o caminho de um diretório, pois o atalho . representa o diretório atual. Portanto, serão adicionados todos os arquivos que estiverem dentro do diretório atual!

### Marcando alterações interativamente

O comando git add pode ser executado de modo *interativo* com a opção -i. Ao rodar o comando completo, git add -i, o modo interativo inicia um prompt de comando específico, aguardando as decisões do usuário sobre quais as alterações detectadas no working directory devem ser adicionadas ao index.

Realize uma alteração substancial ao arquivo **index.html** (por exemplo, adicione um novo parágrafo de texto dentro da tag <body> **e** altere um texto existente).

Agora adicione o arquivo utilizando o modo interativo:

1. Rode o comando git add -i
2. $ git add -i
3. staged unstaged path
4. 1: unchanged +1/-0 index.html
5. \*\*\* Commands \*\*\*
6. 1: status 2: update 3: revert 4: add untracked
7. 5: patch 6: diff 7: quit 8: help

What now>

1. Para adicionar arquivos alterados, utilize a opção update. Você pode digitar tanto o número, quanto a palavra escrita como também somente a primeira letra dela. No caso, utilizamos aqui a abreviação u e pressione enter.
2. What now> u
3. staged unstaged path

1: unchanged +1/-0 index.html

1. Aparecerá uma listagem dos arquivos modificados, com um número atribuído a cada arquivo. Digite o número que representa o arquivo index.html e pressione enter .
2. Update>> 1
3. staged unstaged path

\* 1: unchanged +1/-0 index.html

1. Aparecerá um asterisco ao lado do número do arquivo, indicando que ele será adicionado. Pressione enter novamente para confirmar e voltar ao menu principal.
2. Update>>
3. updated one path
4. \*\*\* Commands \*\*\*
5. 1: status 2: update 3: revert 4: add untracked

5: patch 6: diff 7: quit 8: help

1. Se quiser, confira as modificações. Utilize a opção status para conferir que o arquivo index.html foi adicionado corretamente, digitando a opção e pressionando enter.
2. What now> status
3. staged unstaged path
4. 1: +1/-0 nothing index.html
5. \*\*\* Commands \*\*\*
6. 1: status 2: update 3: revert 4: add untracked

5: patch 6: diff 7: quit 8: help

1. Utilize a opção quit para terminar a interação!
2. What now> 7
3. Bye.

$

Agora você pode fazer o commit do arquivo, que já está adicionado ao index. Confira usando o comando git status!

### Fazendo commit sem criar um index

Num ciclo simples de controle de um projeto com o *Git*, muitas vezes podemos querer adicionar ao HEAD todas as alterações que foram realizadas no working directory. O comando git commitoferece uma opção para que possamos evitar o passo de adicionar manualmente os arquivos ao index, gerando um commit que já inclui todas as modificações feitas no projeto.

Qual é essa opção? Utilize o comando git commit --help para ver quais as opções disponíveis!

Top of Form



git commit --no-index



git commit -a

A opção -a inclui imediatamente no commit todos os arquivos modificados ou removidos! Porém, ela não adiciona os arquivos novos. Podemos combinar a opção de mensagem com git commit -a -m 'Mensagem de commit' para não precisar escrever a mensagem no editor padrão do console.



git commit -m



git commit --force

Bottom of Form

Bottom of Form

### Outro modo de escrever suas mensagens

O comando git commit pode também ser utilizado sem passar a opção -m. Neste caso, o editor de texto padrão de seu terminal vai abrir para que você digite a mensagem de commit. Ao salvar, caso a mensagem não esteja vazia, o commit será criado.

VER OPINIÃO DO INSTRUTOR

### Opinião do Instrutor

O editor abrirá com um espaço para preenchermos a mensagem e, logo abaixo, uma listagem dos arquivos que serão e não serão enviados. Essas linhas que começam com # não serão enviadas na mensagem, pois são comentários!

Esse modo de escrever a mensagem de commit é bastante interessante quando a mensagem que queremos escrever é um pouco mais complicada do que uma simples frase. Conseguimos, por exemplo, colocar itens na mensagem de commit, utilizando diversas linhas ao invés de uma única.

Dica: você pode alterar o editor padrão definindo a variável de ambiente EDITOR com o comando do editor de texto que você quer que ele execute.

## Sincronizando e criando repositórios remotos

Apesar do Git nos permitir realizar os commits de maneira local, quando desejarmos trabalhar com uma equipe na qual várias pessoas precisarão de acesso aos arquivos em seus computadores, precisaremos de um local onde todas os arquivos e suas respectivas modificações estejam disponíveis.

Uma possibilidade inocente para resolver esse problema é, sempre que um arquivo for alterado, efetuarmos sua gravação em um pen-drive e distribuí-lo para as outras pessoas da equipe. No entanto, essa tarefa tende a se tornar muito trabalhosa com o passar do tempo. Imagine a qualquer alteração ter que ficar passando pen-drive para todos os colegas da equipe. Totalmente inviável.

Uma outra possibilidade é termos um diretório na rede onde todos possuem acesso e, sempre que uma informação for alterada por alguma pessoa, a alteração seja refletida nesse diretório. Assim, todos as outras pessoas da equipe também podem recuperar essas alterações.

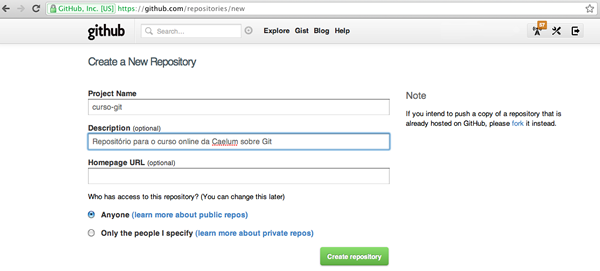
Esse local remoto onde os arquivos ficam centralizados para todos os desenvolvedores é conhecido como repositório remoto, e é justamente o que o Github nos permite criar: repositórios para os quais podemos submeter nossos arquivos e outras pessoas de uma equipe possuam acesso e permissão para manipulá-los de maneira colaborativa. No entanto, ainda não temos nenhum repositório criado para começarmos a enviar nosso arquivo.

# Criação de repositórios no Github para guardar os arquivos

Sempre que desejarmos armazenar os arquivos de um novo projeto, precisamos criar um novo repositório. O Github nos permite a criação de ilimitados repositórios, desde que eles sejam abertos e visíveis a todos os usuários. Em planos pagos, é possível ter repositórios privados, onde apenas determinados usuários possuam acesso.

Para conseguirmos criar nosso repositório, é preciso estar logado no Github, com uma conta já previamente criada e configurada, como visto no capítulo 1.

Uma vez logado no Github, existe na [página inicial](https://github.com/), do lado direito, um botão chamado "New Repository", que permitirá a criação do novo repositório. Em seguida, basta indicarmos o nome do projeto, que no nosso caso é "curso-git", e, opcionalmente, uma descrição. Nesse instante, você deverá estar em uma tela similar à abaixo.



Podemos também definir a visibilidade do repositório que, no caso, deixaremos como público. Pronto, clicando em "Create Repository" teremos o nosso novo repositório remoto, hospedado no Github.

# Configuração do repositório remoto

No capítulo anterior, foram realizados commits. Porém, todos eles de maneira local, ou seja, eles só estão visíveis para o próprio autor do commit e ainda não estão disponíveis no repositório remoto.

Para conseguirmos compartilhar nosso projeto HTML, precisamos indicar que o diretório do nosso projeto apontará para um repositório remoto, no caso, o que acabamos de criar no Github. Para realizarmos esse processo, o Git possui o comando git remote add, com o qual podemos indicar a localização do repositório remoto e o nome que queremos dar para ele (um apelido ou alias).

A sintaxe completa do comando é:

git remote add [alias\_do\_repositorio] [uri\_do\_repositorio]

Portanto, precisamos ainda de duas informações para completar o comando, que são o alias do repositório e a url de onde ele estará disponível.

Uma convenção adotada é a utilização do nome do repositório remoto como "origin". No entanto, qualquer outro nome pode ser utilizado. Em seguida, devemos saber também a URI do repositório, que é um caminho único que indica o local onde ele ficará armazenado. O próprio Github, segue uma convenção com relação à URI de seus repositórios, sendo a seguinte:

https://github.com/[nome\_do\_usuario]/[nome\_do\_repositorio].git

Com isso, para o nosso projeto, teremos o seguinte caminho:

https://github.com/[seu\_nome\_do\_usuario]/curso-git.git

Logo, o comando completo que devemos executar no prompt é:

git remote add origin https://github.com/[seu\_nome\_de\_usuario]/curso-git.git

Ao executarmos o comando, nenhuma saída é mostrada no prompt.

# Envio dos commits locais para o repositório

No capítulo anterior, realizamos alguns commits locais, os quais queremos enviar ao servidor para que outras pessoas consigam visualizar o trabalho feito.

Para isso, uma vez que o repositório já tenha sido inicializado com o comando git init, os commits locais já tenham sido realizados e o repositório remoto configurado, basta executarmos o comando git push, indicando qual é o repositório remoto para onde os commits serão enviados e a branch que será enviada para o servidor. O repositório remoto será o "origin", que acabamos de configurar, enquanto a branch será a "master", criada por padrão sempre que um repositório é criado. Aprederemos mais sobre branches nos próximos capítulos.

Com isso, teremos o comando: git push origin master.

Executando o push, teremos uma saída similar com a seguinte, no prompt:

Counting objects: 3, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 272 bytes, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/[seu\_usuario\_do\_github]/curso-git.git

\* [new branch] master -> master

# Contribuição com o projeto: clone de repositórios

Agora que o repositório do projeto está disponível e inclusive já possui commits realizados, outros desenvolvedores podem querer contribuir com o projeto. O primeiro passo para que um outro usuário realize suas alterações é ter em seu próprio computador uma cópia do repositório. Dessa forma, ele terá todos os arquivos disponíveis em seu computador, para alterar como desejar.

O processo de copiar o repositório remoto para um computador, a fim de realizar alterações nos arquivos ou até mesmo para ter os arquivos no computador, é chamado de "clone" e pode ser realizado através do comando git clone [uri\_do\_repositorio].

O desenvolvedor que quiser colaborar em um repositório pode descobrir a URI de clonagem dele acessando sua página no Github. Nesse caso, o endereço de clonagem será:

https://github.com/[usuario\_do\_dono\_do\_repositorio]/curso-git.git.

Pronto, agora é possível que outros desenvolvedores trabalhem no projeto. Basta executar o clone com o comando: git clone https://github.com/[usuario\_do\_dono\_do\_repositorio]/curso-git.git.

Após a execução do comando, um diretório chamado "curso-git" será criado e a seguinte saída será exibida:

Cloning into curso-git...

remote: Counting objects: 3, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 3 (delta 0)

Unpacking objects: 100% (3/3), done.

Caso queira, é possível que o nome da pasta em seu computador seja diferente. Basta especificar o nome desejado no fim do comando:

git clone [https://github.com/[usuario\_do\_dono\_do\_repositorio]/curso-git.git](https://github.com/%5busuario_do_dono_do_repositorio%5d/curso-git.git) meuprojeto

O exemplo acima copia o repositório "curso-git" para uma pasta chamada "meuprojeto". Lembre-se de utilizar essa opção com cautela, quando o nome do repositório conflitar com outra pasta em seu computador por exemplo, pois é preferível manter a consistência entre o nome do repositório e o nome da pasta. Portanto, vamos manter o primeiro exemplo de comando.

Ao clonar um repositório, não precisamos adicionar o repositório remoto através do comando git remote add, pois tudo já é feito pelo comando git clone, dando um alias "origin" para o repositório.

É importante ter em mente que o processo de clonagem de um projeto pode demorar. O tempo pode variar de acordo com o tamanho do projeto que está sendo clonado e também com a velocidade de conexão.

Uma vez que o colaborador realizou o clone do repositório em seu computador e agora possui todos os arquivos localmente, as alterações desejadas por ele serão feitas localmente, igual aprendido na seção anterior. Ao final das alterações, ele realizará o envio dos commits para o repositório remoto através da execução do comando git push origin master.

Ao executar o "push", veremos uma mensagem similar à abaixo no prompt.

Counting objects: 5, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 304 bytes, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/[usuario\_no\_github]/curso-git.git

c886b6a..b9aa50c master -> master

A partir desse momento, acessando a página do repositório no Github, é possível ver a lista com os commits realizados.

# Sincronização com as novas alterações do repositório

A partir do momento em que fizemos atualizações no repositório, os outros desenvolvedores que já o possuem em seus computadores não estarão sincronizados com estas alterações, ou seja, eles ainda não possuem em seus computadores as novas versões dos arquivos.

Para que a sincronização seja realizada e o desenvolvedor tenha em seu computador as novas versões dos arquivos, basta que ele execute o comando git pull origin master. A saída no prompt será similar à abaixo, indicando os arquivos que tiveram alterações e quantas linhas do arquivo foram afetadas.

remote: Counting objects: 5, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 3 (delta 0)

Unpacking objects: 100% (3/3), done.

From github.com:[usuario\_no\_github]/curso-git

c886b6a..b9aa50c master -> origin/master

Updating c886b6a..b9aa50c

Fast-forward

index.html | 1 +

1 files changed, 1 insertions(+), 0 deletions(-)

# Visualização de log de commits

Uma vez que o projeto possua commits no repositório remoto, conseguimos visualizar no próprio site do Github que os commits estão publicados. Basta acessar o endereço [https://github.com/[seu\_usuario\_no\_github]/[projeto]/commits/master](https://github.com/%5bseu_usuario_no_github%5d/%5bprojeto%5d/commits/master), onde "master" é o nome da branch utilizada que, novamente, estudaremos melhor nos próximos capítulos.

Ao visualizarmos o log dos commits através da página do repositório no Github, temos uma visualização agradável do que acontece com o projeto e sua evolução. No entanto, muitos desenvolvedores podem achar o fato de ter que abrir o navegador para visualizar a lista de commits um processo muito trabalhoso.

Para facilitar a visualização dos commits, é possível também executar o comando git log em um prompt, no diretório de um projeto, que mostrará informações como o autor, a data e hora e a mensagem de commit utilizada. Uma possível saída para esse comando é:

commit b9aa50ccb68dc6c60bca197b7a5ee5e0b1d556f2

Author: Nome de Um Usuário <email1@servidor.com>

Date: Tue Dec 27 17:35:50 2011 -0200

Um outro commit de outro usuário

commit c886b6ae3d8963ddba8cc80e434f572f72276ea8

Author: Nome de outro usuário <email2@servidor.com>

Date: Tue Dec 27 16:47:15 2011 -0200

Primeiro commit

Ao executar o comando git log, note que ele é extremamente rápido, pois, quando clonamos o repositório, todas as informações dele estão localmente em nosso computador, ou seja, não precisamos acessar nada de fora de nossa máquina.

Contudo, às vezes desejamos obter mais informações sobre os commits realizados: conhecer quais os arquivos que foram modificados no commit. Será que temos que colocar na mensagem do commit para saber quais os arquivos modificados? Felizmente, o Git nos fornece uma ferramenta que permite visualizar os arquivos modificados. O comando é o git whatchanged. Uma possível saída para esse comando é:

commit b9aa50ccb68dc6c60bca197b7a5ee5e0b1d556f2

Author: Nome de Um Usuário <email1@servidor.com>

Date: Tue Dec 27 17:35:50 2011 -0200

Um outro commit de outro usuário

:100644 100644 6b6c919... 5270907... M index.html

Nesse caso, está sendo indicado que o arquivo index.html sofreu modificações (indicado pela letra M).

# Controle de permissões do repositório do github

Muitas vezes queremos contribuir com um projeto, mas não possuímos permissão para realizarmos as contribuições como por, exemplo, novos commits.

Conseguimos perceber que não possuímos permissão para um repositório no Github ao acessarmos ele via o navegador e notar que a URL para clonarmos o projeto possui a mensagem "Read-Only access" ao seu lado. Um exemplo pode ser visualizado na imagem a seguir:



No nosso caso, como criamos o repositório curso-git no começo dessa seção, podemos também dar permissão para outros colaboradores. Por exemplo, caso queiramos adicionar o usuário fictício "João Carlos Fonseca" aos colaboradores do projeto, podemos acessar o endereço [https://github.com/[seu\_usuario\_no\_github]/curso-git/settings/collaboration](https://github.com/%5bseu_usuario_no_github%5d/curso-git/settings/collaboration) e adicionarmos o id do usuário que, nesse caso, é **jcfonsecagit**.

# Criação do repositório curso-git

Através da sua conta de usuário no Github, crie um novo repositório chamado curso-git. Para isso, siga as instruções:

1. Acesse e faça login na [página do github](https://github.com/)
2. Logado, clique no botão + no canto superior direito (ao lado na sua imagem de perfil) e escolha **New repository**
3. Digite curso-git no campo **Repository name** e clique **Create repository**

É isso aí! Seu repositório está criado e pronto para trabalhar.

### Opinião do Instrutor

No próximo exercício vamos utilizar o endereço do seu repositório. Ele é algo semelhante ao seguinte:

https://github.com/[seu\_usuario\_no\_github]/curso-git.git

Lembre-se de trocar **[seu\_usuario\_no\_github]** por seu nome de usuário. No caso do João Fonseca, seria **jcfonsecagit**, ficando ao final:

https://github.com/jcfonsecagit/curso-git.git

**Configurando o repositório remoto**

Agora que temos um repositório remoto, precisamos configurá-lo para sincronizar com o repositório local. Para isso, precisamos ir ao diretório do repositório local no terminal e executar o seguinte comando:

git remote add origin https://github.com/[seu\_usuario\_no\_github]/curso-git.git

Com isso, você está indicando que o repositório Git local se conecta com um repositório **remoto**através do **alias** ou atalho origin cujo endereço real é https://github.com/[seu\_usuario\_no\_github]/curso-git.git.

Agora que temos um repositório remoto, precisamos configurá-lo para sincronizar com o repositório local. Para isso, precisamos ir ao diretório do repositório local no terminal e executar o seguinte comando:

git remote add origin https://github.com/[seu\_usuario\_no\_github]/curso-git.git

Com isso, você está indicando que o repositório Git local se conecta com um repositório **remoto**através do **alias** ou atalho origin cujo endereço real é https://github.com/[seu\_usuario\_no\_github]/curso-git.git.

VER OPINIÃO DO INSTRUTOR

### Opinião do Instrutor

Lembre-se de substituir **[seu\_usuario\_no\_github]** por seu nome de usuário. Se quiser conferir o resultado, utilize o comando git remote -v. A saída deve ser algo assim:

origin https://github.com/[seu\_usuario\_no\_github]/curso-git.git (fetch)

origin https://github.com/[seu\_usuario\_no\_github]/curso-git.git (push)

**Fazendo o primeiro push**

No capítulo anterior, você gerou alguns commits, que ainda não foram enviados para o repositório remoto. Agora que configuramos ele, podemos enviar nossos commits para lá através do comando git push origin master.

Esse comando envia os dados da branch local master para a branch remota master, localizada no repositório origin.

VER OPINIÃO DO INSTRUTOR

### Opinião do Instrutor

Se tudo ocorreu certinho, a saída deve ser bem parecida com essa:

Counting objects: 3, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 272 bytes, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/[seu\_usuario\_do\_github]/curso-git.git

\* [new branch] master -> master

Outra alternativa é utilizar, no primeiro push, a opção -u ou --set-upstream. Ela atrela a branch remota à local, fazendo com que não seja mais necessário passar como parâmetros a origem e a branch no comando push, que fica então assim: git push.

# O que é copiado quando clono um projeto?

Quando um desenvolvedor executa um git clone [repositorio], o que é copiado para sua máquina?

Top of Form



Arquivos e histórico dos commits



Apenas os arquivos



Tudo que existe naquele repositorio

Exatamente! Ao realizarmos a clonagem, uma cópia de todo o repositório remoto é criada em nossa máquina. Todos os arquivos, tags, histórico, branches e assim por diante ficam disponíveis localmente na máquina do desenvolvedor.



Somente histórico

**Histórico dos commits do repositório**

Após ter feito o push para o repositório remoto, vamos dar uma olhada no tal histórico de commits, ou log. Há dois jeitos de fazê-lo:

* Na página do seu repositório curso-git, na aba **<> Code**, há uma faixa com o número de commits, branches, releases e contributors. Clique nos **commits** para visualizar o histórico!
* No terminal, execute o comando git log, que mostra o histórico direto na tela.

### Opinião do Instrutor

O endereço do seu histórico de commits será semelhante a https://github.com/[seu\_nome\_de\_usuario]/curso-git/commits/master, com a devida substituição. Já a saída do git log será semelhante a esta:

$ git log

commit cb85f706f988b6946d1c0532235ddca06fc28927

Author: Joao Carlos Fonseca <jcfonsecagit@gmail.com>

Date: Wed Jan 4 15:27:58 2012 -0200

commit no gitgui

commit b0b4d89dce0422394b146af9d5fef2f879860d43

Author: Joao Carlos Fonseca <jcfonsecagit@gmail.com>

Date: Wed Dec 28 17:19:22 2011 -0200

versao 0.2 sem o H3 no index.html

commit c5d65339242003b1a80602414c785a797506a9f7

Author: Joao Carlos Fonseca <jcfonsecagit@gmail.com>

Date: Wed Dec 28 17:18:26 2011 -0200

versao 0.1 do index.html

$

Se precisar, pressione q para sair do histórico no terminal.

**Histórico mostrando as modificações**

Às vezes desejamos visualizar mais informações do que o git log nos fornece, como por exemplo quais arquivos foram modificados no commit. Para isso existe o comando git whatchanged. Mas e se quisermos ver também quais as linhas modificadas?. Para isso, basta passar uma opção ao comando git whatchanged para que o Git nos forneça tal informação. Qual é essa opção?

Top of Form



git whatchanged -v



git whatchanged -d

* Alternativa correta



git whatchanged -p

Ao executarmos o comando git whatchanged -p é possível visualizar quais as linhas que foram modificadas em cada commit do nosso projeto. Obs: também é possível utilizarmos git log -p, que imprime também os commits nos quais não houve modificação.



git whatchanged -a



git whatchanged -r

Bottom of Form

Bottom of Form

**Adicionando permissão para outras pessoas em um repositório**

Um projeto no Github pode ter diversos colaboradores. Para fazer isso, siga as instruções abaixo:

1. Abra a página do projeto no Github
2. Abra a aba **Settings**
3. Clique na seção **Collaborators**, à esquerda
4. Digite o usuário do colaborador no campo ao lado do botão **Add collaborator**. Durante a digitação, aparecerão os usuários mais próximos da busca atual.

Adicione algum amigo seu que tenha conta no Github como colaborador do seu projeto. Caso prefira, você pode adicionar algum desenvolvedor da Caelum ao seu projeto também. Algumas sugestões são adrianoalmeida7, guilhermesilveira, asouza, williammizuta ou gabrielso.

VER OPINIÃO DO INSTRUTOR

### Opinião do Instrutor

A URL que permite adicionar colaboradores a um projeto é parecida com esta: https://github.com/[seu-nome-de-usuario]/curso-git/settings/collaboration

Já a URL que mostra informações dos colaboradores que já fizeram algum push no repositório se parece com essa: https://github.com/[seu-nome-de-usuario]/curso-git/graphs/contributors

Peça para seu amigo fazer alguns commits e veja as informações que aparecem!

## Organização do trabalho com branches

# O que é branch e por que utilizá-las?

É comum, durante o desenvolvimento de novos recursos ou de correção de bugs, interrompermos o trabalho por falta de tempo ou porque surgiu uma nova prioridade do projeto. Mas o que fazer com as alterações que estão pela metade? Deletar e depois ter que refazer tudo novamente? Colocar no repositório algo pela metade, podendo quebrar todo o sistema?

Uma solução que é bem utilizada no dia a dia é a de criar uma seção separada do projeto, uma bifurcação, uma branch. Tal solução possibilita desenvolver separadamente cada uma das funcionalidades sem interferir no desenvolvimento de uma outra parte do projeto.

# Trabalhando com branches: primeiros passos

Já sabemos que a utilização de branches facilita no dia a dia do desenvolvedor. Mas como criar uma nova branch? Para tal, utilizamos o comando git branch, passando como opção o nome da branch que desejamos criar. No nosso caso, criaremos a branch design, onde realizaremos algumas alterações referentes ao design da nossa página html:

git branch design

Ao executarmos o comando, nenhuma saída é mostrada no prompt.

Agora, como verificamos quais são as branches existentes em um projeto? Isso se resolve com o comando git branch. Ele nos fornece todas as branches criadas na máquina. Ele também possibilita visualizar qual a branch que estamos atualmente através de um "\*" que precede o nome da branch atual.

git branch

design

\* master

Observe que o "\*" precede uma branch chamada master. Mas nós não a criamos agora. De onde ela surgiu? A branch master é criada quando executamos o nosso primeiro commit do projeto. Ela é considerada a branch principal do projeto.

Mas, se quisermos alterar o projeto numa outra branch, como é que fazemos para alterar a branch atual? Isto é feito através do comando git checkout, passando o nome da branch para a qual desejamos mudar. No nosso caso, temos:

git checkout design

Switched to branch 'design'

E pronto. Todas as alterações que realizaremos a partir de agora estarão na branch design.

Vamos adicionar estilo para a nossa página. Para tal, copie o seguinte código num arquivo chamado design.css. Adicione este arquivo na pasta do seu projeto.

body {

background-color: blue;

}

Adicione também a seguinte linha no header do seu arquivo index.html:

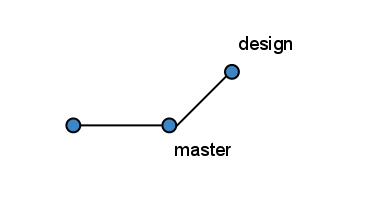
<link rel="stylesheet" type="text/css" href="/design.css"/>

Porém, essas alterações ainda não foram atualizadas no repositório. Para isso, precisamos adicionar os arquivos e commitar as alterações:

git add design.css index.html

git commit -m "Adicionando estilo para a nossa página"

E pronto! As alterações estão salvas na branch design. O esquema do projeto com as branches pode ser vista na figura a seguir:



Se voltarmos para a branch master com o comando git checkout master, vemos que as alterações feitas anteriormente não estão mais presentes.

Por fim, perceba que todas essas alterações foram realizadas sem precisar de conexão com a internet. O Git nos permite trabalhar tanto com o repositório remoto quanto com o nosso próprio repositório local, ao contrário de outros controladores de versão como o SVN e CVS.

# Compartilhando branches locais com outros desenvolvedores

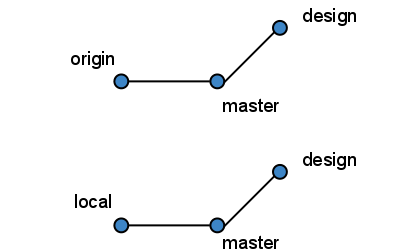
Agora, o que acontece se você começou alguma alteração em um projeto e não terminou? Será que é bom manter apenas localmente? E se alguém quiser continuar as alterações que você iniciou?

E se criamos uma nova ferramenta que quebra a compatibilidade com as versões anteriores? Para resolver esses problemas, é bom manter também essas branches no repositório remoto.

Iniciaremos este trabalho enviando a branch criada localmente para o repositório remoto. Isso é feito utilizando o comando git push passando dois argumentos: o primeiro é o nome do repositório e o segundo, o nome da branch que deseja-se enviar. No nosso caso, temos:

git push origin design

Com isso, o repositório remoto conterá uma cópia fiel da branch design local. Isso pode ser visto na figura a seguir:

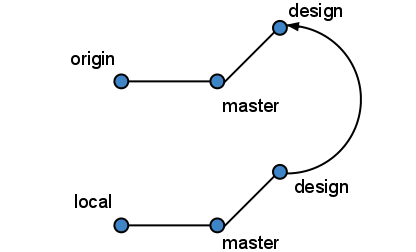


Porém, toda vez que atualizarmos tanto o nosso projeto local quanto o projeto remoto, precisaremos indicar qual o repositório e o nome da branch que a nossa branch local se refere no remoto, isto é, precisaremos digitar git pull origin design e git push origin design para atualizar os repositórios locais e remotos, respectivamente.

Para evitar tal trabalho, podemos indicar o caminho (track) da branch remota para a nossa branch local. Isso pode ser feito no instante em que criamos a branch remota através da opção "-u". No nosso caso, temos:

git push -u origin design

Com isso, a nossa branch local sabe qual a branch remota que ela se referencia.



E como podemos visualizar as branches já existentes em um repositório remoto? Isso é feito através da opção "-r" passado ao comando git branch.

git branch -r

origin/HEAD -> origin/master

origin/design

Uma vez visto as branches remotas, como copiar uma delas para a máquina local? Isso é feito passando o nome do repositório e da branch remota ao comando git branch, além de indicar o nome da branch que será criada. Mais uma vez, temos o problema de indicar o caminho entre as branches. Para este caso, a opção -t resolve.

git branch -t design origin/design

Temos como resultado:

Branch design set up to track remote branch design from origin.

Switched to a new branch 'design'

**Vantagens do uso de branches**

Selecione, das opções abaixo, as que representem vantagens do uso de branches para o desenvolvimento.

Top of Form



Todos os desenvolvedores podem trabalhar na mesma branch, sem chance de conflitos.



A instabilidade de uma funcionalidade em andamento não afeta os outros desenvolvedores.

Como cada branch é um trecho de código isolado, o código do desenvolvedor A incompleto e causando erros não afetará o trabalho do desenvolvedor B, pois o código da branch A não será propagado para a branch B.



Permite remover o código antes que ele seja incorporado ao código principal.

Às vezes, ocorre de uma tarefa ser interrompida ou descartada do projeto. Ao utilizar branches, o código novo está isolado e pode ser descartado com facilidade.



O projeto fica ramificado e cria muitas frentes independentes.

Bottom of Form

**Criando uma branch local**

Deseja-se criar uma nova branch chamada design. Qual é o comando para realizar tal tarefa?

Top of Form



git commit design



git add design



git branch design

Com este comando criamos uma nova branch chamada design. Se quiser criar e já mudar para a branch nova, basta dar o comando git checkout -b design, que indica que você quer mudar para a branch design, mas ao mesmo tempo criá-la com a opção -b que representa *nova branch*.



git checkout design



git status design

**Listando branches locais**

Qual é o comando para listar todas as branches locais?

Top of Form



git status



git checkout



git add



git commit

* Alternativa correta



git branch

Ao digitar o comando git branch, todas as branches locais são listadas. Passando a opção -a as branches remotas são incluídas.

Bottom of Form

Bottom of Form

**Trocando de branch**

Qual é o comando para trocar de branch para a branch chamada design?

Top of Form



git status design



git move design



git branch design



git checkout design

Isso! Com este comando, trocamos o ambiente atual para a branch design.



git add design

**Criando e trocando de branch local**

Uma tarefa bem comum é a criação de uma nova branch seguida da troca para essa branch criada. Poderíamos fazer isso com a sequência de comandos git branch nomeDaBranch e git checkout nomeDaBranch. Para facilitar, o *Git* nos fornece uma opção do comando git checkout que realiza este trabalho. Qual a opção que deve ser passada para realizar tal tarefa?

Top of Form



-t

Esta opção cria uma ligação entre a branch a ser utilizada e uma branch remota, mas não cria uma nova branch.



-d



-r



-b

Sim! Esta opção simula a execução dos dois comandos em sequência, criando e já alterando o espaço de trabalho para a nova branch.



-u

Esta opção não existe e resultará em um erro.



-a



-K

Bottom of Form

Bottom of Form

**Deletando branch local**

Uma vez terminada a tarefa pela qual uma branch foi criada, desejamos deletá-la para não poluir muito o nosso projeto com branches que não serão mais utilizadas. Para tal, passa-se uma opção ao comando git branch. Qual é essa opção?

Top of Form



-u

* Alternativa correta



-b

* Alternativa correta



-d

Sim, este comando remove uma branch, porém somente se ela estiver sincronizada com outra. Senão, é necessário forçar com a opção -D.

* Alternativa correta



-a

* Alternativa correta



-r

Bottom of Form

**Criando branch remota**

Realizando o comando git push origin design, criamos a branch remota design no repositório referente ao origin. Porém, ela não está rastreada, isto é, a nossa branch local não sabe qual é a branch remota à qual ela se refere. Para resolver esse problema, passamos uma opção ao comando anterior. Qual é essa opção?

Top of Form

* Alternativa correta



-d

* Alternativa correta



-a

* Alternativa correta



-u

Ao digitar o comando git push -u origin design, criamos a branch remota design no repositório referente ao origin e criamos uma ligação entre a nossa branch e a branch remota.

* Alternativa correta



-t

* Alternativa correta



-b

* Alternativa correta



-r

Bottom of Form

**Listando branch remota**

Qual a opção que passamos ao comando git branch para que sejam listadas **somente** as branches remotas do nosso repositório?

Top of Form

* Alternativa correta



-t

* Alternativa correta



-u

* Alternativa correta



-a

* Alternativa correta



-b

* Alternativa correta



-d

* Alternativa correta



-r

Esta opção lista todas as branches remotas!

**Listando branch locais e remotas**

O git nos fornece uma opção ao comando git branch para listar tanto as branches locais quanto as remotas. Qual é essa opção?

Top of Form

* Alternativa correta



-u

* Alternativa correta



-d

* Alternativa correta



-a

Esta opção mostra todas as branches, locais e remotas!

* Alternativa correta



-t

* Alternativa correta



-r

* Alternativa correta



-b

**Copiando uma branch remota**

Uma sequência de tarefas bem comum referente à cópia de uma branch localizada num repositório remoto é:

## **Criação de uma branch local com o mesmo nome da branch remota**

## **Mudança para essa nova branch criada**

## **Criação de link entre a branch local e remota**

Para evitar toda essa tarefa, o git nos fornece um atalho para evitar esse trabalho todo com uma opção do comando git checkout. Que opção é essa?

Top of Form

* Alternativa correta



-t

Com o comando git checkout -t origin/design, todo esse trabalho é feito automaticamente para a branch chamada design.

* Alternativa correta



-u

* Alternativa correta



-r

* Alternativa correta



-b

* Alternativa correta



-a

* Alternativa correta



-d

Bottom of Form

Curiosidade: caso seu repositório local só tenha uma origin, já saiba que existe uma branch remota chama design, e não exista ainda uma branch local de mesmo nome, o comando git checkout design também faz esse processo todo de uma vez!

**Deletando uma branch remota**

Uma vez concluída a tarefa referente à uma branch remota, desejamos excluí-la para não acumular branches que não serão mais usadas. Quais os possíveis comandos para deletar uma branch remota?

Top of Form

* Alternativa correta



git branch origin :design

* Alternativa correta



git push -d origin design

Este comando remove a branch remota design.

* Alternativa correta



git push origin design

* Alternativa correta



git checkout -d origin design

* Alternativa correta



git push origin :design

Este comando remove a branch remota design.

* Alternativa correta



git push origin/design

Bottom of Form

**Verificando atualizações do repositório**

Num repositório remoto, as alterações são realizadas, geralmente, por mais de uma pessoa ao mesmo tempo. Como saber se foram criadas branches novas no repositório remoto?

Top of Form

* Alternativa correta



git checkout origin

* Alternativa correta



git branch origin

Este comando cria uma nova branch chamada origin, o que pode criar confusão com o nome do repositório remoto.

* Alternativa correta



git fetch origin

Este comando verifica todas as atualizações que foram realizadas no repositório de atalho origin.

* Alternativa correta



git push origin

Bottom of Form

Realizando o comando git fetch origin, podemos verificar todas as atualizações que foram realizadas no repositório referente ao origin.

## Resolução de conflitos

É muito comum no desenvolvimento de um software existirem várias pessoas trabalhando sobre a mesma base de código. Com isso, todos os desenvolvedores possuem acesso ao código fonte do projeto e podem manipulá-lo, tornando-se possível que mais de um desenvolvedor realize alterações em um mesmo arquivo fonte. Um exemplo é o designer precisar alterar algumas tags HTML em uma página, enquanto um desenvolvedor precisa adicionar um código Javascript na mesma página.

Existem algumas estratégias para que os desenvolvedores possam mexer nos arquivos sem se preocupar em conflitar com alterações de outros desenvolvedores.

Uma das estratégias é realizar um **"Lock"** em um arquivo antes de editá-lo. Dessa maneira, nenhum outro desenvolvedor consegue manipulá-lo até que o "Lock" seja liberado. Essa abordagem possui alguns pontos falhos como, por exemplo, o desenvolvedor passa o dia inteiro trabalhando naquele arquivo e, ao ir embora para casa no final do dia, esquece de liberar o "Lock". Nenhum outro desenvolvedor conseguirá editá-lo, provavelmente até o dia seguinte, quando o "dono do Lock" chegar ao trabalho e liberar.

Uma outra abordagem possível é permitir que vários desenvolvedores manipulem o mesmo arquivo ao mesmo tempo. No entanto, o envio do arquivo para o repositório só pode ser feito se o desenvolvedor que deseja enviá-lo estiver devidamente sincronizado (atualizado) com o repositório. Essa é justamente a maneira seguida pelo Git.

# Como o Git trata as várias alterações no mesmo arquivo?

Vamos considerar o repositório "propostas\_homepage", cujo objetivo é ser um repositório para manter as propostas para homepage de uma empresa. O dono desse repositório é o usuário "jcfonsecagit", que adicionou a usuária "mmsoaresgit" como colaboradora. Vamos também considerar que já existe um arquivo chamado "proposta\_1.html", que é uma página com o seguinte conteúdo:

<html>

<head>

<title>Proposta 1 para homepage da empresa</title>

</head>

<body>

<h1>Cabeçalho do site</h1>

Conteúdo do site

<h4>Rodapé do site</h4>

</body>

</html>

Nesse instante, **ambos os usuários estão sincronizados**, ou seja, possuem a mesma base de código atualizado em seus computadores.

O usuário "jcfonsecagit" decide alterar o título da página, trocando a palavra "homepage" para "site". Enquanto isso, ao mesmo momento, a usuária "mmsoaresgit" decide alterar o mesmo arquivo para fazer com que o rodapé utilize a tag h1 em vez da h4. Ambos realizam o commit em seus computadores. Porém, o usuário "jcfonsecagit" realiza logo em seguida o push. Nesse instante, a outra usuária, "mmsoaresgit", tenta realizar o git push. Nessa situação, o comando falhará com a seguinte mensagem:

To git@github.com:jcfonsecagit/propostas\_homepage.git

! [rejected] master -> master (non-fast-forward)

error: failed to push some refs to 'git@github.com:jcfonsecagit/propostas\_homepage.git'

To prevent you from losing history, non-fast-forward updates were rejected

Merge the remote changes (e.g. 'git pull') before pushing again. See the

'Note about fast-forwards' section of 'git push --help' for details.

Apesar da mensagem soar estranha e grande em um primeiro momento, ela está simplesmente dizendo que a usuária "mmsoaresgit" não estava sincronizada com o repositório e, por esse motivo, o git push falhou. Note também que na própria mensagem está a solução. Ele nos pede que façamos o git pull antes e, assim, fiquemos atualizados com o repositório. Nesse momento, se notarmos bem, um problema poderia surgir, pois vamos trazer as alterações do mesmo arquivo que mexemos, ou seja, o arquivo "proposta\_1.html". O que acontecerá com o arquivo da usuária "mmsoaresgit" se ela realizar esse git pull?

Quando "mmsoaresgit" executa o git pull, a seguinte mensagem surge no console:

remote: Counting objects: 5, done.

remote: Compressing objects: 100% (1/1), done.

remote: Total 3 (delta 1), reused 3 (delta 1)

Unpacking objects: 100% (3/3), done.

From github.com:jcfonsecagit/propostas\_homepage

2844856..601c3dc master -> origin/master

Auto-merging proposta\_1.html

Merge made by recursive.

proposta\_1.html | 2 +-

1 files changed, 1 insertions(+), 1 deletions(-)

Essa mensagem indica que o Git não perdeu as alterações da "mmsoaresgit" e, melhor ainda, ele percebeu que os arquivos foram alterados em partes distintas e que uma alteração não conflita com a outra. Repare que o Git automaticamente tratou as diferenças entre os arquivos. Chamamos esse processo de tratar as diferenças existentes no mesmo arquivo, gerados por diferentes commits, de **merge**. Uma das grandes vantagens do Git é ter um mecanismo de verificação extremamente avançado. Dessa maneira, se uma alteração não sobrepõe a outra, ele consegue fazer o trabalho para nós, o que foi justamente o que acabou de acontecer com a usuária "mmsoaresgit". Uma indicação de que esse merge automático aconteceu pode ser vista na lista ou log de commits com o comando git log, onde agora se encontrará o seguinte commit:

commit 8b864af3b7d7e48e32731fc168db6c959677c964

Merge: bf04eb4 601c3dc

Author: Maria Soares <mmsoaresgit@gmail.com>

Date: Thu Dec 29 15:10:57 2011 -0200

Merge branch 'master' of github.com:jcfonsecagit/propostas\_homepage

# O tratamento manual de conflitos

Mas o que acontece se ambos os usuários desejam alterar a mesma parte do conteúdo? Vamos ver um exemplo em que isso acontece.

O usuário "jcfonsecagit" deseja alterar o conteúdo da página para uma tag div contendo o seguinte texto: "Aqui está o conteúdo da página". Em seguida, ele realiza o commit com a página modificada.

<html>

<head>

<title>Proposta 1 para homepage da empresa</title>

</head>

<body>

<h1>Cabeçalho do site</h1>

<div>

Aqui está o conteúdo da página

</div>

<h1>Rodapé do site</h1>

</body>

</html>

Enquanto isso, a usuária "mmsoaresgit" deseja utilizar outra mensagem para o conteúdo da página. Ela coloca a mensagem "Isso aqui é o corpo da página", dentro também de uma div, e realiza o commit localmente. Com isso, ela terá o seguinte código:

<html>

<head>

<title>Proposta 1 para homepage da empresa</title>

</head>

<body>

<h1>Cabeçalho do site</h1>

<div>

Isso aqui é o corpo da página

</div>

<h1>Rodapé do site</h1>

</body>

</html>

Nesse momento, o usuário "jcfonsecagit" realiza o git push, ou seja, envia seus arquivos para o repositório remoto. Em seguida, a usuária "mmsoaresgit" também tenta realizar o git push, que novamente falhará, pois ela não está sincronizada com o repositório. Como aprendemos a resolver há pouco, a usuária "mmsoaresgit" pode fazer o git pull para sincronizar seus repositórios e pedir pro Git realizar o merge das alterações existentes. Porém, ao executar o git pull, receberemos a seguinte mensagem:

Auto-merging proposta\_1.html

CONFLICT (content): Merge conflict in proposta\_1.html

Automatic merge failed; fix conflicts and then commit the result.

Repare que a mensagem diz explicitamente que o Git tentou realizar o merge para nós, mas não conseguiu. A razão para essa falha é que foram feitas alterações no mesmo trecho do arquivo e ele não sabe qual dos dois trechos deve manter. Para resolver esse problema, é preciso fazer o merge manualmente.

O primeiro passo para realizar o merge manualmente é abrir o arquivo onde ocorreu o conflito (propostas\_1.html), que agora estará com o seguinte conteúdo:

<html>

<head>

<title>Proposta 1 para homepage da empresa</title>

</head>

<body>

<h1>Cabeçalho do site</h1>

<div>

<<<<<<< HEAD

Isso aqui é o corpo da página

=======

Aqui está o conteúdo da página

>>>>>>> e5600a4fd30fa7df61e6c1f156f55222f3041de4

</div>

<h1>Rodapé do site</h1>

</body>

</html>

Não se assuste com o conteúdo estranho que surgiu no código, vamos entendê-lo. Lembre-se que HEAD significa o que tínhamos commitado em nosso ambiente local. No caso da "mmsoaresgit", o bloco que aparece entre <<<<<<< HEAD e ======= representa a alteração que ela fez: "Isso aqui é o corpo da página". Já o que está entre ======= e >>>>>>> e5600a4fd30fa7df61e6c1f156f55222f3041de4representa o conteúdo com o qual o conteúdo anterior conflitou: "Aqui está o conteúdo da página". Repare também que a sequência e5600a4fd30fa7df61e6c1f156f55222f3041de4 é um identificador do commit. Dessa forma, conseguimos também saber com qual commit o conflito ocorreu.

Nesse momento, é necessário decidir com qual dos trechos de código queremos ficar ou se realizaremos uma mescla dos dois. Nessa situação, a "mmsoaresgit" decidiu permancer com a frase: "Isso aqui é o conteúdo da página", que é uma combinação dos dois commits. Nesse instante, deve-se retirar todo o trecho gerado pelo Git, colocando apenas o conteúdo que deve ficar no arquivo. Em seguida, adicionamos o arquivo através do git add propostas\_1.html à lista de arquivos que serão commitados, realizamos o git commit e o git push. Pronto, conseguimos fazer o merge manualmente.

**Problemas com o merge**

O processo de tratar conflitos é o mais crítico quando se trabalha com controle de versões e também um dos mais complicados de se compreender.

Se em nenhum momento você já precisou realizar o processo de merge quando trabalhou com controle de versões, imagine como seria o processo de corrigir os conflitos feitos por vários desenvolvedores.

Faça o teste! Crie um repositório local e acesse por duas abas diferentes do terminal, em duas branches diferentes. Faça diversas alterações em cada uma, jogue as alterações para a master e veja quais as dificuldades.

VER OPINIÃO DO INSTRUTOR

### Opinião do Instrutor

Entre as maiores dificuldades está o fato de você não necessariamente saber o que um outro desenvolvedor quis fazer. Por isso, nem sempre nos sentimos seguros em tratar conflitos. Para isso, é bom que cada commit tenha uma mensagem clara que indique a intenção do desenvolvedor para aquelas alterações.

**Evitando conflitos com o Lock**

O Git possui um processo automático de merge que é um recurso que não existe, por exemplo, no CVS. Logo, qualquer alteração que seja feita no mesmo arquivo, por mais que tenha sido em seções totalmente distintas do mesmo, gera um conflito que precisa ser tratado manualmente. Muitos desenvolvedores trabalham utilizando a técnica de realizar o Lock, ou travamento, no arquivo, criada para evitar conflitos.

Ao realizar um Lock, nenhum outro desenvolvedor poderá fazer alterações no arquivo. A principal vantagem da estratégia é que não existirão conflitos durante a execução de um merge.

### Opinião do Instrutor

Tudo tem seus pontos positivos e negativos, em geral chamados trade-offs (trocas) que tanto surgem no mercado da informática.

Um ponto negativo importante de perceber é que essa abordagem pode travar a equipe toda em um projeto, quando o mesmo arquivo precisar ser modificado ao mesmo tempo por mais desenvolvedores. No caso, o trade-off existente é dar menos dor de cabeça no futuro (resolvendo conflitos) contra uma menor velocidade de desenvolvimento (já que não haverão conflitos, nunca).

**Quando o merge automático falha?**

Os sistemas de controle de versão que permitem a mais de uma pessoa manipular o mesmo arquivo, ao mesmo tempo, geralmente permitem a realização do processo de merge, ou mescla. No *Git*, o merge pode ocorrer automaticamente, mas terá que ser feito manualmente quando houver um conflito. Em qual situação o merge precisa ser manual?

Top of Form

* Alternativa correta



Quando arquivos diferentes são alterados, em linhas diferentes

* Alternativa correta



Quando o mesmo arquivo é alterado nas mesmas linhas

Quando a mesma linha de um mesmo arquivo é alterada por dois desenvolvedores ao mesmo tempo, ocorre um conflito. Neste caso, é necessário que o desenvolvedor que for executar o merge decida como lidar com o conflito, seja escolhendo entre uma das alterações ou escolhendo a ordem em que as duas serão inseridas.

* Alternativa correta



Quando arquivos diferentes são alterados, mas na mesma linha

* Alternativa correta



Quando o mesmo arquivo é alterado em linhas diferentes

Bottom of Form

**Identificando conflitos nos arquivos**

Quais são os 3 caracteres utilizados pelo *Git* para identificar as áreas conflitantes de um arquivo?

Top of Form

* Alternativa correta



>, ! e <

* Alternativa correta



>, ? e <

* Alternativa correta



>, $ e <

* Alternativa correta



(, = e )

* Alternativa correta



>, = e <

O *Git* utiliza os caracteres >, < e =. Entre o < e o = fica o conteúdo antigo e entre o > e = fica o conteúdo novo.

**Minimizando conflitos**

O processo de merge muitas vezes pode ser confuso, demorado e trabalhoso. Justamente por isso, muitos desenvolvedores não gostam de fazê-lo e alguns deles inclusive possuem técnicas para minimizar a quantidade de conflitos que se gera em um projeto ou até mesmo simplificá-los. Quais técnicas você imagina que possam ser utilizadas para simplificar o processo de merge?

Top of Form

* Alternativa correta



Sincronização (pull) do repositório remoto somente ao fim da tarefa

* Alternativa correta



*Commits* com bastante conteúdo

* Alternativa correta



*Commits* com pouco conteúdo

Pequenos *commits*, com conteúdo reduzido, gerarão conflitos menores e menos trabalhosos.

* Alternativa correta



Sincronização (pull) frequente do repositório remoto

Ao sincronizar com bastante frequência o repositório, é mais fácil evitar conflitos ou minimizar o trabalho de mescla dos conflitos, pois seu repositório local estará sempre atualizado.

## Trabalhando com múltiplas branches

Como vimos no capítulo de branches, a branch local master é uma branch que mantém referência para a branch master remota (origin/master). O processo de merge, quando acontece, pode confundir muitos desenvolvedores a respeito dessa referência. Outro problema é a geração de mensagens de merge, poluindo o nosso log.

Justamente para evitar tais situações, **é uma boa prática** sempre trabalhar em uma branch local que não seja a master. Então, para isso, um próximo passo é criar uma branch para realizar todo o trabalho e apenas utilizar a master para sincronizar os commits com o repositório remoto (origin).

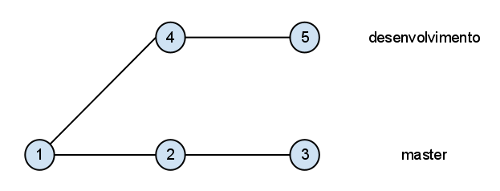
Então, um primeiro passo, que pode ser feito tanto pelo "jcfonsecagit" quanto pela "mmsoaresgit", é criar uma branch local chamada "desenvolvimento" e utilizá-la. Isso pode ser feito com o comando: git checkout -b desenvolvimento. Nesse momento, a branch atual é trocada para a que acabamos de criar, ou seja, não está mais na branch master.

A partir de então, todo os novos commits serão feitos na branch local desenvolvimento. Porém, como fazer para enviar esses commits para a branch master do repositório remoto, se a única que a rastreia é a nossa branch local master? Repare que agora precisamos levar os commits que fizemos na branch desenvolvimento, para a branch master.

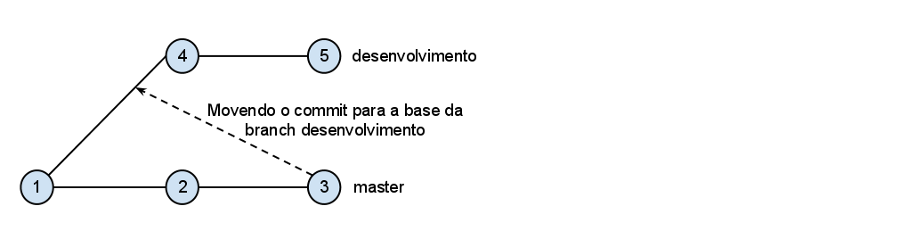
## Envio dos commits das branches locais para o master remoto

Considere que o usuário "jcfonsecagit" realizou 2 commits em sua branch local "desenvolvimento" e pretende sincronizar esses commits com a branch remota master. Para realizar essa tarefa, o primeiro passo, é verificar se a branch master não possui nenhuma nova atualização, ou seja, temos que fazer um checkout da master com git checkout master e, em seguida, realizarmos o git pullpara buscar as novas alterações feitas por outras pessoas naquele repositório. Considere também que existem alterações para baixar do repositório e, justamente por isso, nesse instante, temos um problema para resolver.

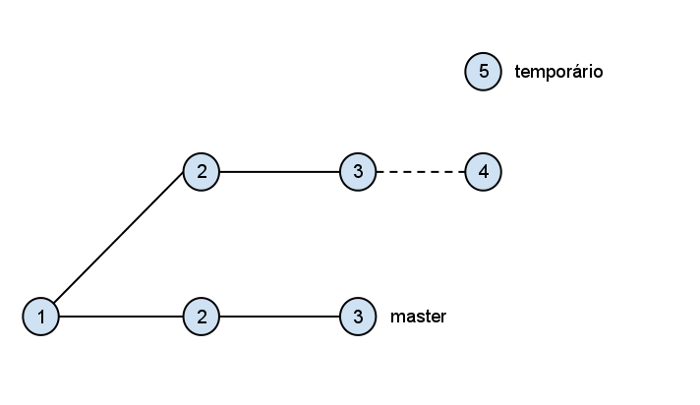
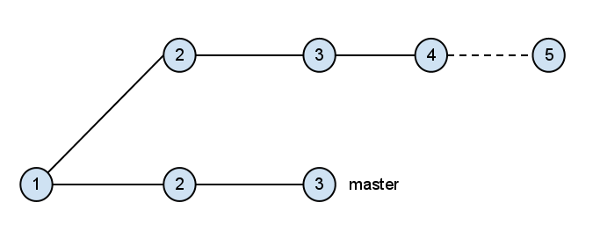
Temos a branch "desenvolvimento", que foi criada no commit 1. Porém, o último commit da master não é mais o commit 1 e sim, o 3.



Se simplesmente jogássemos o conteúdo da branch "desenvolvimento" para a master, poderíamos ter que tratar conflitos de vários commits ao mesmo tempo além do log dos commits ficarem confusos. Justamente para evitar essa situação, o Git possui o comando git rebase, onde podemos indicar qual é a nova base de commits que deve ser utilizada e resolver os conflitos commit por commit. No nosso caso, queremos que a branch "desenvolvimento" utilize como base de commits a "master" que acabamos de atualizar.



Para isso, temos que ir à branch "desenvolvimento" com o comando git checkout desenvolvimento e dizer que a nova base dela é o último commit que está na master, portanto git rebase master. Nesse instante, os commits feitos em "desenvolvimento" são movidos para uma branch temporária e o git atualiza a nova base de commits. Após essa atualização, o próprio Git traz de volta os commits que realizamos e os aplica sobre a nova base, um de cada vez.

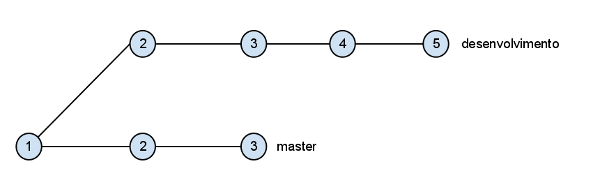
No caso, uma saída possível para a execução do git rebase master é a seguinte:

First, rewinding head to replay your work on top of it...

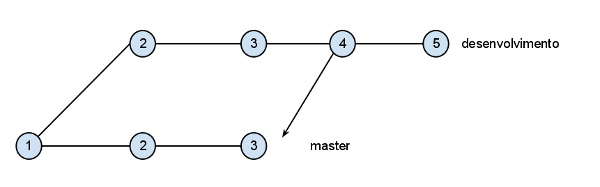
Applying: commit 1 de 2

Applying: commit 2 de 2

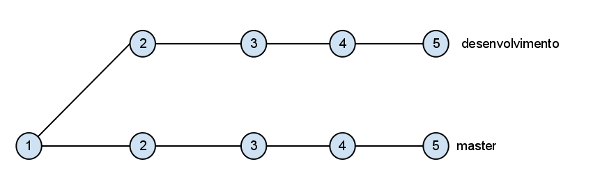
Nesse momento, todos os commits estão aplicados e organizados na branch "desenvolvimento".



Mas agora, é necessário levar esses commits para a branch "master". Repare que é um processo diferente do rebase, onde apenas queríamos trocar a base de commits utilizada. Agora, é necessário colocar apenas os commits novos na branch "master", porém, para isso, existe o comando git merge que move os commits de uma determinada branch para outra branch.



Para utilizá-lo, primeiro é preciso ir para a branch para a qual se quer levar os commits, no caso, a "master" (git checkout master). Em seguida, deve-se dizer para o comando git merge de qual branch virão os commits novos, que, no caso, é a branch "desenvolvimento". Para isso, basta executar git merge desenvolvimento.



Nesse instante, tem-se todos os novos commits na branch "master", prontos para serem enviados para o repositório remoto através do comando git push.

Note que agora, por mais que alterações tenham ocorrido no mesmo arquivo, ao visualizar o log, não existe mais aquele commit indicando que houve um merge. Muitos desenvolvedores que utilizam o Git usufruem desse ciclo para trabalhar em seus projetos, usando a branch local "master" apenas como uma transição entre a branch remota "master" e as outras branches locais.

## Mas o rebase pode falhar

O nosso último git rebase foi bem sucedido e não tivemos nenhum problema para enviar os commits para o nosso "master" remoto. No entanto, existem situações em que o git rebase pode falhar e, à primeira vista, trabalhar com essa situação pode não ser trivial.

Considere que os dois usuários, "jcfonsecagit" e "mmsoaresgit", estejam com seus repositórios locais totalmente atualizados, ou seja, sem mais nada para baixar. Nesse momento, o arquivo proposta\_1.html, que ambos possuem, está com o seguinte conteúdo:

<html>

<head>

<title>Proposta 1 para homepage da empresa</title>

</head>

<body>

<h1>Cabeçalho do site</h1>

<div>

Isso aqui é o conteúdo da página

</div>

<h1>Rodapé do site</h1>

</body>

</html>

Em seguida, "mmsoaresgit" **faz o git push** de 3 commits. O primeiro commit altera o title para, em vez de usar o número 1, utilizar o número por extenso, ou seja, "um". O segundo commit muda a palavra "site" no cabeçalho para "sistema". E o terceiro commit muda a palavra "site" no rodapé também para "sistema". Por fim, "mmsoaresgit" realiza o git push de suas alterações, enviando-as para o repositório remoto. Com isso, "mmsoaresgit" gerou o seguinte log de commits:

commit 9ee6a2e5344ff14ba38461ab65f51927bc2d7096

Author: Maria Soares <mmsoaresgit@gmail.com>

Date: Fri Dec 30 14:41:33 2011 -0200

troca de site para sistema no rodape

commit 658ed785d5e5c933d6ccead69b5d1801dd52e331

Author: Maria Soares <mmsoaresgit@gmail.com>

Date: Fri Dec 30 14:41:12 2011 -0200

troca de site para sistema no cabecalho

commit 12ec2eb6cba5e1021e8ed609ac26188397dc8ed2

Author: Maria Soares <mmsoaresgit@gmail.com>

Date: Fri Dec 30 14:40:47 2011 -0200

trocando de 1 para um no title

O estado final em que "mmsoaresgit" deixou o arquivo é o seguinte:

<html>

<head>

<title>Proposta um para homepage da empresa</title>

</head>

<body>

<h1>Cabeçalho do sistema</h1>

<div>

Isso aqui é o conteúdo da página

</div>

<h1>Rodapé do sistema</h1>

</body>

</html>

O usuário "jcfonsecagit", trabalhando em seu computador, decide realizar alterações na mesma página numa branch local chamada "desenvolvimento", conforme vimos anteriormente. Contudo, a sua branch master não está sincronizada com a do repositório remoto. Ele altera a mensagem do rodapé para "Copyright - Caelum 2012" e realiza o commit de sua alteração, localmente, na branch "desenvolvimento". Ele realiza um outro commit alterando o "1" do title para "I" em algarismos romanos.

Agora, ele precisa enviar seus commits para o repositório remoto para que todos tenham acesso ao que ele acabou de fazer. Para tanto, antes de fazer o git push, ele vai para a branch "master" e realiza o git pull para trazer os novos commits que "mmsoaresgit" realizou. Para juntar as alterações que a "mmsoaresgit" realizou com as suas, "jcfonsecagit" precisa alterar a base de commits da branch "desenvolvimento", para incluir os novos commits que foram baixados. Para isso, ele vai à branch "desenvolvimento", com o git checkout desenvolvimento, e realiza o rebase, indicando que a base de commits é o que está na branch "master" com o comando git rebase master.

Como explicado anteriormente, os commits que o "jcfonsecagit" realizou serão colocados em um lugar temporário e a base da branch "desenvolvimento" ganhará os 3 novos commits. Em seguida, o rebase faz uma verificação para saber se os commits feitos por "jcfonsecagit" possuem algum conflito com algum dos commits que entraram na nova base, um de cada vez.

Nesse instante, o Git perceberá que a troca do rodapé que o "jcfonsecagit" realizou, adicionando o Copyright, conflitará com a alteração no rodapé que "mmsoaresgit" também fez e mostrará a seguinte mensagem:

First, rewinding head to replay your work on top of it...

Applying: colocando mensagem de copyright no rodapé

Using index info to reconstruct a base tree...

Falling back to patching base and 3-way merge...

Auto-merging proposta\_1.html

CONFLICT (content): Merge conflict in proposta\_1.html

Failed to merge in the changes.

Patch failed at 0001 colocando mensagem de copyright no rodapé

When you have resolved this problem run "git rebase --continue".

If you would prefer to skip this patch, instead run "git rebase --skip".

To restore the original branch and stop rebasing run "git rebase --abort".

Novamente, apesar de a mensagem ser grande, fica mais fácil a sua interpretação. O Git primeiro tirou o commit do "jcfonsecagit" para alterar a base de sua branch, utilizando os novos commits da outra usuária, "mmsoaresgit". Após a troca da base de commits, o primeiro commit realizado pelo "jcfonsecagit" é aplicado. Porém, ele conflita com as alterações da outra usuária. Repare que o próprio Git percebe o conflito e tenta fazer o processo de "merge" automaticamente, como aprendemos no começo do capítulo. Mas, dessa vez, ele não consegue fazer, pois houve alteração no mesmo trecho de código.

Nesse momento, o Git nos deixa resolver o conflito existente no rebase, de forma manual. Para isso, o próprio Git nos coloca em um "ambiente temporário", no caso, uma branch temporária, só para resolvermos o conflito. Nesse momento, se fosse executado o comando git branch, teríamos o seguinte resultado:

\* (no branch)

desenvolvimento

master

Note a branch chamada "(no branch)" em que estamos. É nessa branch em que deveremos resolver o conflito. Nesse momento, temos 3 opções, que o próprio Git nos avisou anteriormente: continue, skip e abort.

Uma primeira opção é abortar a resolução do conflito e voltar atrás no rebase, ou seja, voltar ao ponto que se estava antes de realizar o rebase. Para isso, bastaria executar o comando git rebase --abort.

Uma segunda opção é descartar o seu commit atual que gera o conflito, o que pode ser realizado com o comando git rebase --skip.

Por fim, caso queiramos resolver o conflito, seja para ficar com nossa alteração, ou seja para fazer alguma outra modificação, é necessário permanecer nessa branch temporária, e corrigir os arquivos conflitados. O primeiro passo para descobrir quais arquivos devem ser mexidos é executar o comando git status:

# Not currently on any branch.

# Unmerged paths:

# (use "git reset HEAD <file>..." to unstage)

# (use "git add/rm <file>..." as appropriate to mark resolution)

#

# both modified: proposta\_1.html

#

no changes added to commit (use "git add" and/or "git commit -a")

Note que a primeira linha já nos diz que não estamos em uma branch válida e que também temos "merge" por fazer. O arquivo indicado pela sentença "both modified", ou seja, que "ambos modificaram", é o arquivo que temos que corrigir o conflito. Abrindo-o, encontraremos o trecho conflitante, demarcado como aprendemos anteriormente:

<<<<<<< HEAD

<h1>Rodapé do sistema</h1>

=======

<h1>Copyright - Caelum 2012</h1>

>>>>>>> mudando o rodapé para utilizar copyright

Agora, basta deixar o código que deve permanecer após o merge. No caso, vamos deixar a linha do Copyright. Assim, o conteúdo final do arquivo será:

<html>

<head>

<title>Proposta um para homepage da empresa</title>

</head>

<body>

<h1>Cabeçalho do sistema</h1>

<div>

Isso aqui é o conteúdo da página

</div>

<h1>Copyright - Caelum 2012</h1>

</body>

</html>

Agora que já sabemos qual é o estado que queremos manter no repositório, basta adicionarmos o nosso arquivo no Index do Git, executando o comando git add proposta\_1.html. Nesse momento, se executar o comando git status, teremos a seguinte saída:

# Not currently on any branch.

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: proposta\_1.html

#

Note que o proposta\_1.html **não** está mais marcado como "both modified". Nesse ponto, quando todos os arquivos conflitantes já estiverem nesse estado, é possível continuar o rebase através do comando git rebase --continue para juntar os outros commits. Novamente, teremos um outro conflito:

Applying: colocando mensagem de copyright no rodapé

Applying: trocando de 1 para I no title

Using index info to reconstruct a base tree...

Falling back to patching base and 3-way merge...

Auto-merging proposta\_1.html

CONFLICT (content): Merge conflict in proposta\_1.html

Failed to merge in the changes.

Patch failed at 0002 title

When you have resolved this problem run "git rebase --continue".

If you would prefer to skip this patch, instead run "git rebase --skip".

To restore the original branch and stop rebasing run "git rebase --abort".

Dessa vez, vamos ignorar a alteração feita por "jcfonsecagit" e manter o que a "mmsoaresgit" fez com o comando git rebase --skip, excluindo esse último commit do rebase. Como não há mais nenhum commit que "jcfonsecagit" realizou, o processo de rebase está completo.

Por fim, precisamos atualizar a nossa branch master com os novos commits. Para isso, mudamos para a branch master com o comando git checkout master e juntamos os novos commits com git merge desenvolvimento. Agora, a nossa branch master está pronta enviar as atualizações para o repositório remoto (git push).

**Qual o nome da branch temporária para tratar os conflitos durante o rebase?**

Quando estamos no meio do processo de rebase, e houve um conflito que devemos tratar manualmente, como se chama a branch temporária para a qual somos movidos?

Top of Form

* Alternativa correta



(system branch)

* Alternativa correta



(temp branch)

* Alternativa correta



(no branch)

A branch (no branch), que dá a entender que você não está em branch nenhuma, é apenas temporária, criada pelo Git para que possamos resolver o conflito.

* Alternativa correta



(default branch)

Bottom of Form