

TREINAMENTO

SUCTI11



PRIMEIROS PASSOS COM GIT E GITLAB

O CURSO

- Módulo Básico
- Ainda em construção
- Está totalmente franqueado



ESCOPO E NÃO ESCOPO DO TREINAMENTO

4 ESCOPO DESTE MÓDULO

- estrutura do Git
- estados da operação do Git
- estados dos arquivos no Git
- git init
- git config
- .gitignore
- comandos essenciais
- comandos de reversão
- tags
- branches, merges e rebase
- Git Lab, recursos e políticas



NÃO ESCOPO DESTE MÓDULO

- interfaces gráficas para o Git (GUI)
- integração do Git com outras ferramentas
- webhooks
- detalhamento de Rebase, pick e squash
- detalhamento do branching model
- detalhamento do git stash
- Git LFS
- ClearCase, subversion e migração



CONTROLE DE VERSÃO DO SOFTWARE

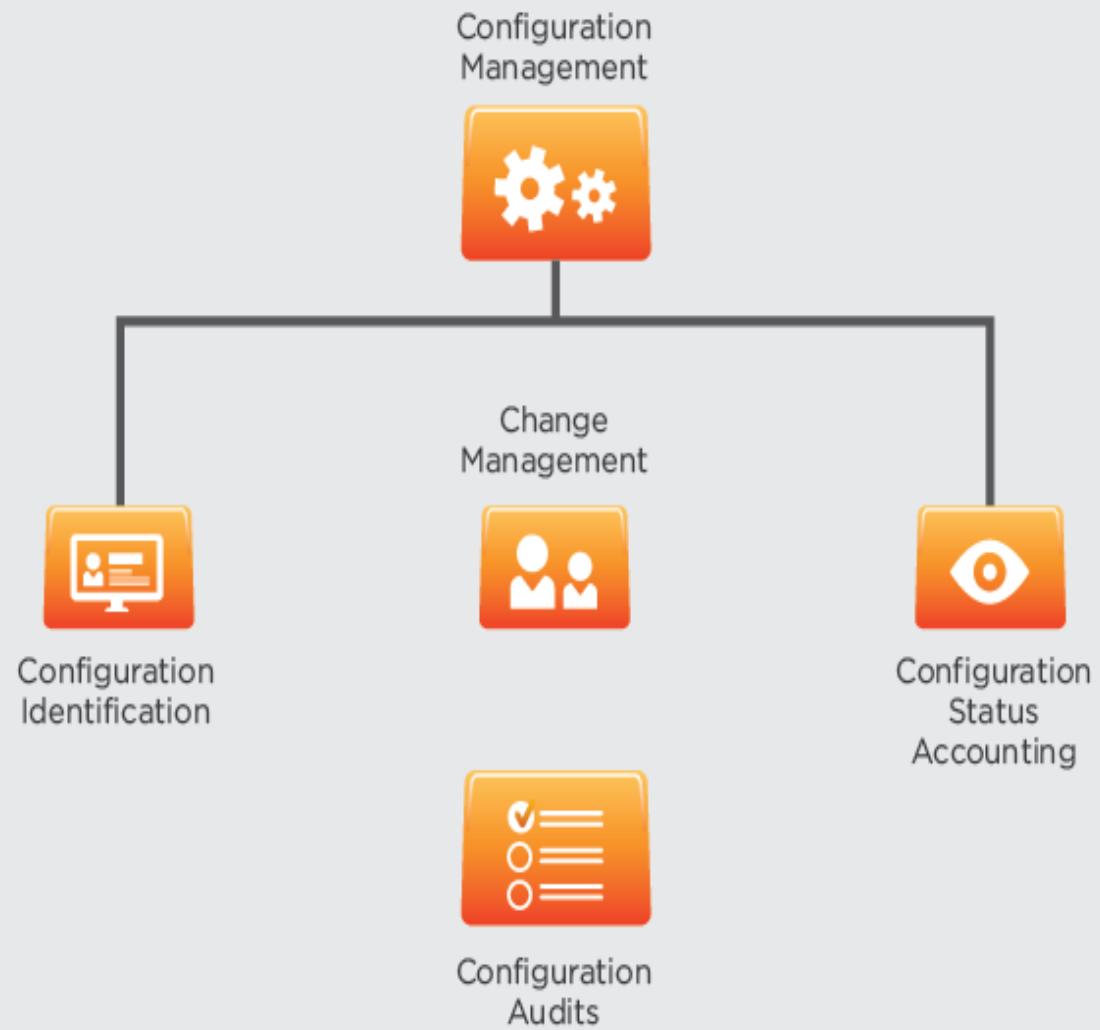
DEFINIÇÕES

- O git basicamente é uma ferramenta para controle de versões do seu projeto de desenvolvimento de software (SCM).
- O GitLab é um repositório que nos permite hospedar projetos versionados com a tecnologia git, colaborar com outros desenvolvedores e sincronizar o nosso projeto.



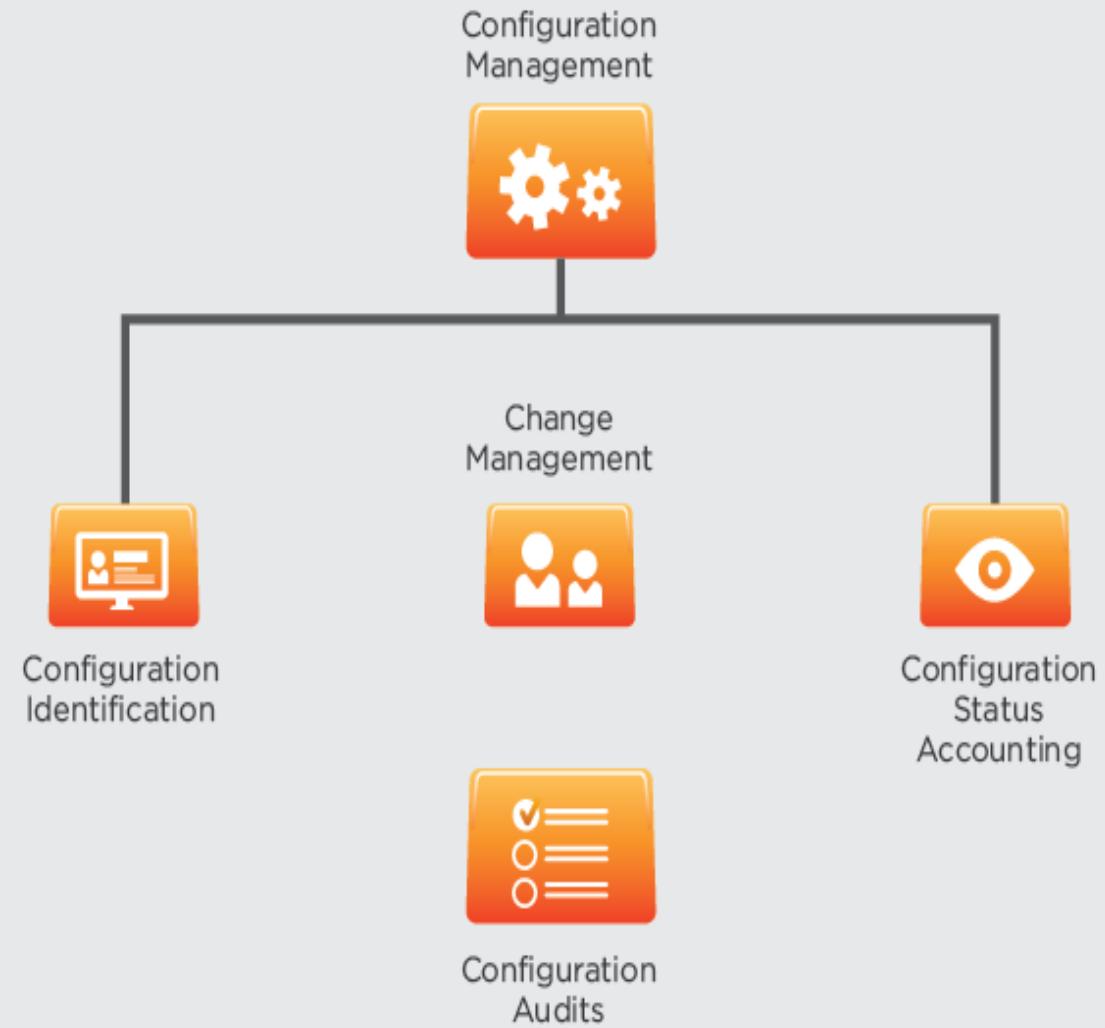
O QUE QUEREMOS SABER

- O que queremos saber durante o desenvolvimento de software?
 - O que mudou
 - Quando mudou?
 - Por que mudou?
 - Quem fez a mudança?
 - Podemos reproduzir essa mudança?



O QUE QUEREMOS TER

- O que queremos ter durante o desenvolvimento de software em um controlador de versão?
 - Identificação
 - Auditoria
 - Segurança
 - Qualidade
 - Tecnologia universal
 - Condições para automatização
 - Repositório seguro da cópia oficial do código-fonte



UM POUCO SOBRE O GIT

COMO SURGIU

- O Git é um software para controle de versões criado por **Linus Torvalds** em 2005.
- A ideia de seu desenvolvimento surgiu quando Torvalds e os desenvolvedores do kernel Linux optaram por não utilizar mais o software proprietário BitKeeper, após Larry Macvoy (detentor dos direitos autorais do BitKeeper) remover o acesso gratuito ao software.
- Torvalds tinha a intenção de desfrutar de um sistema distribuído que fosse rápido, fluído e que funcionasse de maneira similar ao BitKeeper.
- Sem muitas opções, Torvalds decidiu desenvolver o próprio software controlador de versões e assim nascia o Git.
- o Git foi desenvolvido e projetado por Torvalds para auxiliar no desenvolvimento do kernel do Linux, porém, com a ênfase em velocidade e praticidade, aliadas à licença GNU.



PROJETO GIT

A comunidade então resolveu desenvolver o próprio versionador baseado nas lições aprendidas no BitKeeper.

As premissas do novo projeto eram:

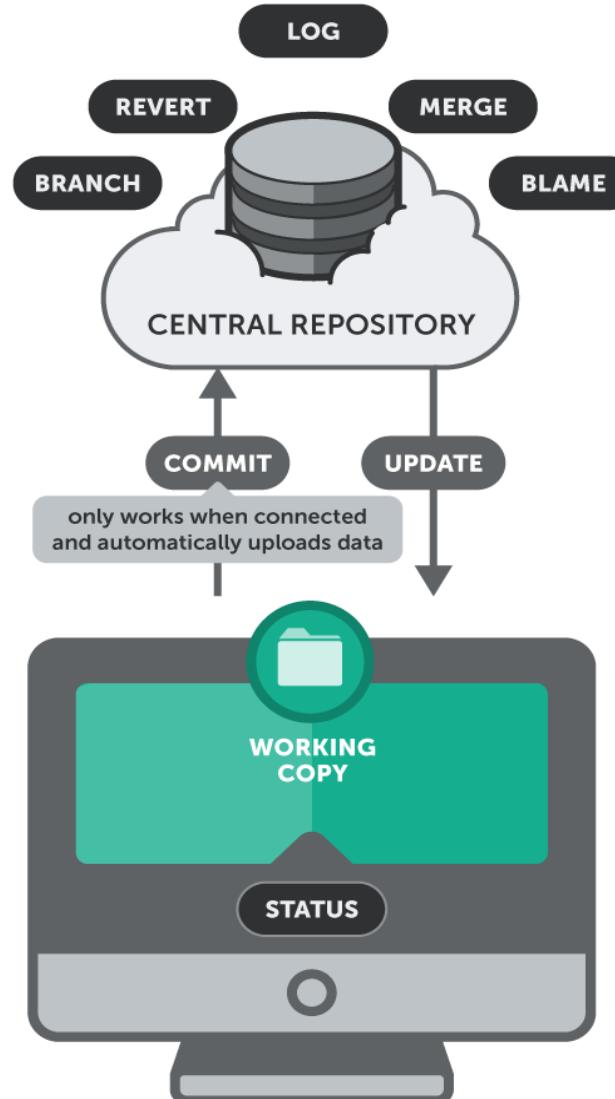
- Design simples;
- Velocidade;
- Suporte robusto a desenvolvimento não linear (milhares de branches paralelas) ;
- Totalmente distribuído;
- Capaz de lidar eficientemente com grandes projetos como o kernel do Linux (velocidade e volume de dados).



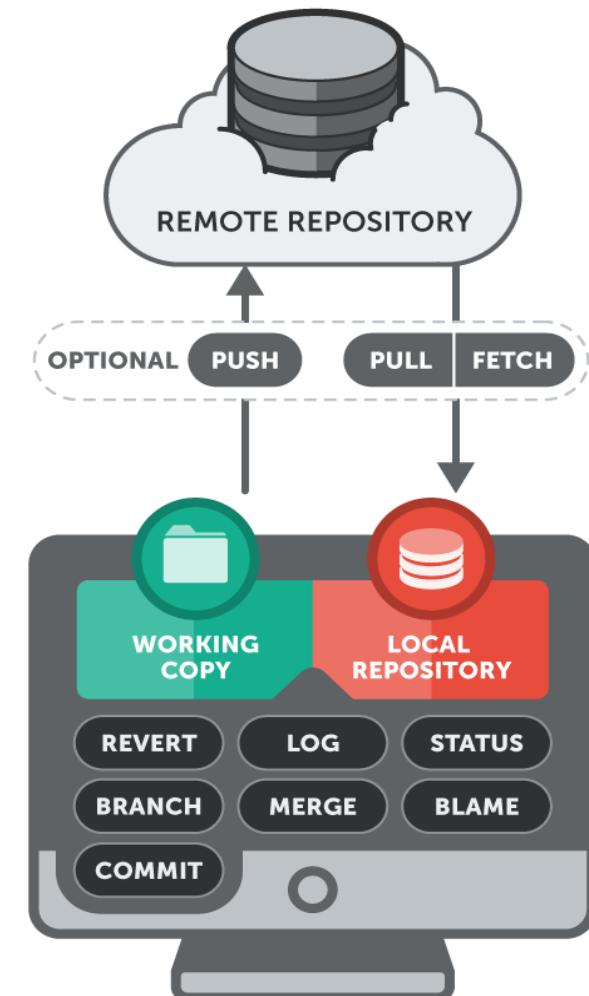
CONTROLE DE VERSÃO DESCENTRALIZADO

- Quase todas operações do Git são feitas de forma local.
- Desta forma, não existem, ao utilizar o Git, problemas que normalmente acontecem quando se usa um controlador de versão que depende muito de uma rede como, por exemplo, problemas no tráfego de arquivos ou latências de rede.
- Como o Git opera em sua maioria localmente, quase que todas operações são efetuadas de forma instantânea.

SUBVERSION



GIT





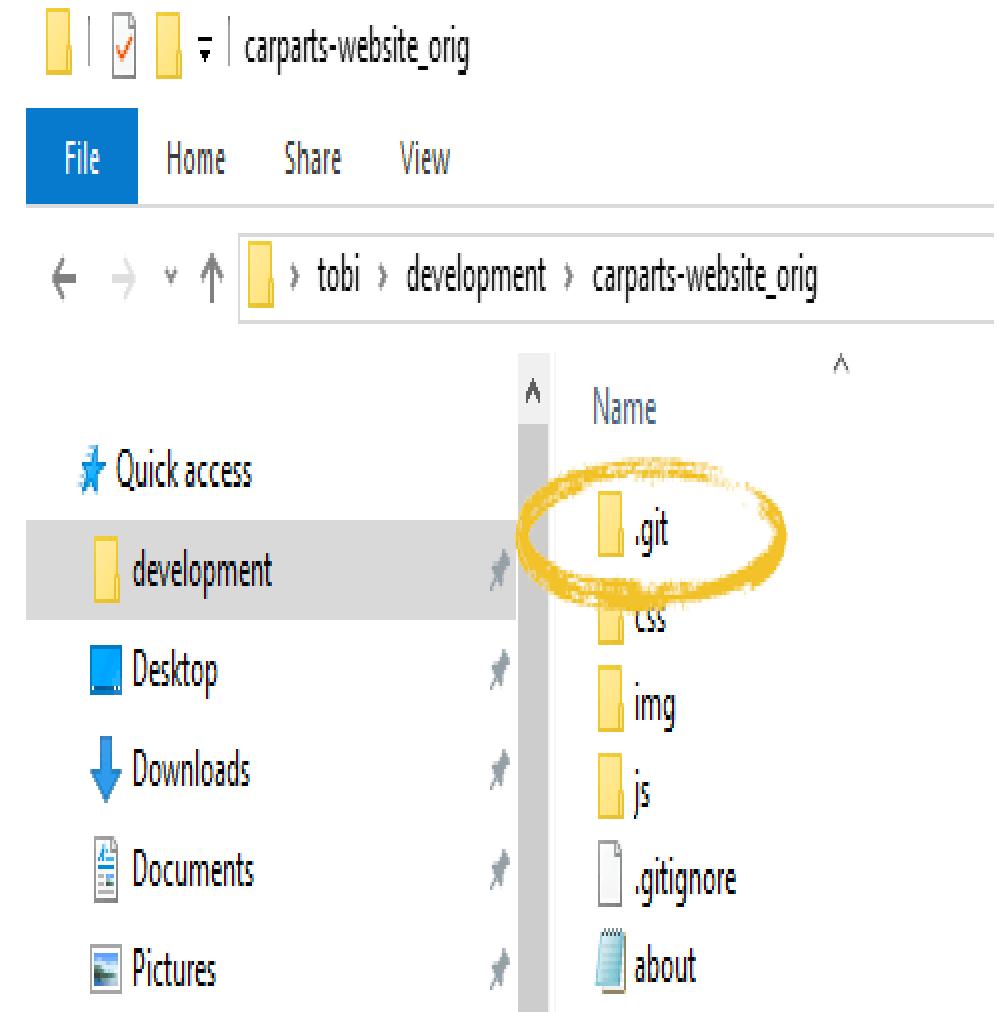
GIT INIT

REPOSITÓRIO GIT LOCAL

Após selecionar uma pasta de sua preferência como um diretório para trabalhar seu projeto/arquivos você deve inicializar o seu repositório git.

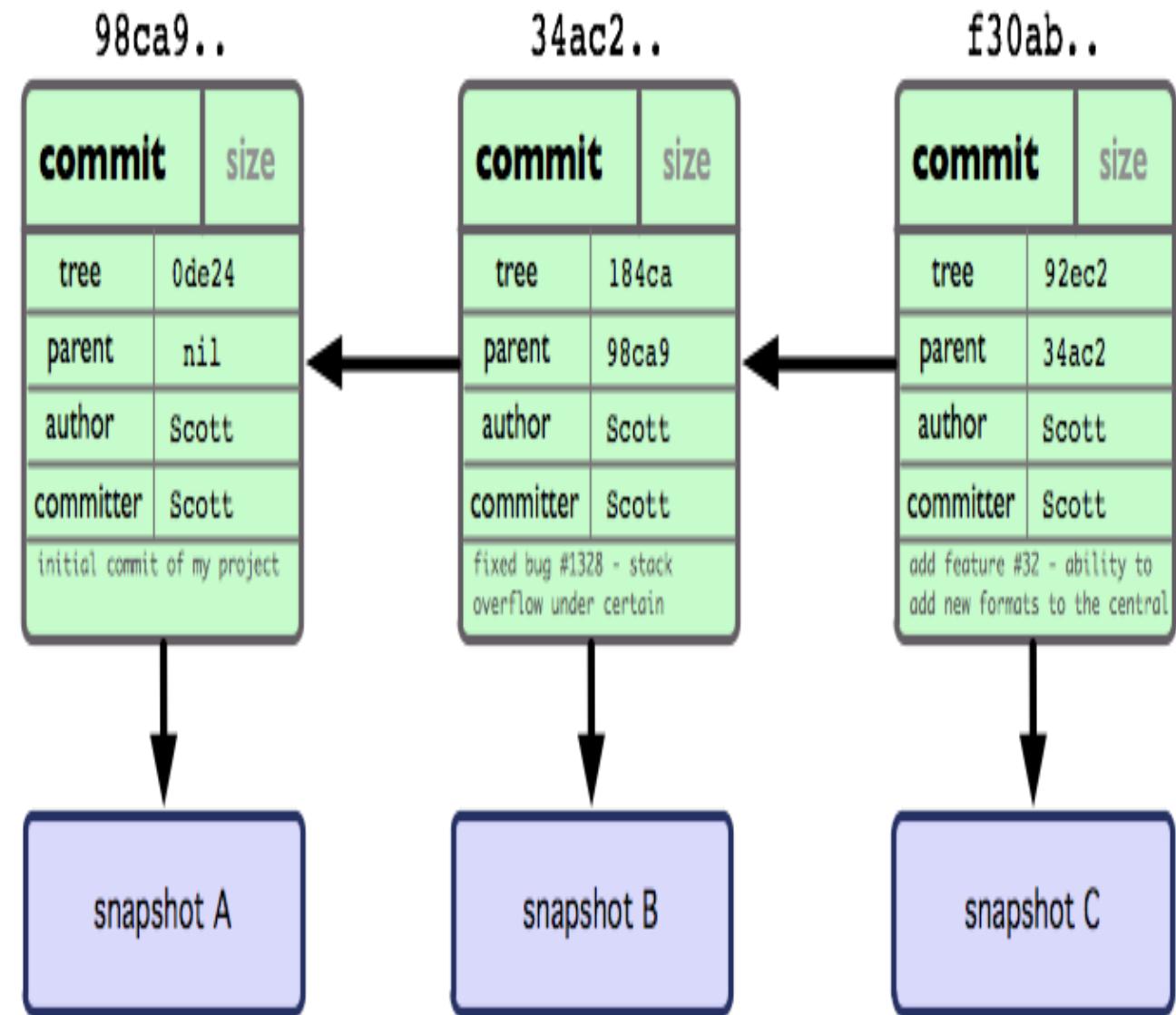
Após o comando `$ git init` no repositório do seu projeto se inicia toda a tecnologia git para **controle de versão** e uma pasta no diretório do seu projeto é criada: o repositório `.git`

É neste repositório onde ficam todas as referências e objetos do git.



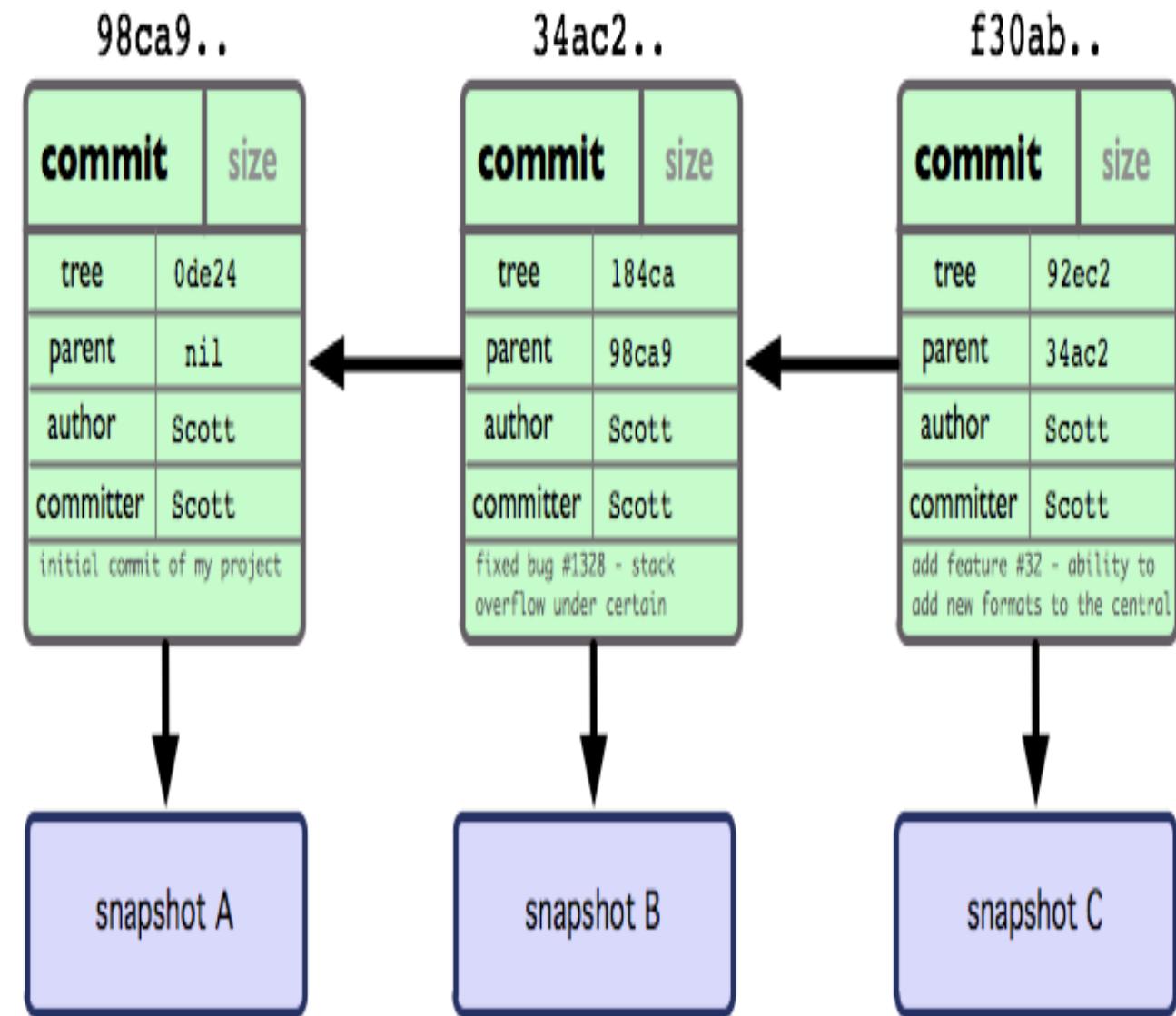
SNAPSHOTS

- A forma como o Git manuseia as informações se difere em muito dos outros controladores de versão.
- Ao invés de armazenar uma lista contendo as mudanças efetuadas nos arquivos, como a maioria dos controladores, o Git registra “momentos”, os quais são chamados de snapshots, contendo uma espécie **foto dos arquivos** que serão armazenados em blobs.
- Cada commit que o usuário faz no seu diretório Git, é como se fosse feita uma captura **de todos os arquivos presentes no diretório e criada uma referência para essas capturas**.



INTEGRIDADE

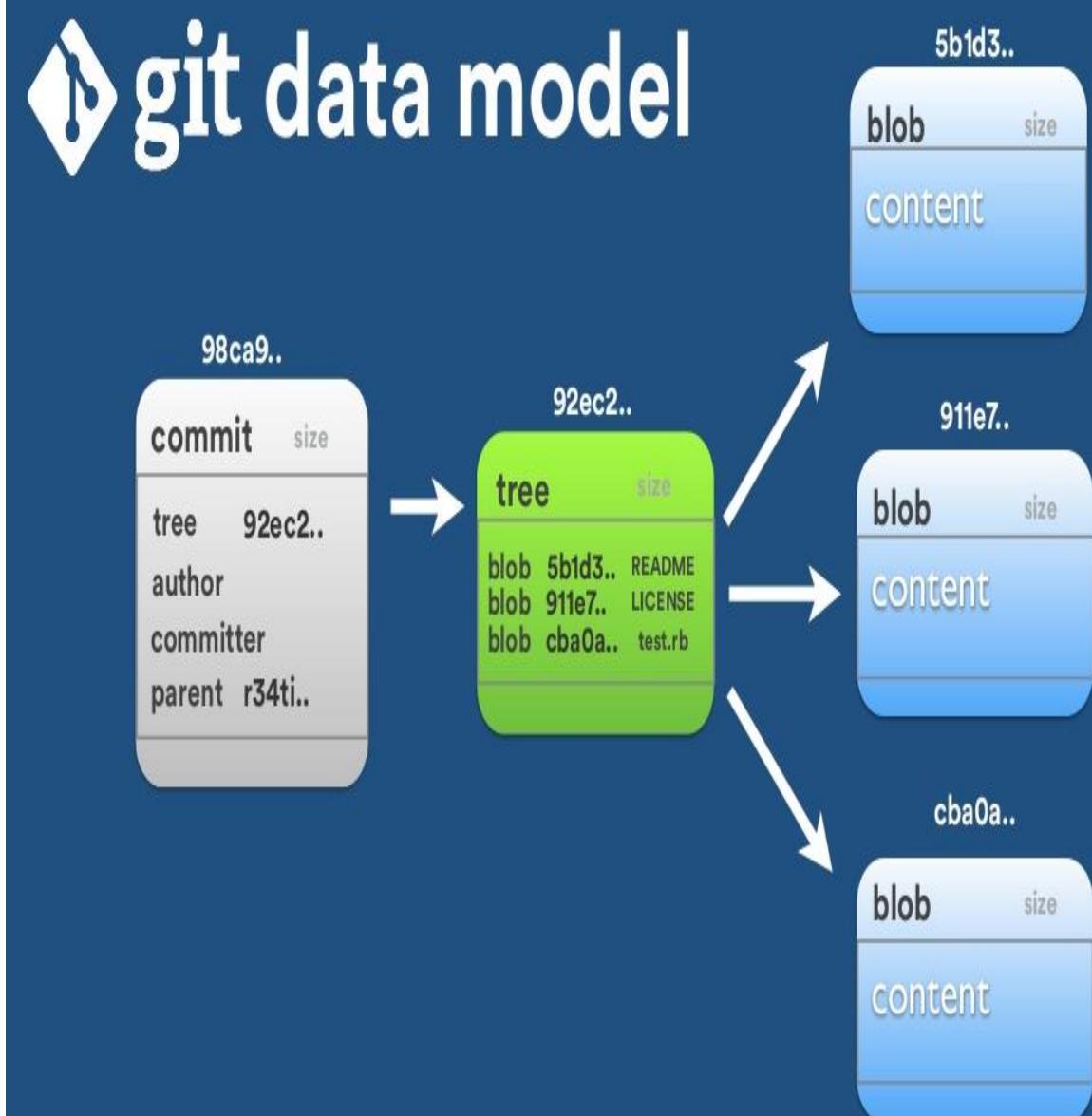
- A integridade das informações de cada projeto são asseguradas por um **checksum** que gera um código.
- Dessa forma, o Git garante um item fundamental de sua filosofia, que trata da integridade dos seus arquivos.
- Com o **código hash**, o Git assegura que o usuário não irá perder informação em movimentação ou corromper arquivos sem que ele mesmo perceba.



ELEMENTOS DO GIT

O .git armazena diferentes tipos de elementos:

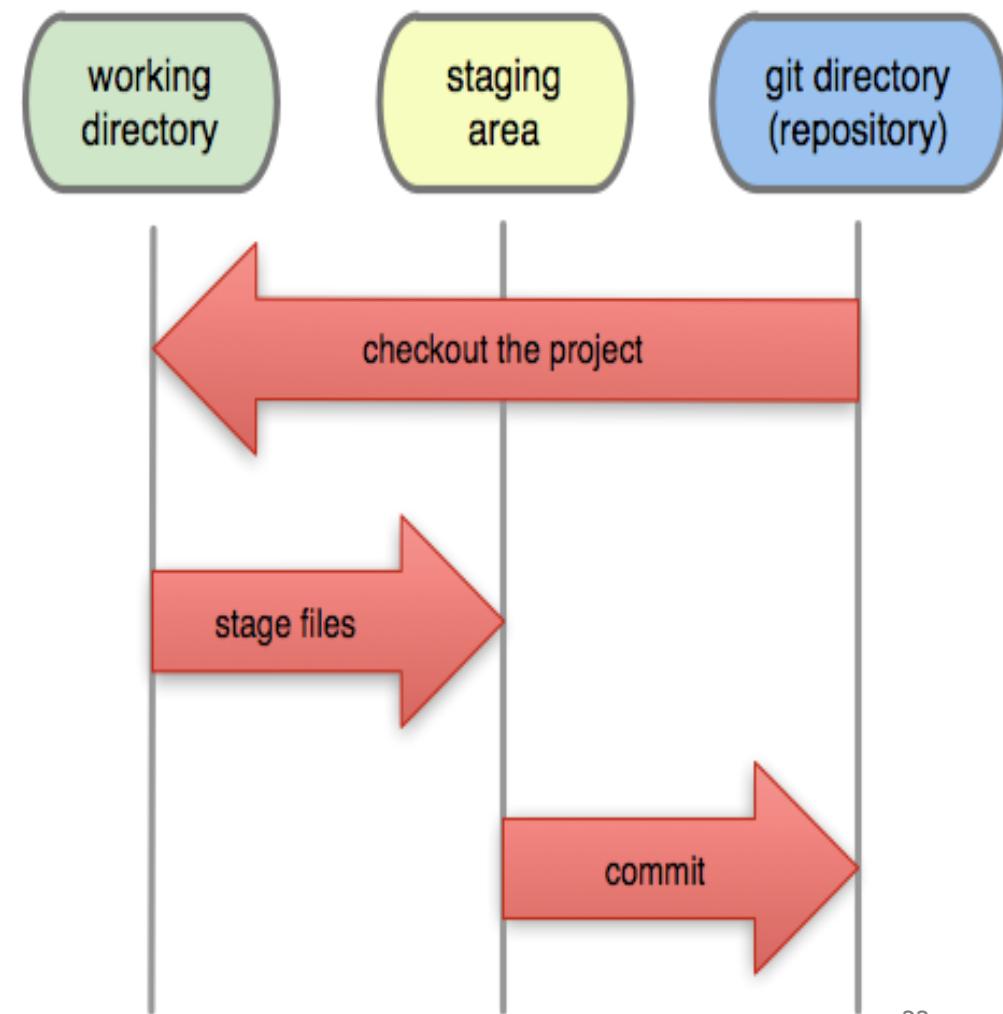
- **Commit:** contém o hash do elemento “árvore”, o hash do commit pai, o autor, o committer, a data e a mensagem.
- **Árvore:** contém caminhos de diretórios dos elementos do commit. É o elemento que permite rastrear todos os objetos de um commit.
- **blob :** Este é um elemento incremental do tipo "blob" e contém cada instantâneo de um determinado arquivo:
- **tag anotada:** Elemento que armazena informações para identificação ou rastreabilidade de determinados instantâneos que auxiliam no controle de versões do projeto.



LOCAIS DE OPERAÇÃO

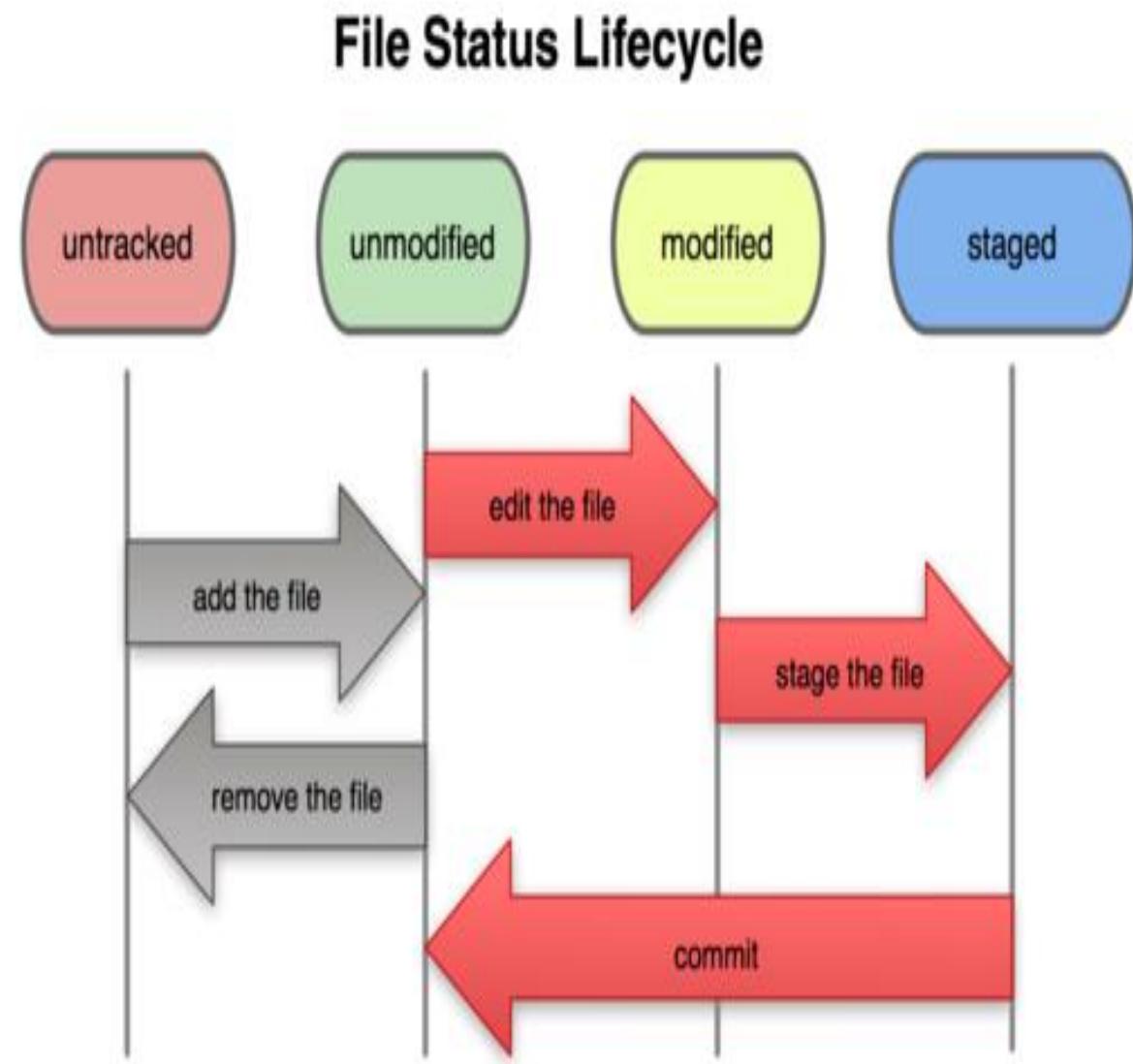
- Ao iniciar o git, os arquivos ainda não têm suas versões controladas. Pode-se trabalhar a vontade nos arquivos, dentro do processo de criação e desenvolvimento em uma **área de trabalho**. Os arquivos ainda não estão sendo monitorados .
- Ao decidir que seus arquivos já estão **prontos** para uma versão do seu processo de desenvolvimento, deve-se colocar estes arquivos em um estado “preparado”. Estes arquivos então irão para uma **área de seleção**. Esta área existe para que se possa **adicionar aos poucos os arquivos considerados prontos**.
- Quando você encerrar este momento você pode gerar uma foto. A geração desta “foto” é o commit e seus elementos ficam no **repositório local (.git)**.

Local Operations



ESTADOS DOS ARQUIVOS

- Os arquivos podem estar **untracked**, quando ainda não estão sendo monitorados ou **modified**, quando os arquivos foram modificados e, neste caso, sua versão ainda não está sendo monitorada. Arquivos novos ou suas versões não monitorados estão na área de trabalho
- Arquivos **unmodified** permanecem conforme o último commit e, portanto, não são visíveis. (não confundir área de trabalho com a pasta do projeto git).
- Agora, o arquivo se encontra **staged** quando o usuário considerou-o **pronto** para compor uma versão nova do sistema e o adicionou em uma área de seleção.
- Os dados se encontram **committed** quando já estão assegurados na base de dados do diretório Git (.git).



QUANDO DEVO DAR COMMIT?

O commit não é para ser usado a todo momento ou ciclo diário, seja para salvar “o que se está fazendo” ou realizar um backup.

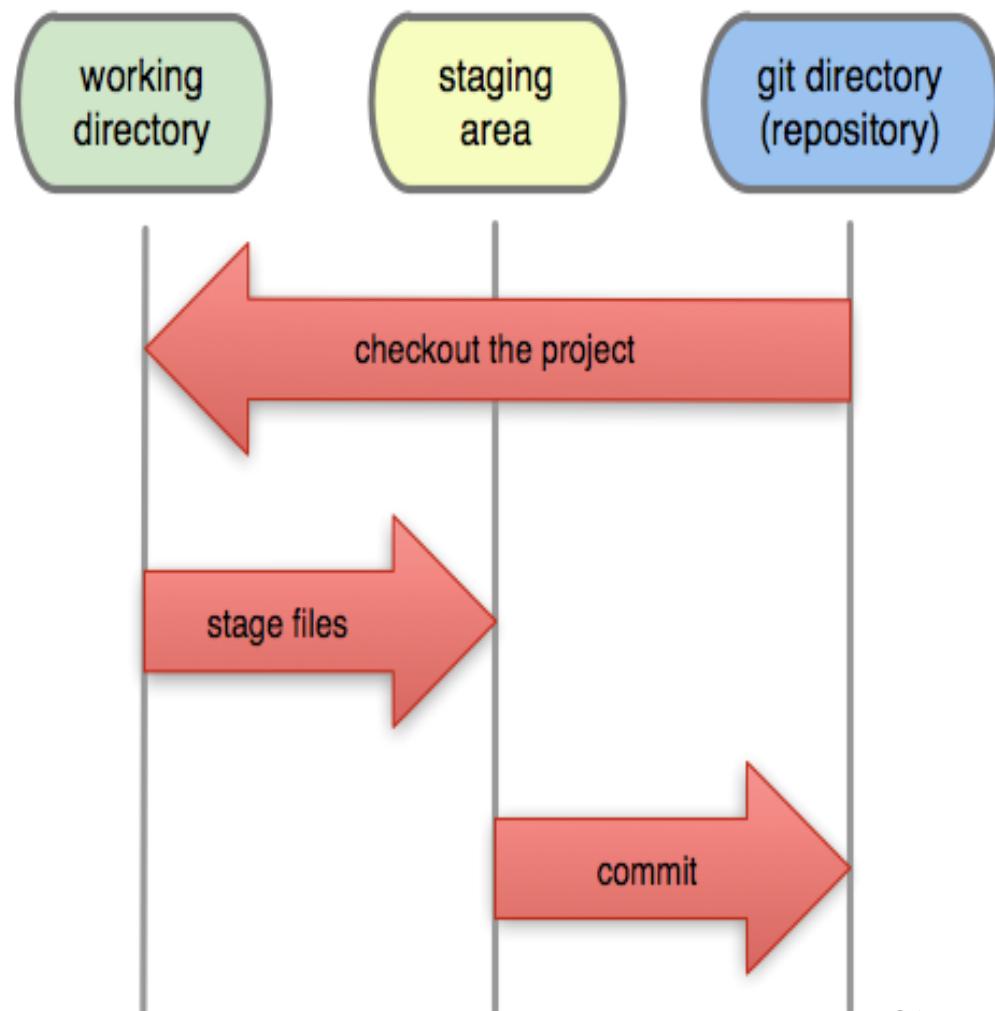
Quando se deve realizar um commit?

Entre outras situações:

- Quando o trabalho já está **pronto** e pode ser entregue em um repositório para ser conhecido, evoluído por um par (local) ou outros desenvolvedores do projeto (remoto) e integrado a branches superiores (merge request).
- Quando sua tarefa (ou item de trabalho) foi **finalizada**, em um processo que pode durar horas ou dias, dependendo do caso.

Commits realizados sem critério aumentam a quantidade de elementos no seu repositório e poluem a sua branch!

Local Operations



PREPARANDO O AMBIENTE PARA O GIT

GIT NO WINDOWS

Para instalar o Git no Windows deve acessar o link: <https://gitforwindows.org/>

The screenshot shows the official website for Git for Windows. At the top, there's a navigation bar with the "git para windows" logo, the version "VERSÃO 2.19.1", and links for "PERGUNTAS FREQUENTES", "REPOSITÓRIO", and "LISTA DE DISCUSSÃO". Below the navigation is a large orange diamond-shaped graphic containing a white stylized "git" logo (three nodes connected by lines). To the right of the graphic, the text reads: "Nós trazemos o incrível **Git** SCM para o Windows". Below this text are two blue buttons: "Faça o download do" and "Contribute".

git para windows

VERSÃO 2.19.1

PERGUNTAS FREQUENTES REPOSITÓRIO LISTA DE DISCUSSÃO

Nós trazemos o incrível **Git** SCM para o Windows

Faça o download do

Contribute

EXERCÍCIO

Instale o Git na sua máquina!

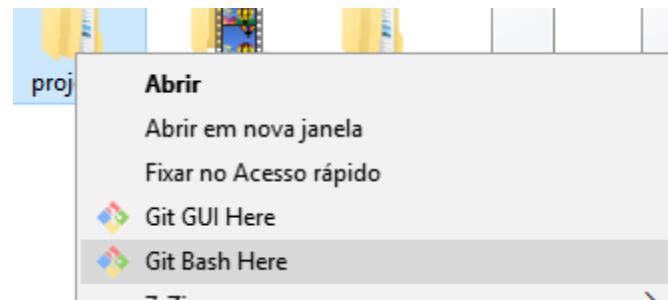


RECURSOS DO GIT

O Git for Windows se concentra em oferecer um conjunto nativo e leve de ferramentas que traz o conjunto completo de recursos do [Git SCM](#) para o Windows, ao mesmo tempo em que fornece interfaces de usuário apropriadas para usuários experientes e novatos do Git.

Git BASH

O Git for Windows fornece uma emulação BASH usada para executar o Git a partir de linhas de comando (similar ao LINUX).



```
MINGW32~/git
Welcome to Git (version 1.8.3-preview20130601)

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

Bacon@BACON ~
$ git clone https://github.com/msysgit/git.git
Cloning into 'git'...
remote: Counting objects: 177468, done.
remote: Compressing objects: 100% (52057/52057), done.
remote: Total 177468 (delta 133396), reused 166093 (delta 123576)
Receiving objects: 100% (177468/177468), 42.16 MiB | 1.84 MiB/s, done.
Resolving deltas: 100% (133396/133396), done.
Checking out files: 100% (2576/2576), done.

Bacon@BACON ~
$ cd git

Bacon@BACON ~/git (master)
$ git status
# On branch master
nothing to commit, working directory clean

Bacon@BACON ~/git (master)
$
```

A screenshot of a terminal window titled 'MINGW32~/git'. It shows the user cloning the 'git' repository from GitHub. The terminal then changes directory to the cloned repository and runs 'git status', which indicates there is nothing to commit and the working directory is clean. The terminal window has a standard Windows title bar and scroll bars.

ALGUNS COMANDOS NO BASH

| COMANDOS | AÇÕES |
|----------|----------------------------------------------------------|
| ls /dir | lista os arquivos |
| cd | navega entre diretórios |
| clear | limpa a tela |
| cat | exibe o conteúdo do arquivo |
| mkdir | cria diretório |
| cp | copiar arquivo |
| mv | mover ou renomear arquivo/diretório |
| rm | apaga arquivo |
| touch | cria um arquivo |
| diff | apresenta a diferença entre dois arquivos |
| q | sair de uma execução ou parar a execução de uma listagem |
| :wq | salva e sai do arquivo |

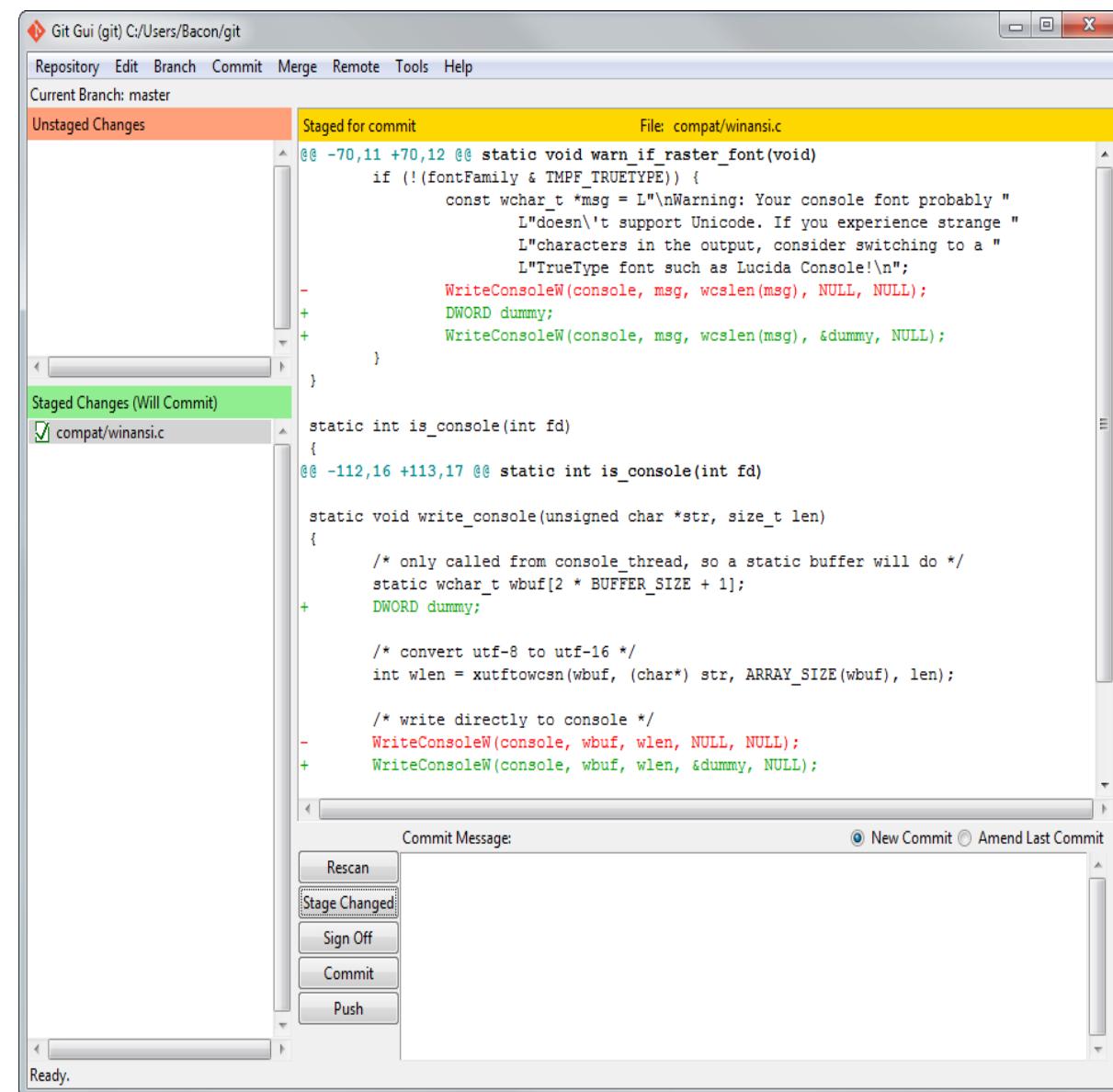
RECURSOS DO GIT

GUI do Git

Como os usuários do Windows normalmente esperam interfaces gráficas, o Git for Windows também fornece a Git GUI, uma poderosa alternativa ao Git BASH, oferecendo uma versão gráfica de praticamente todas as funções de linha de comando do Git, bem como ferramentas de diff visual.

Integração Shell

Shel é uma interação na interface do usuário para o acesso a serviços do kernel, como uma camada. Neste caso, basta clicar com o botão direito do mouse em uma pasta no Windows Explorer para acessar o BASH ou GUI.



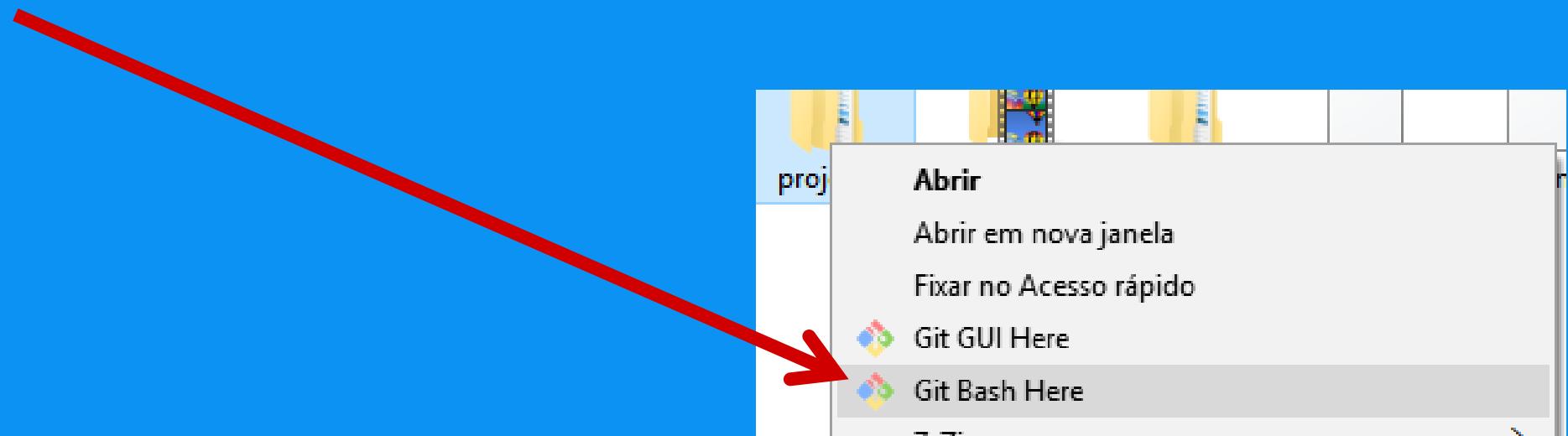
GIT CONFIGURAÇÃO

A large, dark grey filing cabinet with multiple drawers. Several drawers are open, revealing stacks of papers and files inside. The cabinet is set against a plain, light-colored wall.

Criando um repositório local...

EXERCÍCIO

Escolha uma pasta e selecione a opção “Git Bash Here” e crie seu próprio ambiente Git!



GIT CONFIG

O git apresenta diferentes tipos de comandos ou funcionalidades, com diversas opções.

Pode-se configurar o ambiente Git com o comando **git config**.

Este comando permite ler, definir ou redefinir variáveis de configuração que controlam todos os aspectos de como o Git opera.

\$ git config <opção> <variável> <valor>

O git também apresenta um recurso de usabilidade na sua tela “preta” que é uso de cores para identificação de situações como o verde para adição e vermelho quando algo foi deletado. Para ativar use:

\$ git config --global color.ui true

Git Cheat Sheet

Git: configurations

```
$ git config --global user.name "FirstName LastName"  
$ git config --global user.email "your-email@email-provider.com"  
$ git config --global color.ui true  
$ git config --list
```

Git: starting a repository

```
$ git init  
$ git status
```

OPÇÕES DE CONFIGURAÇÃO

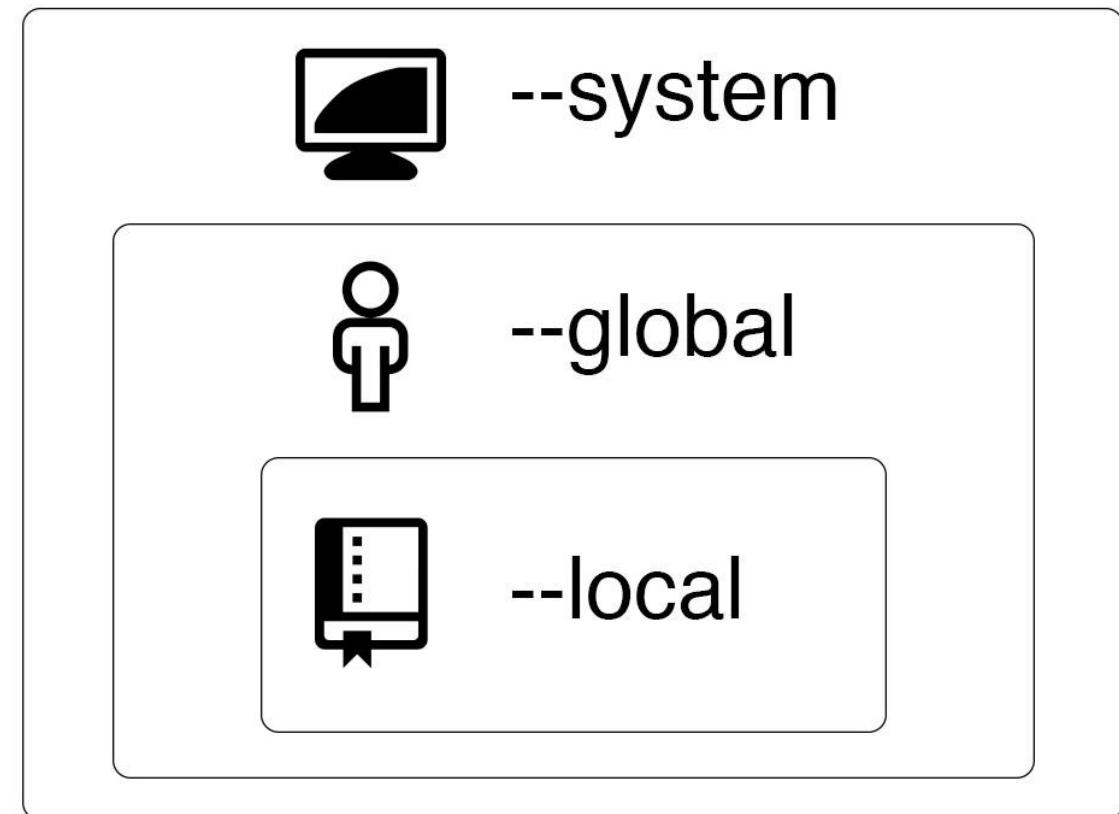
Há três tipos de opção para a configuração de variáveis:

- sistema (--system),
- global (--global) e
- configuração local.

Por **padrão** a configuração é local. Se o objetivo é realizar uma configuração por usuário de um sistema operacional utilize a opção **--global**. Se o objetivo é realizar uma configuração para todos os usuários de um sistema operacional utilize **--system**.

Para desfazer uma configuração:

```
$ git unset <opção> <variável>
```



EXERCÍCIO

A primeira coisa a se fazer ao iniciar o git é definir nome do usuário e seu endereço de e-mail. Isso é importante porque a cada commit, o git armazena esta informação! Vamos agora configurar local e globalmente o usuário, seu e-mail e cores:

1. \$ git config user.name fulano da silva
2. \$ git config user.email fulano.silva@exemplo.com
3. \$ git config --global user.name sicrano da silva
4. \$ git config user.email sicrano.silva@exemplo.com
5. \$ git config --global color.ui true



EXERCÍCIO

O comando `git config --list` permite verificar todas as configurações do seu git. Verifique agora as suas configurações!

```
$ git config --list
```



EXERCÍCIO

A opção `--unset` permite desfazer uma configurações do seu git.
Desfaça agora as suas configurações de nome, liste e refaça.

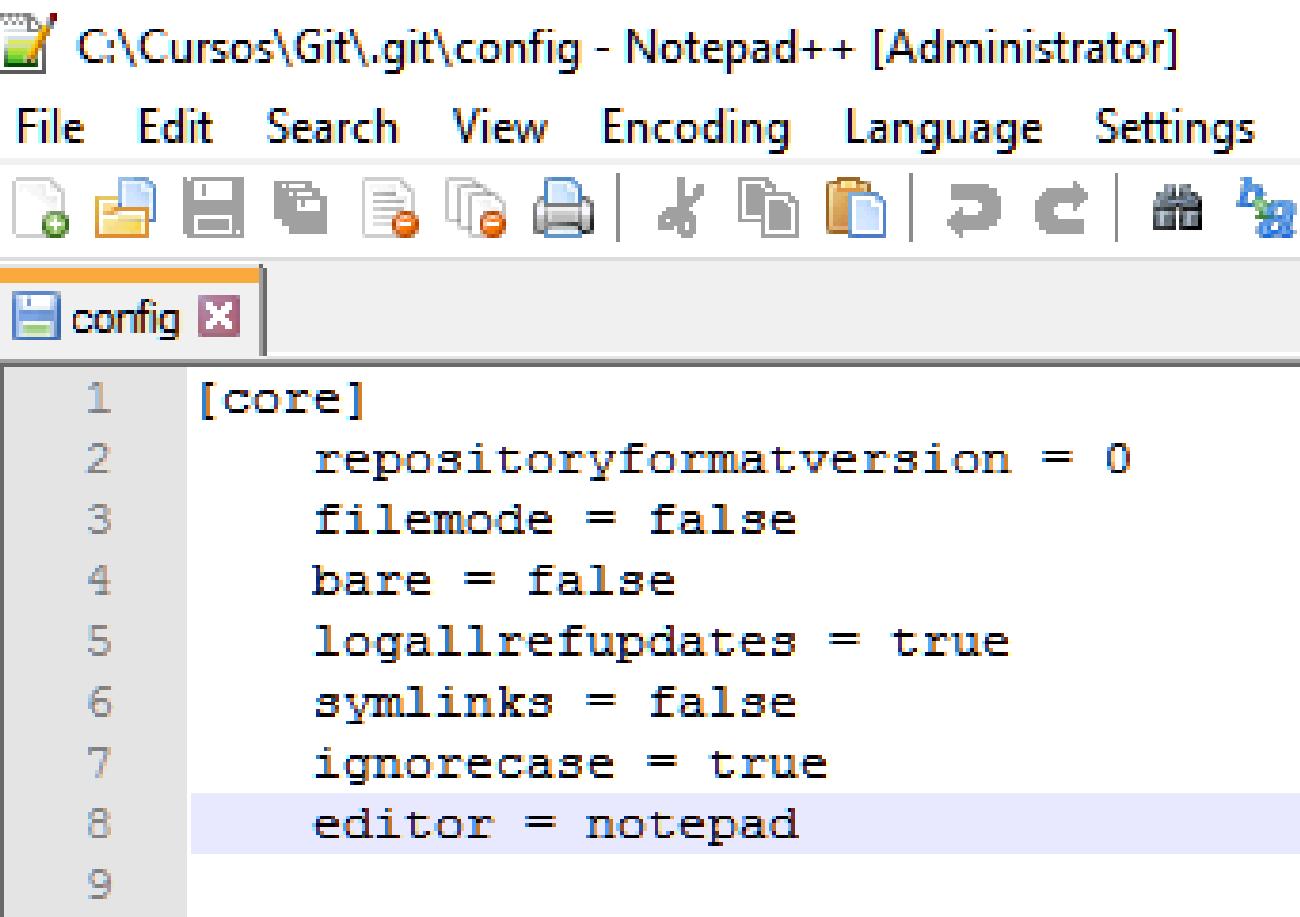
1. `$ git config --unset user.name`
2. `$ git config --unset --global user.name`
3. `$ git config --list user.name`



ARQUIVO .GIT/CONFIG

O arquivo config apresenta opções de configurações locais realizadas, além das configurações padrão.

-  hooks
-  info
-  objects
-  refs
-  config
-  description
-  HEAD



The screenshot shows the Notepad++ application window displaying the contents of the `C:\Cursos\Git\.git\config` file. The window title bar reads "C:\Cursos\Git\.git\config - Notepad++ [Administrator]". The menu bar includes File, Edit, Search, View, Encoding, Language, and Settings. The toolbar contains various icons for file operations like Open, Save, Print, and Cut/Paste. The main editor area has a tab labeled "config" and shows the following configuration options:

```
1 [core]
2     repositoryformatversion = 0
3     filemode = false
4     bare = false
5     logallrefupdates = true
6     symlinks = false
7     ignorecase = true
8     editor = notepad
9
```

EXERCÍCIO

Inclua como usuário global o nome beltrano, verifique em git config --list e observe o arquivo .git/config no seu repositório local. Houve alteração? Porque? Como faço para modificar uma configuração diretamente no arquivo?

Para saber onde suas configurações estão armazenadas:

```
$ git config --list --show-origin
```

Uma vez localizado o arquivo de origem da informação você pode alterá-lo.



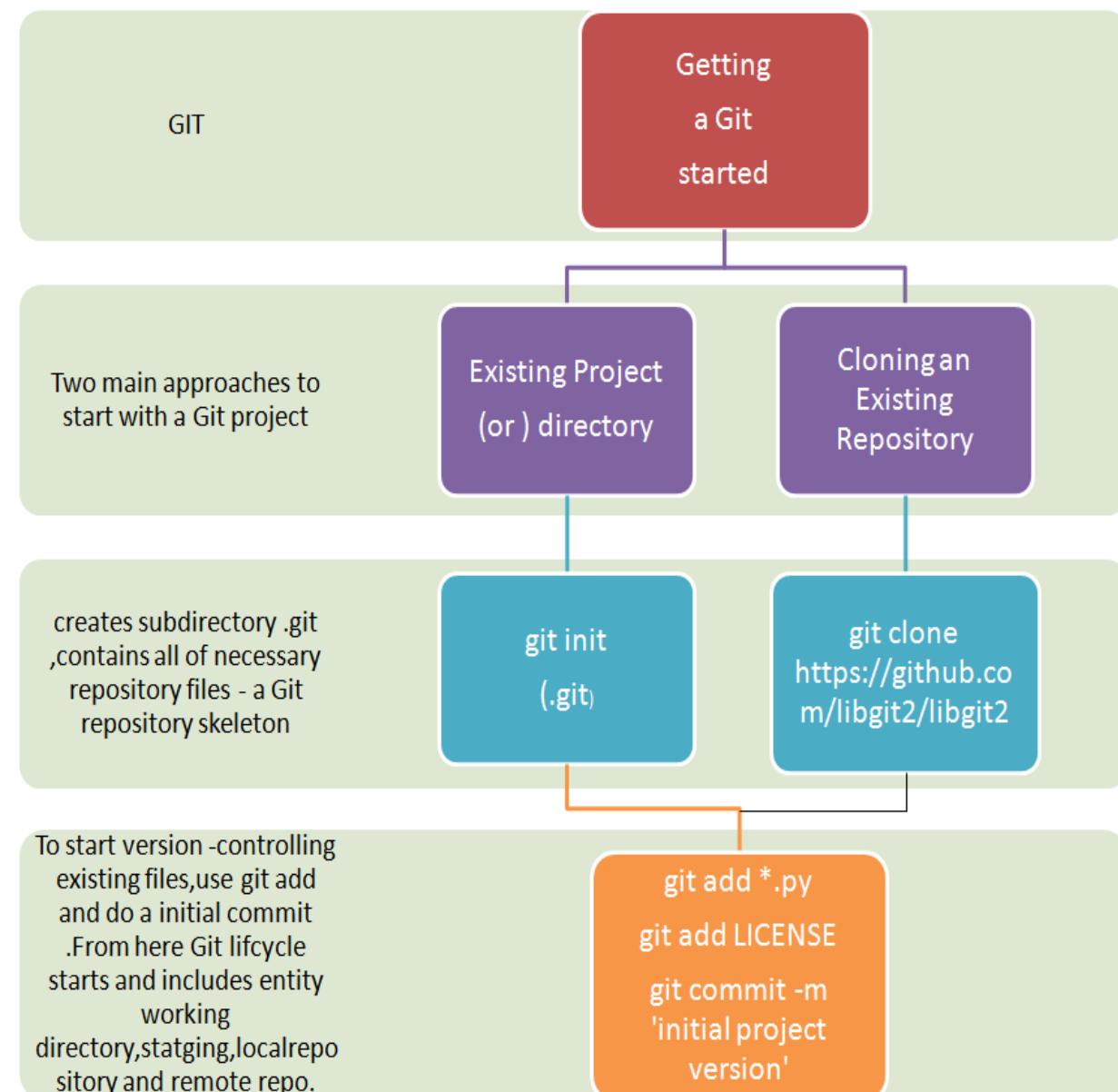
GIT ESSENTIALS

INICIALIZAÇÃO COM GIT

Você pode obter um projeto controlado pelo git de duas maneiras:

A primeira de um diretório ou de um projeto existente e então o git é inicializado.

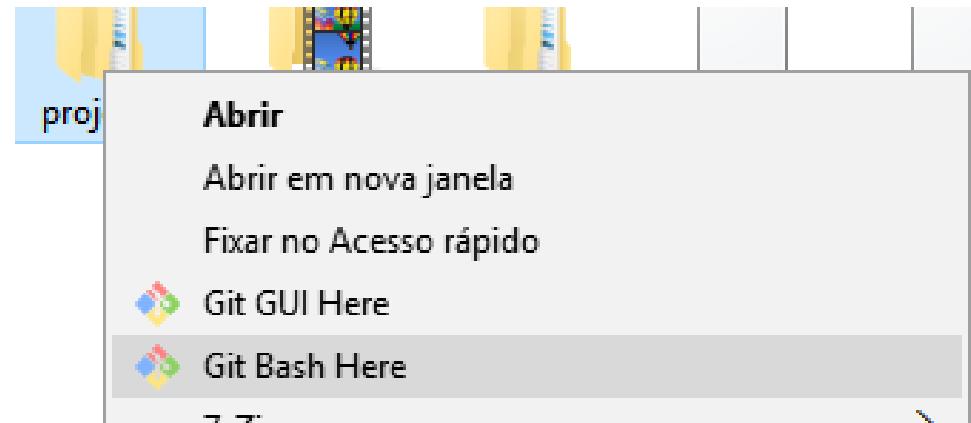
A segunda quando você clona um repositório git existente a partir de outro servidor.



GIT INIT

Caso você esteja queira iniciar o monitoramento de um projeto novo ou pré-existente com o git:

\$ git init



Quando houve a criação do repositório usado o recurso integrado shell, este é o comando que foi utilizado.

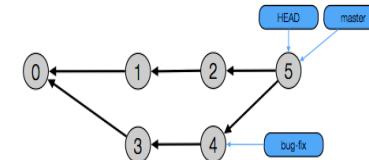
GIT CLONE

Caso você queira copiar um repositório git já existente:

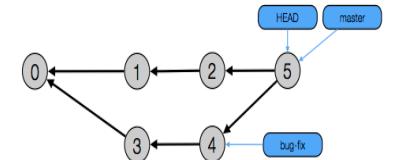
```
$ git clone <url>
```

Caso você queira clonar o repositório em um diretório diferente da definição original do projeto remoto, é possível especificar um outro diretório utilizando a opção abaixo:

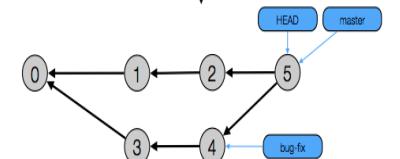
```
$ git <url> <diretório>
```



**Use “git clone”
to copy a repo
to your machine**



git clone



EXERCÍCIO

clone o projeto remoto do endereço:

`git://github.com/manupe61/curso-git`

`$ git clone git://github.com/manupe61/curso-git`



EXERCÍCIO

Apague o projeto “curso-git” do seu diretório e clone novamente o projeto remoto em um outro diretório de sua livre escolha.

```
$ git clone git://github.com/manupe61/curso-git <diretório>
```



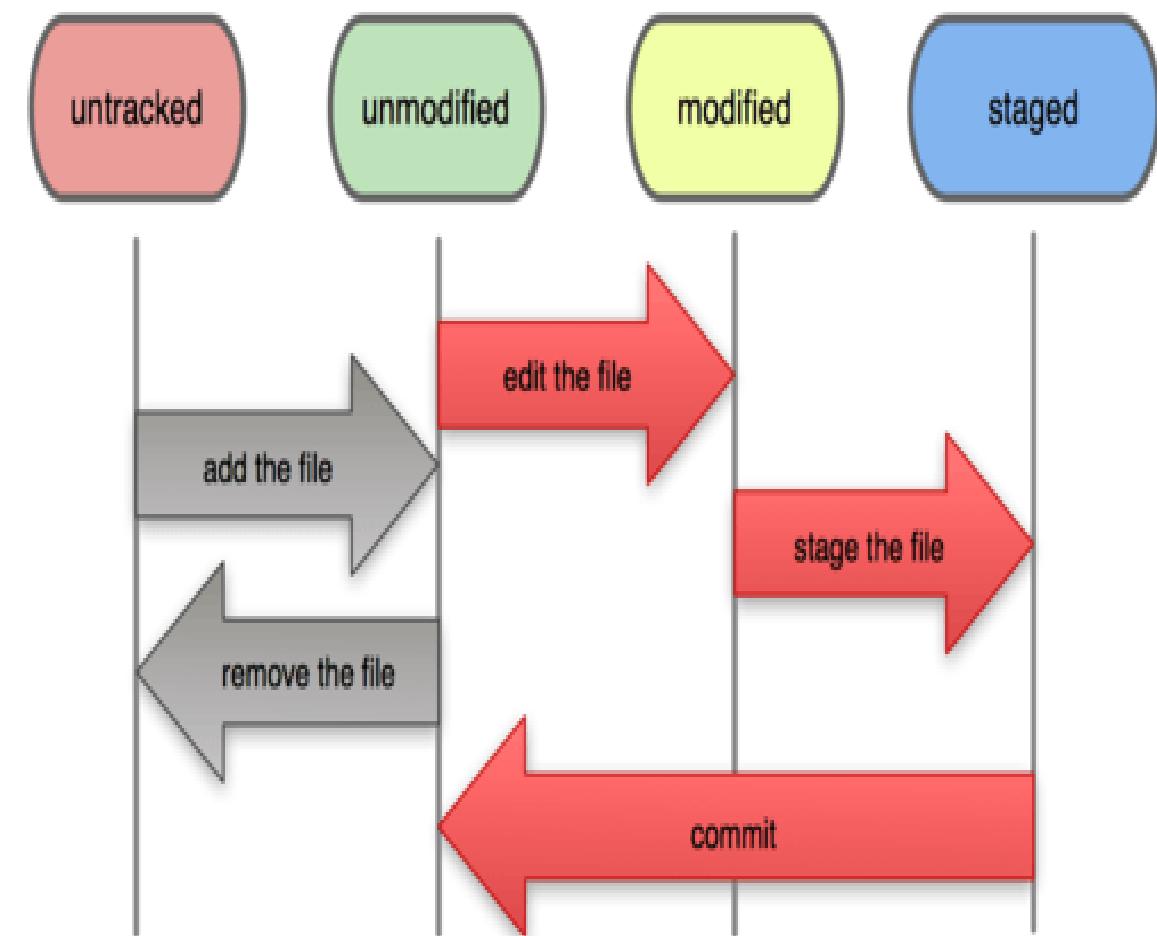
GRAVANDO ALTERAÇÕES NO REPOSITÓRIO

Quando um repositório é inicialmente clonado, todos os seus arquivos estarão monitorados e inalterados (unmodified) porque você simplesmente os obteve e ainda não os editou.

Conforme você edita esses arquivos, o git passa a vê-los como modificados, porque você os alterou desde que estes sofreram seu último commit.

A principal ferramenta utilizada para determinar quais arquivos estão em quais estados é o comando **git status**.

File Status Lifecycle



EXERCÍCIO

Vamos verificar o status dos arquivos no repositório clonado?

Selecione o novo repositório (a pasta que você escolheu) e digite o comando:

```
$ cd <pasta do projeto clonado>  
$ git status
```



ANÁLISE DO EXERCÍCIO

Isso significa que o diretório clonado ainda é um repositório limpo (tree clean), ou seja, arquivos **unmodified**.

O comando mostra em qual **branch** você se encontra. Neste caso é a branch **master**, que é a branch padrão.

Há informação sobre o branch de origem, neste caso ‘origin/master’

O git adota como “origin”, o nome do repositório remoto, que é o nome de repositório padrão.

```
Administrador@DESKTOP-037U9EO MINGW64 /c/Cursos/Git/mygrit (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

EXERCÍCIO

1. Altere o arquivo readme.md incluindo as linhas:

```
### autor
```

```
João da Silva
```

2. verifique novamente o status

```
$ git status
```



ANÁLISE DO EXERCÍCIO

Você pode ver que o seu novo arquivo README está agora como “modified”, ou seja, o git informa que há uma nova versão do arquivo.

E agora? O que se deve fazer para que a nova versão do arquivo possa fazer parte do meu projeto?

```
Administrador@DESKTOP-037U9E0 MINGW64 /c/Cursos/Git/curso (master)
```

```
$ git status
```

```
On branch master
```

```
Your branch is up to date with 'origin/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: README.md
```

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

GIT ADD

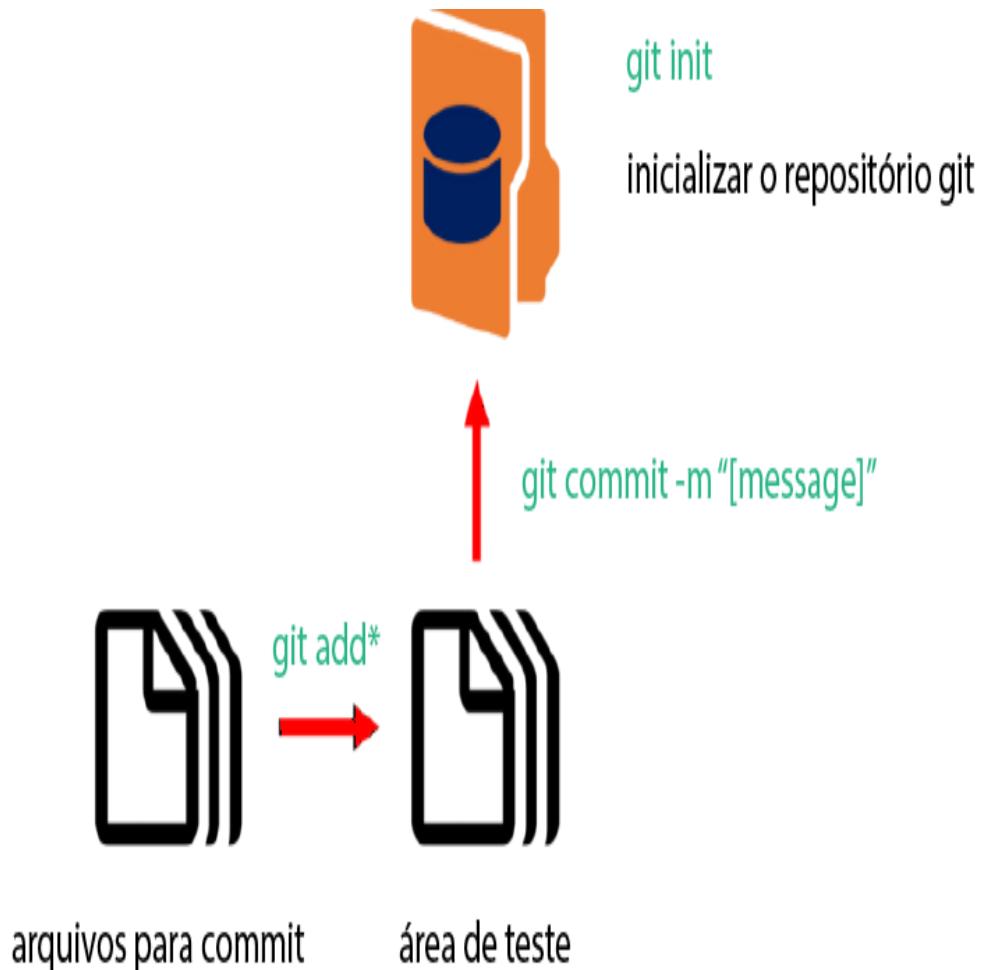
Para que esta versão do arquivo `readme.md` faça parte da nova versão do projeto você deve primeiro finalizar o seu trabalho até que o arquivo esteja pronto e depois colocá-lo na “área de seleção”

`$ git add <arquivo>`

É possível adicionar todos os arquivos de um diretório com o comando:

`$ git add .`

Você também pode mover os arquivos para uma pasta específica e realizar o mesmo comando.



EXERCÍCIO

Faça com que o arquivo `readme.md` se torne um arquivo monitorado e verifique novamente o status.

1. `$ git add readme.md`
2. `$ git status`



ANÁLISE DO EXERCÍCIO

Pode-se dizer que o arquivo `readme.md` está selecionado (staged) pois está sob o rótulo “Changes to be committed”.

Internamente, o git executa o checksum de todos os arquivos e gera um hash (cabeçalho) (integridade).

Para retirar um arquivo de selecionado, basta “resetar” ou limpar o cabeçalho e ele volta a ser referenciado apenas pelo cabeçalho anterior

\$ `git reset HEAD <arquivo>`

\$ `git reset HEAD .`

```
Administrador@DESKTOP-037U9EO MINGW64 /c/Cursos/Git/curso (master)
$ git add README.md
```

```
Administrador@DESKTOP-037U9EO MINGW64 /c/Cursos/Git/curso (master)
$ git status
On branch master
```

Your branch is up to date with 'origin/master'.

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

modified: README.md

EXERCÍCIO

Você cometeu um erro ao informar no arquivo REAME que o autor era o Fulano da Silva. Na verdade era o Beltrano da Silva. Faça com que o arquivo readme.md volte ao estado anterior e verifique novamente o status

1. \$ git reset HEAD readme.md
2. \$ git status



RECUPERANDO VERSÃO DE ARQUIVOS

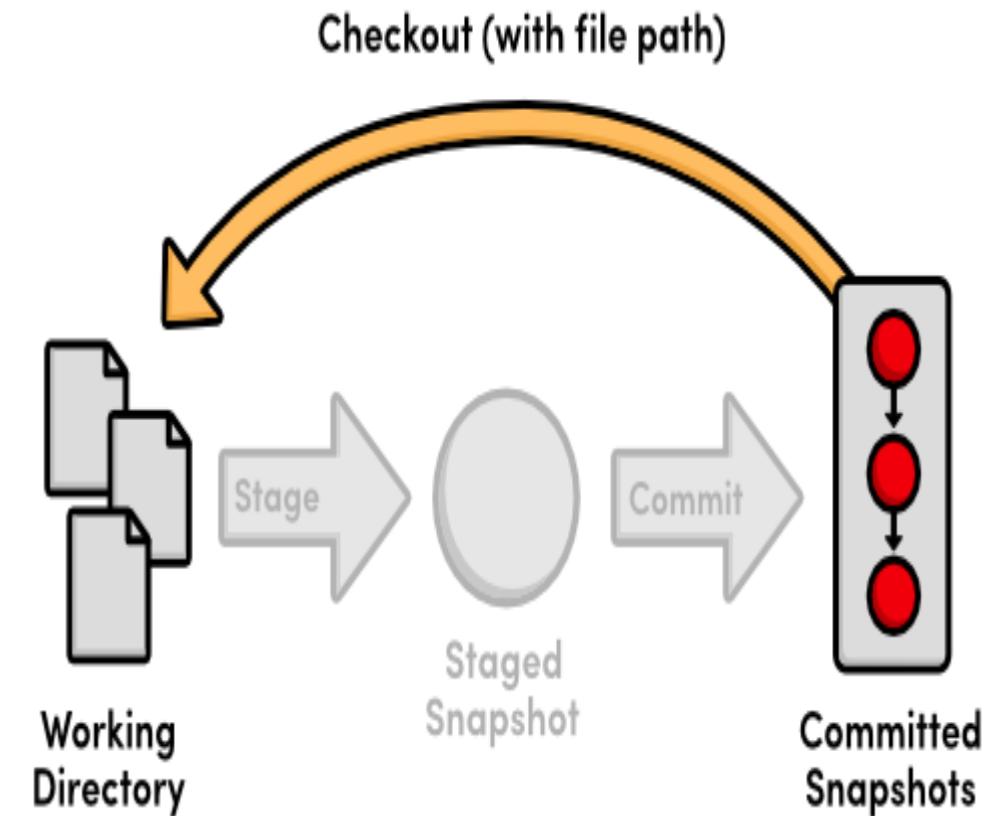
E se você quiser descartar as modificações realizadas no arquivo e quer voltá-lo para a versão original desde o último commit?

`$ git checkout -- <arquivo>`

Atenção: o comando `git checkout` não permite modificações se o arquivo já estiver selecionado. Você precisa resetar o cabeçalho de seleção primeiro!

`$ git reset head <arquivo>`

Reset
soft!



Git checkout movimenta o ponteiro HEAD!

EXERCÍCIO

Após retirá-lo da área de seleção, você também gostaria que as mudanças no arquivo fossem descartadas. Faça com que o arquivo volte a versão anterior e verifique o status.

1. \$ git checkout -- <arquivo>
2. \$ git status



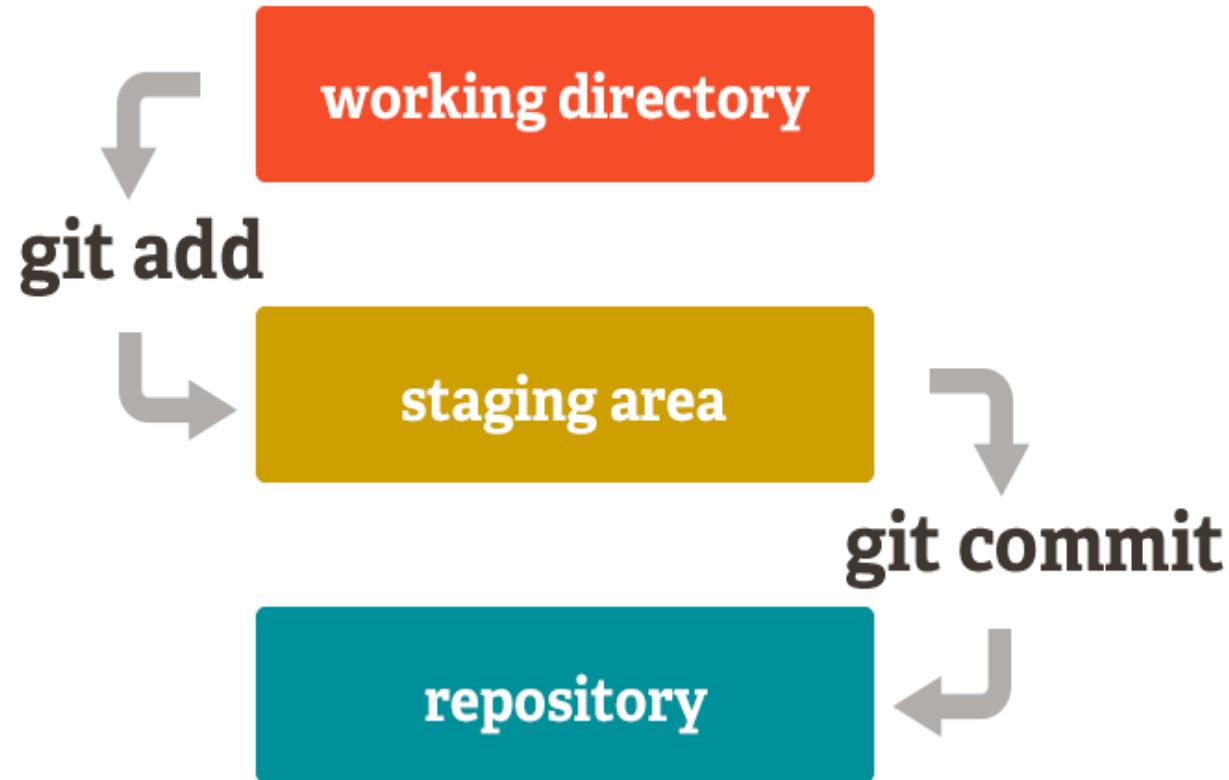
GIT COMMIT

Quando a área de seleção está do jeito que você quer, você pode fazer o commit de suas mudanças.

Lembre-se que tudo aquilo que ainda não foi selecionado não fará parte deste commit.

```
$ git commit -m "qualquer mensagem"
```

Obs.: edição de mensagens podem ser feito através de um editor de sua escolha. Este editor pode ser configurado com o comando `git config --global core.editor <editor>`. O editor padrão do git bash é o “vi”



EXERCÍCIO

1. Com o erro de autor, ajuste o arquivo `readme.md` incluindo as linhas:
autor
Beltrano da Silva
2. Adicione o arquivo na área de seleção e realize o commit. Verifique o status.
 1. `$ git add .`
 2. `$ git commit -m "incluso autor"`
 3. `$ git status`



ANÁLISE DO EXERCÍCIO

Agora você acabou de criar o seu primeiro commit! Você pode ver que o commit te mostrou uma saída sobre ele mesmo:

- qual o branch que recebeu o commit (master),
- qual o checksum SHA-1 que o commit teve (330795a),
- quantos arquivos foram alterados, e
- estatísticas gerais.

Lembre-se que o commit grava a captura da área de seleção, ou seja, está gravando a captura do seu projeto o qual poderá reverter ou comparar posteriormente.

Arquivos que receberam commit estão no repositório do .git e até que ocorram novas inclusões ou alterações, os arquivos ficarão como **unmodified** e não aparecem mais no **git status** (tree clean).

```
Administrador@DESKTOP-037U9EO MINGW64 /c/cursos/Git/curso (master)
$ git add .
```

```
Administrador@DESKTOP-037U9EO MINGW64 /c/cursos/Git/curso (master)
$ git commit -m "incluindo autor"
[master 330795a] incluindo autor
 1 file changed, 4 insertions(+)
```

```
Administrador@DESKTOP-037U9EO MINGW64 /c/cursos/Git/curso (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
```

```
nothing to commit, working tree clean
```

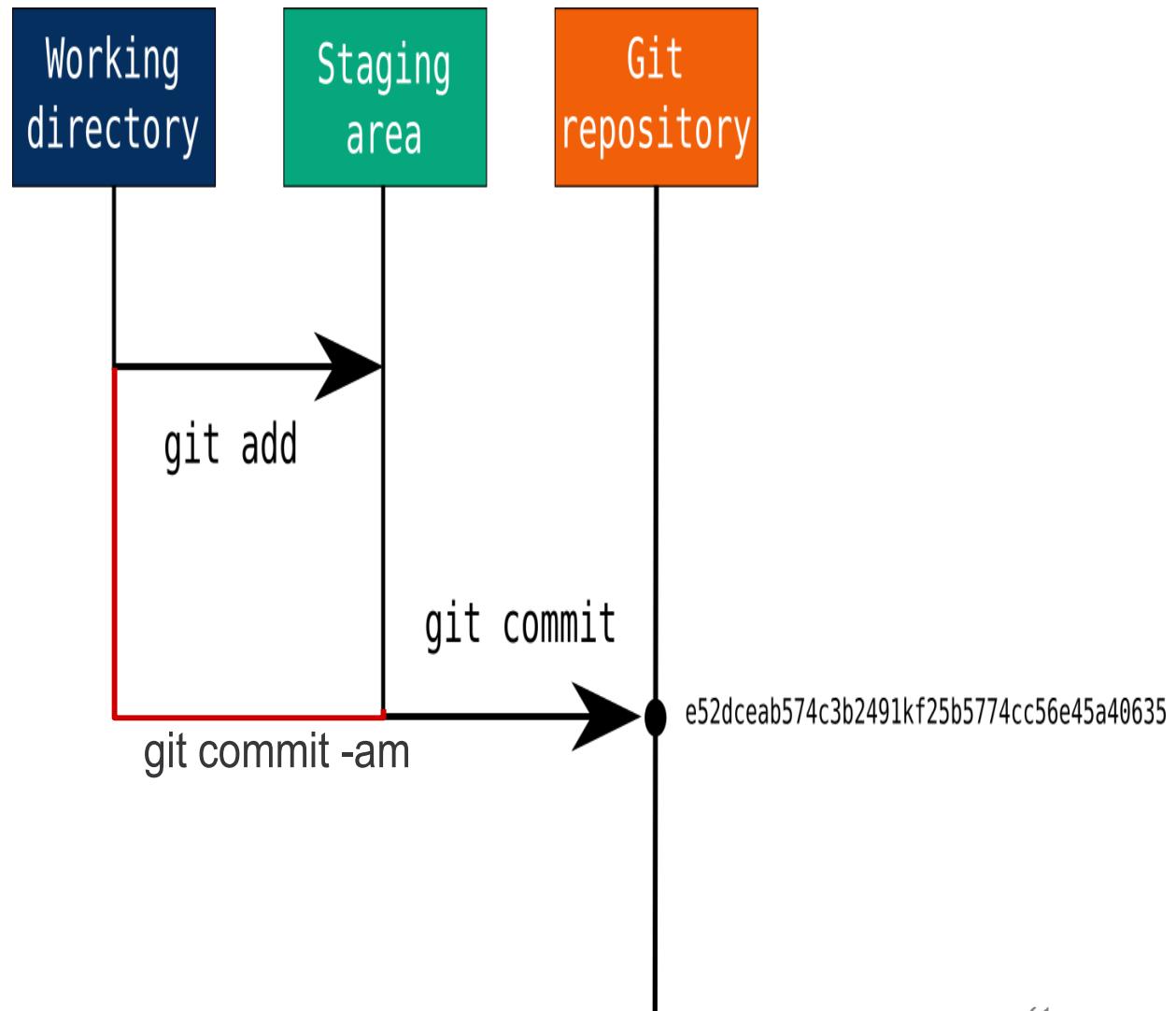
PULANDO ÁREA DE SELEÇÃO

Embora possa ser extraordinariamente útil para a elaboração de commits exatamente como você deseja, a área de seleção às vezes é um passo a mais no seu fluxo de trabalho.

Se você quiser pular a área de seleção, o git provê um atalho simples. Informar a opção `-a` ao comando `git commit` faz com que o git selecione automaticamente cada arquivo que está sendo monitorado antes de realizar o commit, permitindo que você pule a parte do `git add`:

```
$ git commit -am "pulando a área de seleção"
```

Esta opção só funciona em arquivos modificados



GIT CHECKOUT

O comando de checkout não é usado apenas para desfazer alterações, ele permite navegar por commits no repositório. **O git checkout movimenta o ponteiro HEAD.**

Por exemplo, podemos ver repositório no penúltimo commit:

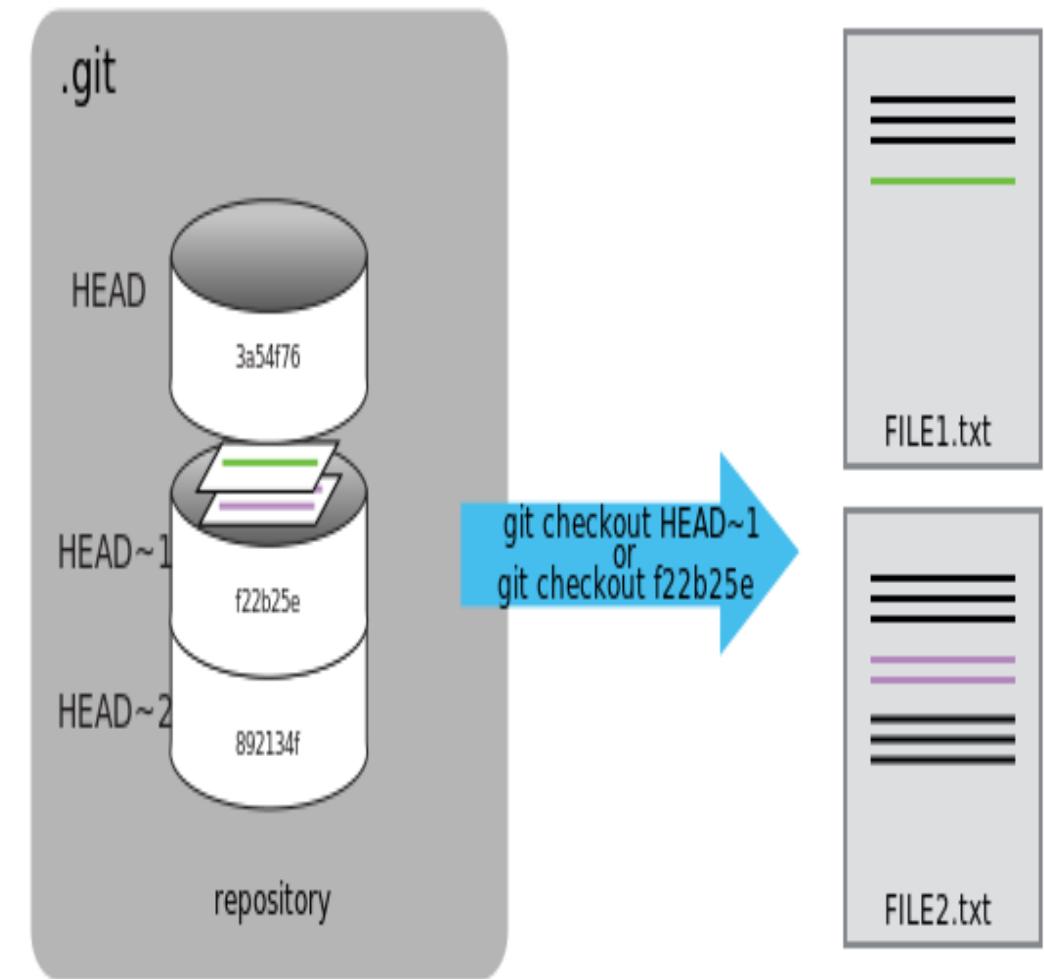
```
$ git checkout HEAD~1
```

Movimentando o ponteiro HEAD~1 significa se mover para um commit anterior ao atual.

Também podemos trocar HEAD por nome da branch e se mover entre branches.

Use git checkout master para voltar para a master no último commit.

```
$ git checkout <branch>
```



EXERCÍCIO

1. Altere novamente o arquivo readme.md incluindo as linhas:

```
### objetivo
```

Apresentar as diferentes regiões do mapa da cidade

2. Faça um commit direto, visualize o arquivo, volte para o commit anterior e visualize novamente o arquivo. Ao final, volte para o último commit.

1. \$ git commit -am “incluindo objetivo”
2. vi README.md (para sair :q)
3. \$ git checkout HEAD~1
4. vi README.md (para sair :q)
5. \$ git checkout master



GIT LOG

Depois da criação de vários commits, ou se clonou um repositório com um histórico de commits existente, você provavelmente vai querer ver o que aconteceu.

A ferramente mais básica e poderosa para fazer isso é o comando git log.

O git log lista os commits feitos em ordem cronológica reversa. Isto é, os commits mais recentes primeiro, com seu checksum SHA-1, o nome e e-mail do autor, a data e a mensagem do commit.

```
$ git log --format=medium
commit 5bf53efc38e1d780b32daf7c1d9acb54a7e93936
Author: Will Anderson <will@itsananderson.com>
Date: Mon Jul 21 08:27:50 2014 -0700

    Normalize indents to 4 spaces for all source files

commit bb82a7ab75f14cce5d8f8a60e97bfa8a2ceabbe
Author: Will Anderson <will@itsananderson.com>
Date: Mon Jul 21 01:11:38 2014 -0700

    Bump version to 0.2.4

commit d9bb2ff54751797208c0e551004cab1c1adb01cc
Author: Will Anderson <will@itsananderson.com>
Date: Mon Jul 21 01:11:18 2014 -0700

    Update .npmignore with new dev files

commit c75067e0f76c0e0adad58bb1ac663834f572f696
Author: Will Anderson <will@itsananderson.com>
Date: Sun Jul 20 23:00:44 2014 -0700

    Add full coverage for application mixin

commit 98ee9cbbb4454912d352b8bdb6a3dcbcd64a38f9
Author: Will Anderson <will@itsananderson.com>
Date: Sun Jul 20 14:34:18 2014 -0700

    Add full coverage for routeParams utility
```

EXERCÍCIO

Visualize o log do seu projeto. Quantos commits existem?

4 commits:

2 localmente: incluindo objetivo e incluindo autor

2 remoto: arquivo readme atualizado e o commit inicial



GIT LOG: OPÇÕES

git log --oneline: Este comando lista um commit por linha, mostra os primeiros 7 caracteres do commit SHA e a mensagem de commit.

git log --stat: Este comando mostra o (s) arquivo (s) que foram modificados, o número de linhas que foram adicionadas ou removidas e também exibe uma linha de resumo do número total de arquivos alterados e as linhas que foram adicionadas ou removidas.

git log -p: esse comando (patch) exibe os arquivos que foram modificados, a localização das linhas que foram adicionadas ou removidas e as alterações reais que foram feitas.

git show: exibe todas as informações padrão quando um commit não é especificado.

git show <hash>: exibe informações sobre apenas esse commit.

git shortlog: permite agrupar os commits de seus autores.

git log --<autor> = filtra por autor

git log --pretty: personaliza a saída do log no terminal.

EXERCÍCIO

Treine as opções de `$ git log` apresentadas



TABELA DE FORMATAÇÃO --pretty

| Opção | Descrição de Saída |
|-------|-------------------------------------------------|
| %H | Hash do commit |
| %h | Hash do commit abreviado |
| %T | Árvore hash |
| %t | Árvore hash abreviada |
| %P | Hashes pais |
| %p | Hashes pais abreviados |
| %an | Nome do autor |
| %ae | Email do autor |
| %ad | Data do autor (formato respeita a opção -date=) |
| %ar | Data do autor, relativa |
| %cn | Nome do committer |
| %ce | Email do committer |
| %cd | Data do committer |
| %cr | Data do committer, relativa |
| %s | Assunto |

git log --pretty: format:<opção><opção>~<opção>

Qual a diferença entre autor e committer? O autor é a pessoa que originalmente escreveu o trabalho, enquanto o committer é a pessoa que realizou o último patch (commit).

EXERCÍCIO

Treine as opções de `$ git log --pretty` apresentadas



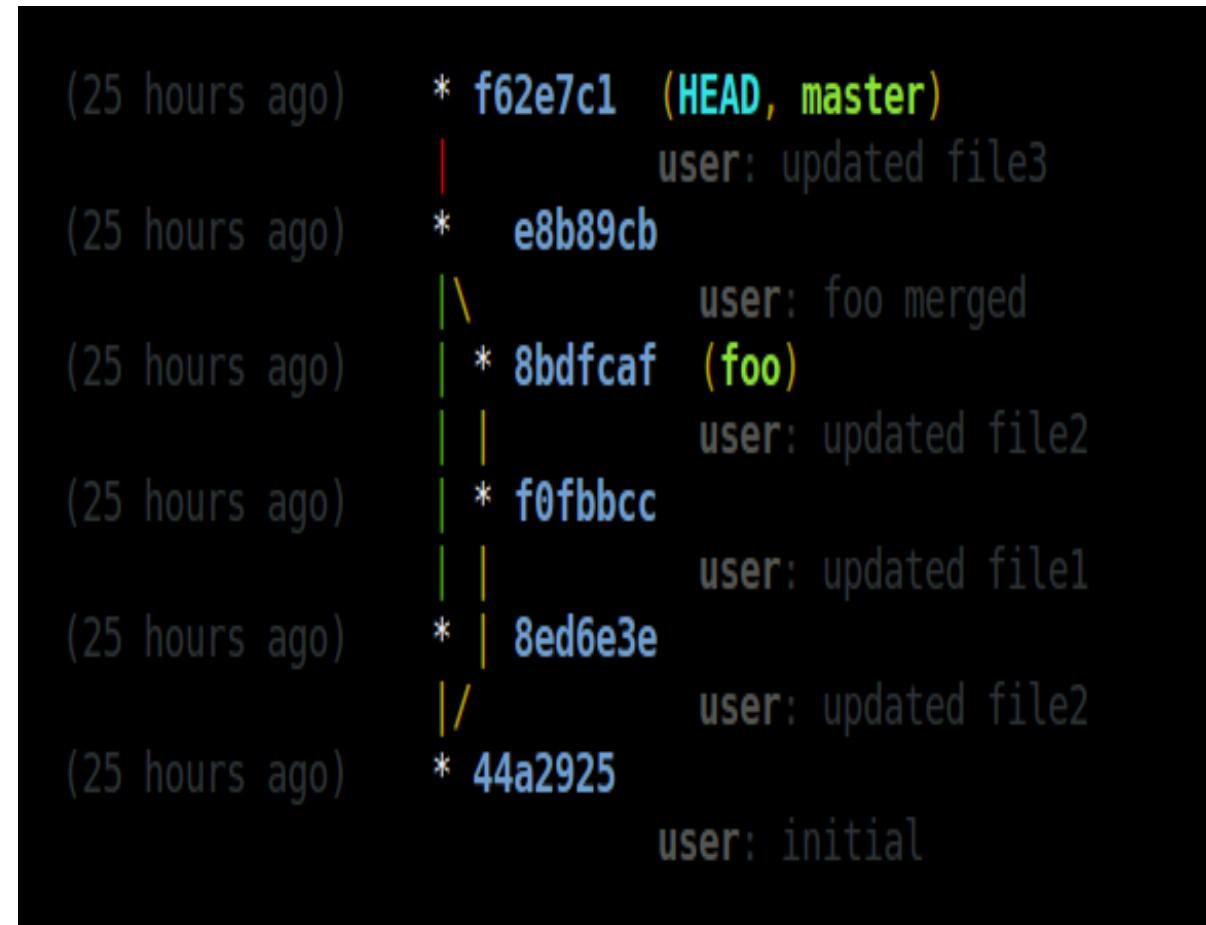
VISUALIZANDO HISTÓRICO GRAFICAMENTE

O comando `git log --graph` desenha um gráfico ASCII representando a estrutura de ramificação do histórico de confirmação.

Ele é comumente usado em conjunto com os comandos `--oneline` e/ou `--decorate` para facilitar a visualização de qual commit pertence a qual ramificação.

```
$ git log --graph
```

Você pode optar por uma interface GUI (**Graphical User Interface**) para obter um resultado mais interessante.

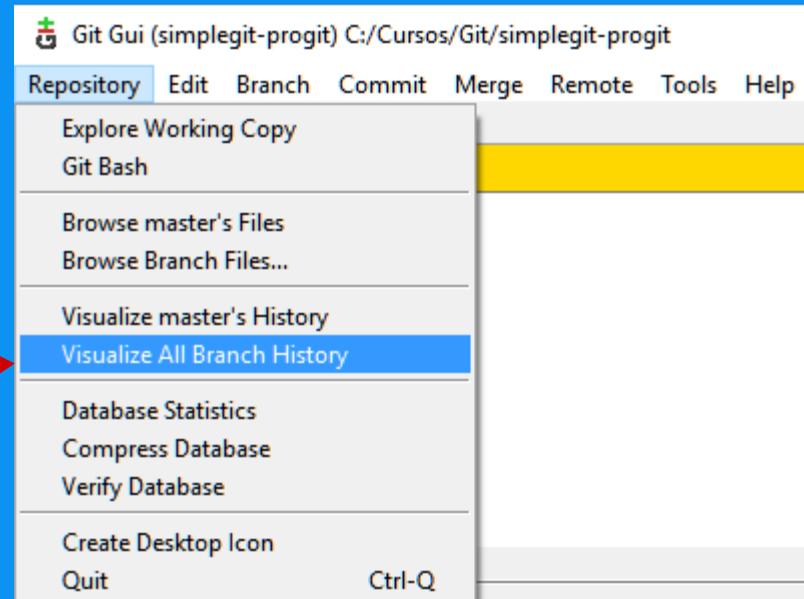


EXERCÍCIO

Utilize a opção --graph e a opção Git GUI Here do Windows para visualizar o histórico do projeto.

```
$ git log --graph  
$ git log --oneline --graph
```

Vamos conhecer o Git GUI?



curso: --all - gitk

File Edit View Help

master incluindo objetivo
incluir autor
remotes/origin/master Update README.md
Add files via upload
Initial commit

sicrano <jsilva@exemplo.com.br>
sicrano <jsilva@exemplo.com.br>
Manuel <30840022+Manupe61@users
Manuel <30840022+Manupe61@users
Manuel <30840022+Manupe61@users

2018-12-01 10:30:02
2018-12-01 09:45:37
2018-11-30 16:08:36
2018-11-30 16:00:50
2018-11-30 15:44:35

SHA1 ID: e689aed2898b46c11d2d82d6245dac9f27aea616

Find commit containing: Exact All fields

Search

Diff Old version New version Lines of context: 11 Ignore space changes

Patch Tree

Comments README.md

-- README.md --
index 132f0c6..016e015 100644
@@ -4,12 +4,16 @@ Mapa que interage com o usuário demarcando as regiões de
Inicializar
Rodar o arquivo de inicialização
Pre-requisitos
Não há;
Autor
-Beltrano da Silva
\ No newline at end of file
+Beltrano da Silva
+
+### Objetivo
+
+A definir

commits

autores

Diferenças por commit

Visualização por patch (incremento) ou por árvore

Pesquisar na Web e no Windows

11:17
PTB2 01/12/2018

AJUSTE NO COMMIT

Uma das situações mais comuns para desfazer algo, acontece quando você faz o commit muito cedo e possivelmente esqueceu de adicionar alguns arquivos, ou você errou sua mensagem de commit.

Se você quiser tentar fazer novamente esse commit, você pode executá-lo com a opção --amend:

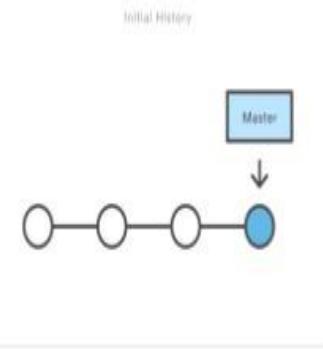
```
$ git commit --amend  
$ git commit --amend -m "nova mensagem"
```

Atenção: evite usar amend quando o commit já faz parte do histórico do repositório remoto!

Common Git Routine (Local)

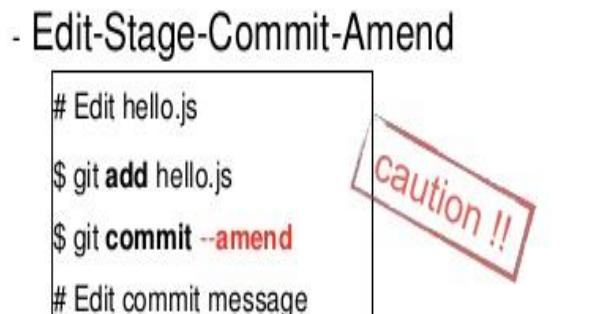
- Edit-Stage-Commit

```
$ vim hello.js  
# Edit  
$ git add hello.js  
$ git commit  
# Edit commit message
```



- Edit-Stage-Commit-Amend

```
# Edit hello.js  
$ git add hello.js  
$ git commit --amend  
# Edit commit message
```



**** Don't amend commit that you already push to public repository**

EXERCÍCIO

No seu último commit você declarou o objetivo e este não foi aceito pelo time que prefere “a definir”. Além disso você esqueceu de criar o arquivo “notas de release.txt” com a seguinte informação “v1.0 – Criação do Mapa Iterativo. Faça os ajustes necessários.

1. Ajustar arquivo README.md
2. Criar e editar arquivo Notas de Release.txt
3. \$ git add .
4. \$ git commit –amend -m “incluso objetivo e notas de release”



RESETANDO COMMIT

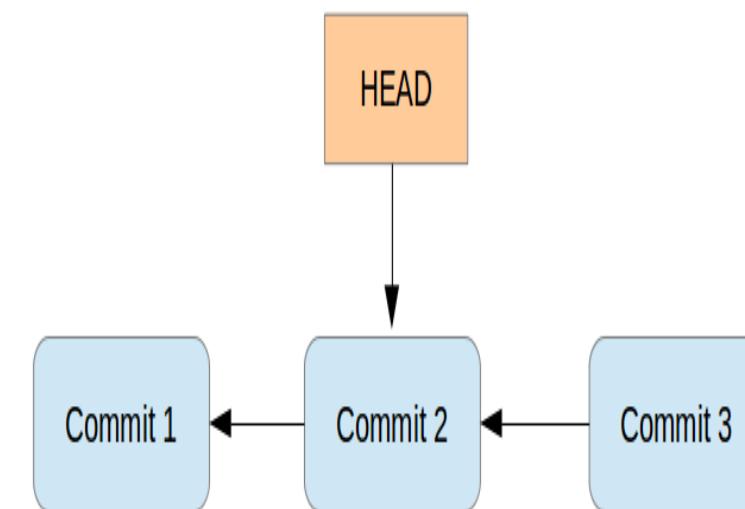
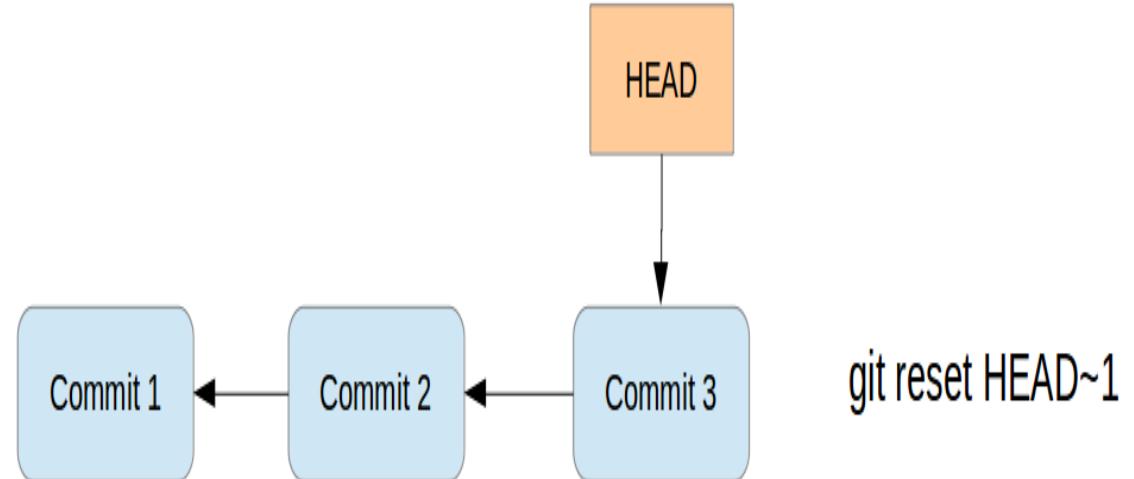
O comando git reset apaga o commit feito e repontera o HEAD para o commit desejado.

\$ **git reset HEAD~1** (neste caso reponera para um commit anterior).

Veja que as alterações nos arquivos permanecem (modo soft).

Neste caso, pode-se fazer um novo commit com o mesmo conteúdo do commit anterior, mas o commit será outro (com outro hash).

Obs.: o valor 1 pode ser alterado para “n”



EXERCÍCIO

Faça uma cópia do projeto. Use esta cópia agora. Verifique a diferença entre os dois últimos commits, apague o último commit realizado e verifique se houve alteração no arquivo README.md. Refaça o commit e verifique se é o mesmo commit ou um commit diferente.

1. Crie uma cópia da pasta do projeto e a utilize.
2. \$ git log
3. \$ git reset HEAD~1
4. \$ git log
5. \$ cat README.md
6. \$ git commit --am “refazendo tudo”



RESETANDO COMMIT E ALTERAÇÕES

Ao usar o git reset a opção padrão dele é soft. O reset soft não altera os arquivos, apenas o commit.

O outro modo de desfazer o commit é o modo hard.

Neste caso as alterações nos arquivos também serão desfeitas com o commit.

Para o reset hard use:

```
$ git reset --hard HEAD~1
```

Perceba que os arquivos estão exatamente como no commit anterior.



EXERCÍCIO

Verifique a diferença entre os dois últimos commits, apague o último commit realizado e verifique se houve alterações nos arquivos README.md e Notas de Release

1. \$ git log
2. \$ git reset --hard HEAD~1
3. \$ git log
4. \$ cat README.md



GIT REVERT

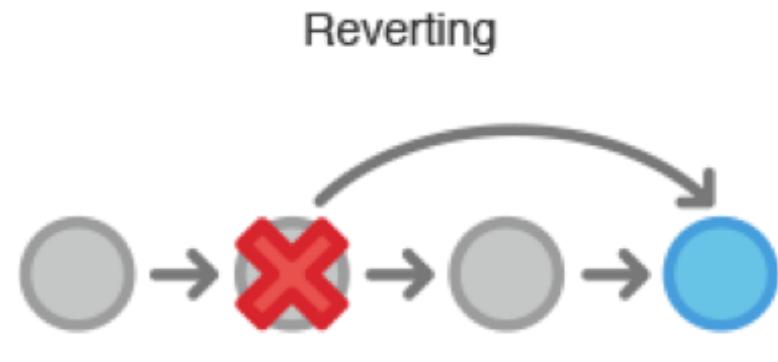
O comando `git reset --hard HEAD~1` não aceita caminho, neste caso a alternativa é o `git revert`.

O `revert` cria um novo commit que faz o reverso do commit especificado.

`$ git revert <hash>`

Pode haver conflitos nessa operação, pois ao reverter um commit que criou uma linha, pode causar conflito com outro commit que editou aquela linha.

Neste caso o `git` dará um aviso e você terá que resolver os conflitos.



EXERCÍCIO

Delete a cópia da projeto, faça uma nova cópia do projeto e a utilize. Verifique se o status é clean tree. Se não for, realize os commits. Reverta até o penúltimo commit realizado e verifique o log tanto no bash como no git GUI.

1. Delete a cópia anterior, crie uma nova cópia e a utilize.
2. \$ git status (deve estar clean tree, senão adicione e commit)
3. \$ git log
4. \$ git revert <hash>
5. \$ git log
6. Use o git GUI



.GITIGNORE

Muitas vezes, você terá uma classe de arquivos que não quer que o git adicione ou mostre como arquivos não monitorados.

Normalmente estes arquivos são gerados automaticamente como arquivos de log ou produzidos pelo seu sistema de build.

Nestes casos, você pode criar um arquivo contendo uma lista de padrões a serem checados chamado **.gitignore**.

Eis um exemplo de arquivo **.gitignore**:

```
1 # Windows image file caches
2 Thumbs.db
3 ehthumbs.db
4
5 # Folder config file
6 Desktop.ini
7
8 # Recycle Bin used on file shares
9 $RECYCLE.BIN/
10
11 # Windows Installer files
12 *.cab
13 *.msi
14 *.msm
15 *.msp
16
17 # Windows shortcuts
18 *.lnk
19
20 # =====
21 # Operating System Files
22 # =====
23
24 # OSX
25 # =====
26
27 .DS_Store
28 .AppleDouble
29 .LSOverride
--
```

EXERCÍCIO

Faça com que o arquivo Notas de Release seja ignorado em um futuro commit do projeto.

1. \$ touch .gitignore (não é possível pelo Windows!)
2. Inclua Notas de Release.txt
3. Crie ou altere o arquivo Notas de Release.txt
4. git status



GLOB

As regras para os padrões que você pode pôr no arquivo `.gitignore` são as seguintes:

- Linhas em branco ou iniciando com `#` são ignoradas.
- Padrões glob* comuns funcionam.
- Você pode terminar os padrões com uma barra (/) para especificar diretórios.
- Você pode negar um padrão ao iniciá-lo com um ponto de exclamação (!).

*glob é um termo utilizado no contexto de programação de computadores para descrever uma forma de convergência de padrões (ver tabela)

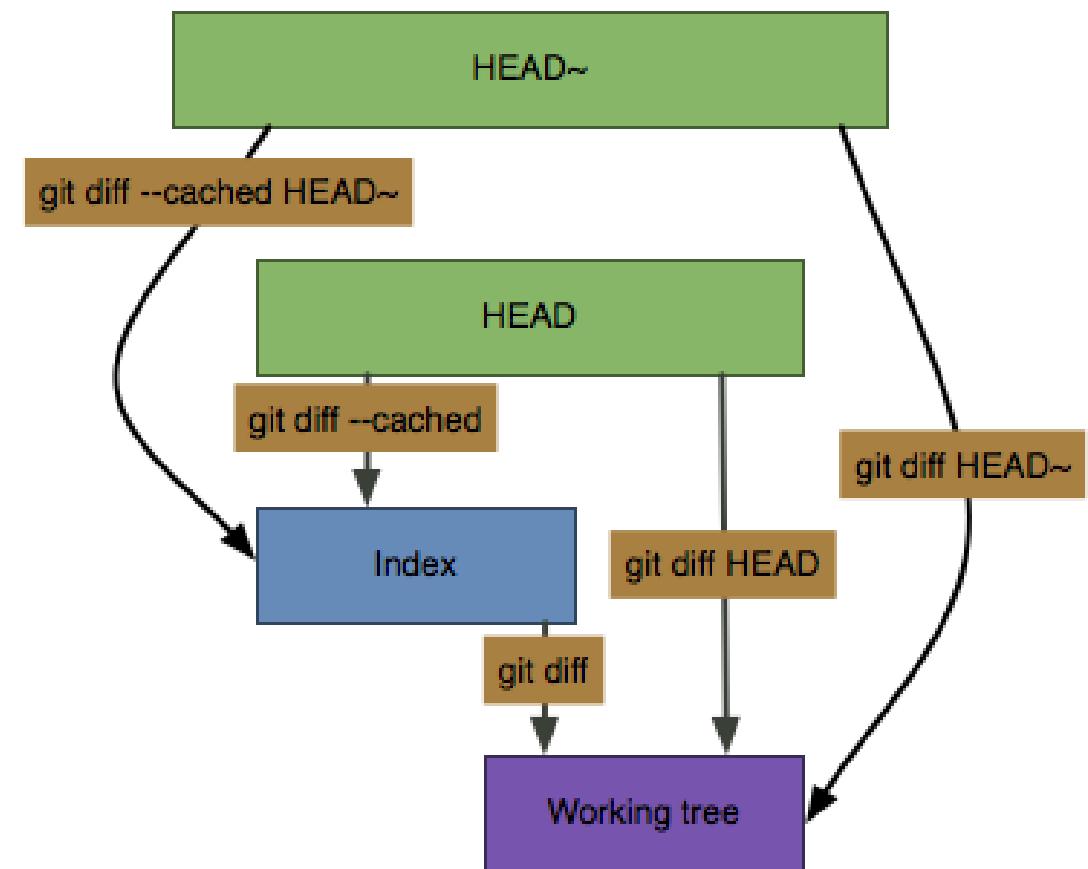
| Função | Exemplos |
|-------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| Casar com um caractere desconhecido | <code>?anto</code> casa com <code>santo</code> , <code>tanto</code> ou <code>manto</code> |
| Casar com qualquer número de caracteres desconhecidos | <code>Auto*</code> casa com <code>Automóvel</code> ou com <code>Autonomia</code> |
| Casar um caractere com um grupo de caracteres | <code>[ST]anto</code> casa com <code>Santo</code> ou <code>Tanto</code> mas não com <code>Manto</code> |
| Caractere de escape | <code>Auto*</code> só casará com <code>Auto*</code> |

GIT DIFF

Se o comando git status for muito vago, você pode saber exatamente o que foi modificado, não apenas quais arquivos foram alterados, usando o comando git diff.

O comando git status pode responder perguntas de maneira geral, mas o comando git diff mostra as linhas exatas que foram incluídas ou modificadas.

\$ git diff HEAD~1



EXERCÍCIO

Modifique o arquivo README.md e faça um novo commit. Verifique a diferença entre as suas versões.

1. Altere o arquivo README.md
2. \$ git commit --am “mensagem qualquer”
3. \$ git diff HEAD~1



REMOVENDO ARQUIVOS

Como remover um arquivo ou como não monitorar mais um arquivo (ou arquivos)?

O git rm é usado para remover ou não monitorar arquivos de um repositório Git.

Para remover:

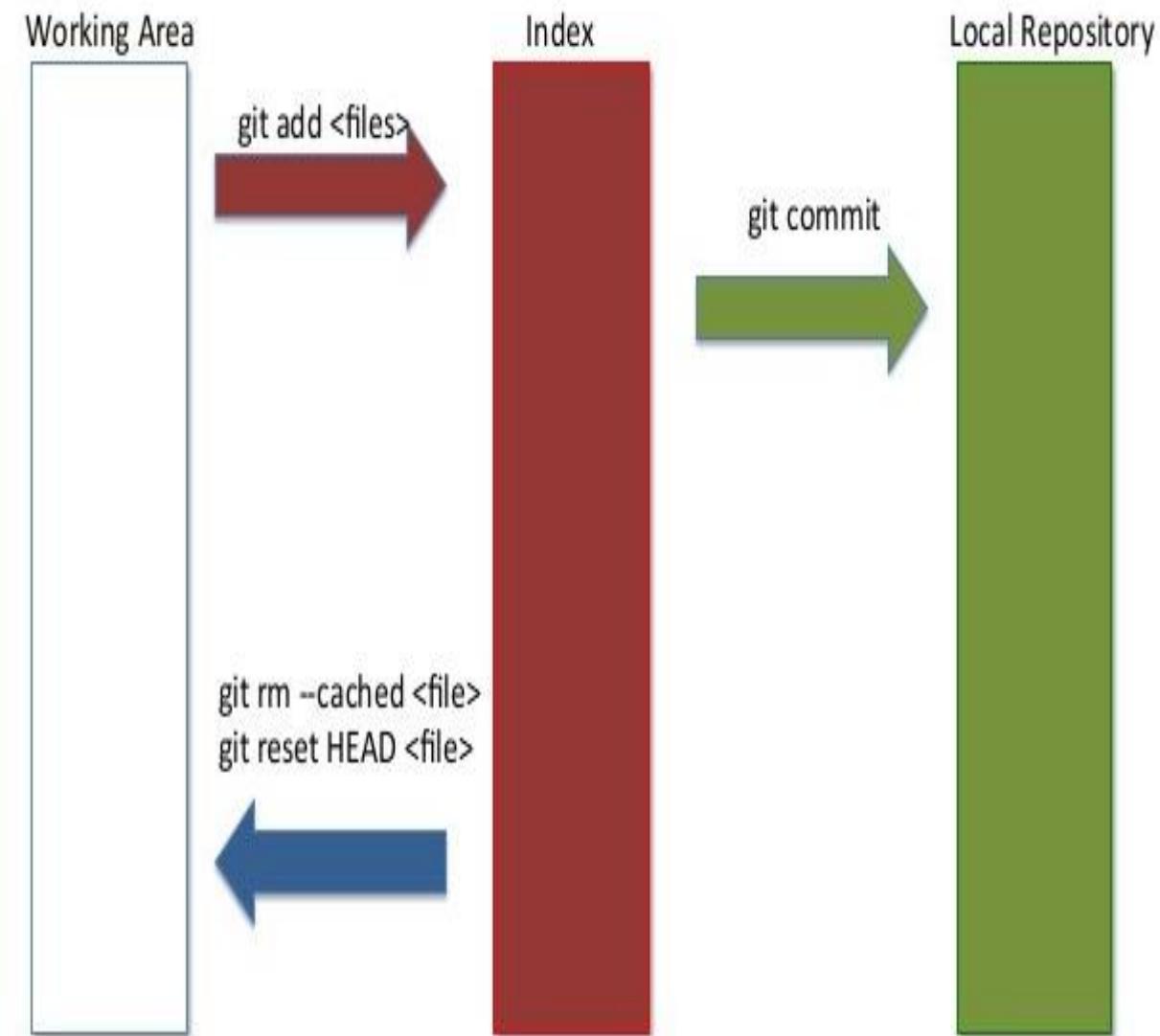
```
$ git rm <arquivo>
```

Para deixa-lo não monitorado:

```
$ git rm --cached <arquivo>
```

Para remover definitivamente:

```
$ git rm -f <arquivo>
```



EXERCÍCIO

Crie um arquivo qualquer com conteúdo, verifique o status, adicione-o, remova-o e verifique novamente o status. Use ambas as opções --cached e -f e diga qual a diferença entre elas.

1. \$ git add <arquivo>
2. \$ git rm --cached <arquivo>
3. \$ git status
4. \$ git add <arquivo>
5. \$ git rm -f <arquivo>
6. \$ git status



TAGS

A tag representa uma **etapa** do desenvolvimento.

\$ git tag

Para criar uma tag basta usar o comando git tag <tag> ou use a opção -a <tag> para uma tag “anotada”. Neste caso deve-se adicionar uma mensagem:

\$ git tag <tag>

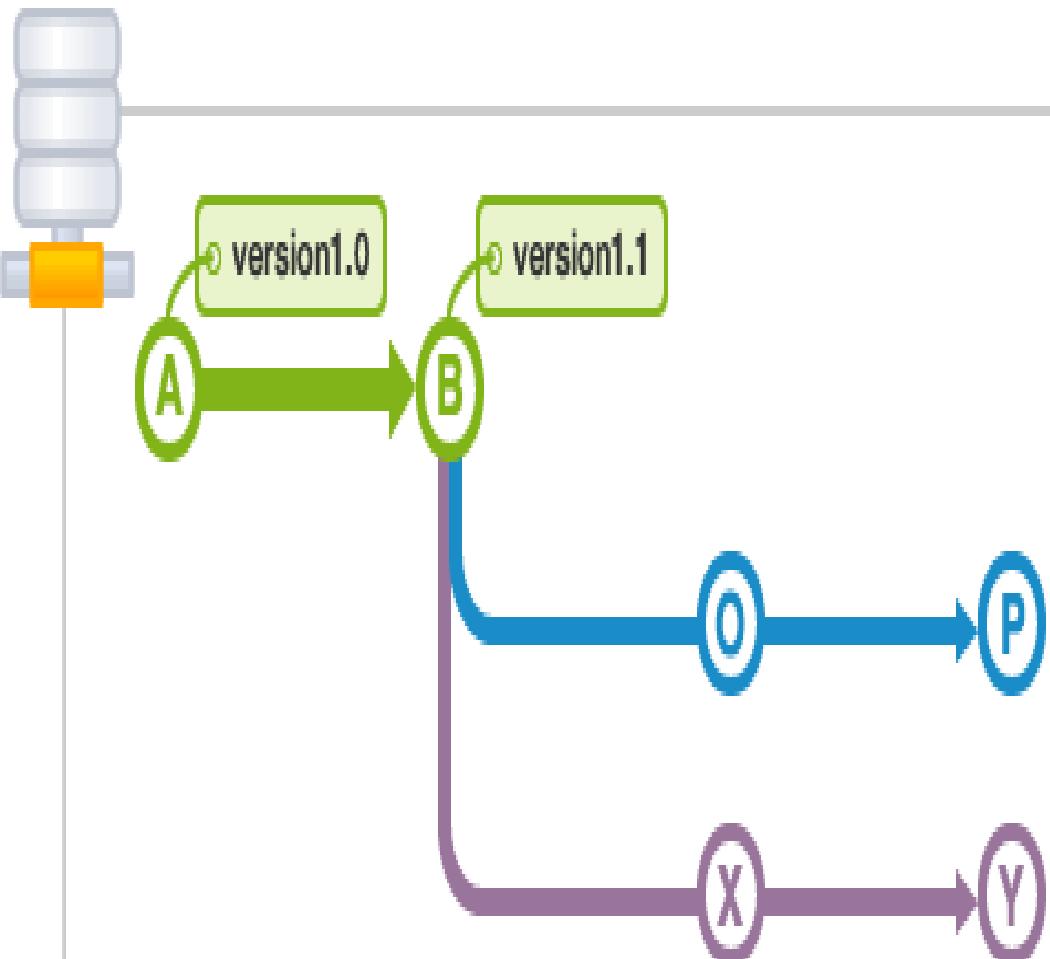
\$ git tag -a <tag> -m “minha mensagem”

Para verificar tags:

\$ git tag -v <tag>

Para deletar:

\$ git tag -d <tag>



EXERCÍCIO

Crie uma nova tag “anotada”. Liste as tags disponíveis e verifique a última tag. Delete a última tag. Recrie a tag anotada com a versão 2.0.

1. \$ git tag -a v1.1 -m “tag1.1”
2. \$ git tag
3. \$ git tag -v <tag>
4. \$ git tag -d <v1.1>
5. \$ git tag -a v2.0 -m “tag2.0”



EXERCÍCIO

Copie o seu projeto. Verifique as tags através do log do seu projeto. Navegue até a tag 1.0 usando o recurso checkout. Volte para a versão atual. Apague as versões 2.0. e 1.0

1. \$ cp -r <path> <path>
2. \$ git log
3. \$ git checkout <tag>
4. \$ git log
5. \$ git reset --hard HEAD~2 (não é possível pela tag)



Muito cuidado no uso desse comando em projetos remotos!



GIT WORKFLOW

BRANCHES

A criação de um branch não cria um novo objeto, mas um tipo de ponteiro, que evolui a cada commit. Você faz isso com o comando:

```
$ git branch <nome>
```

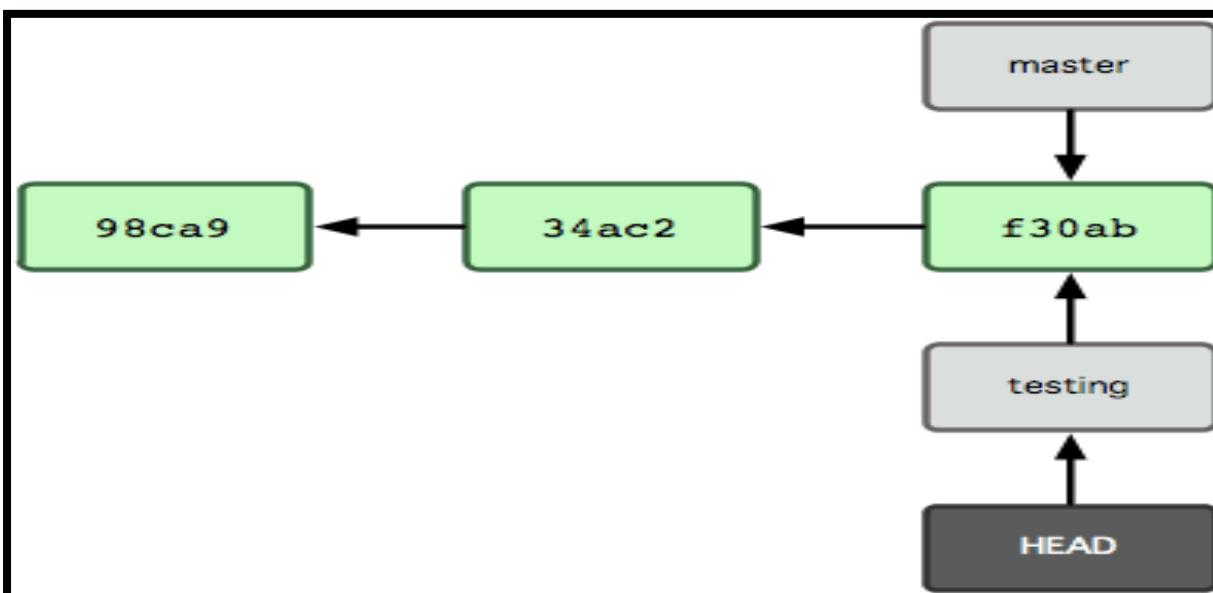
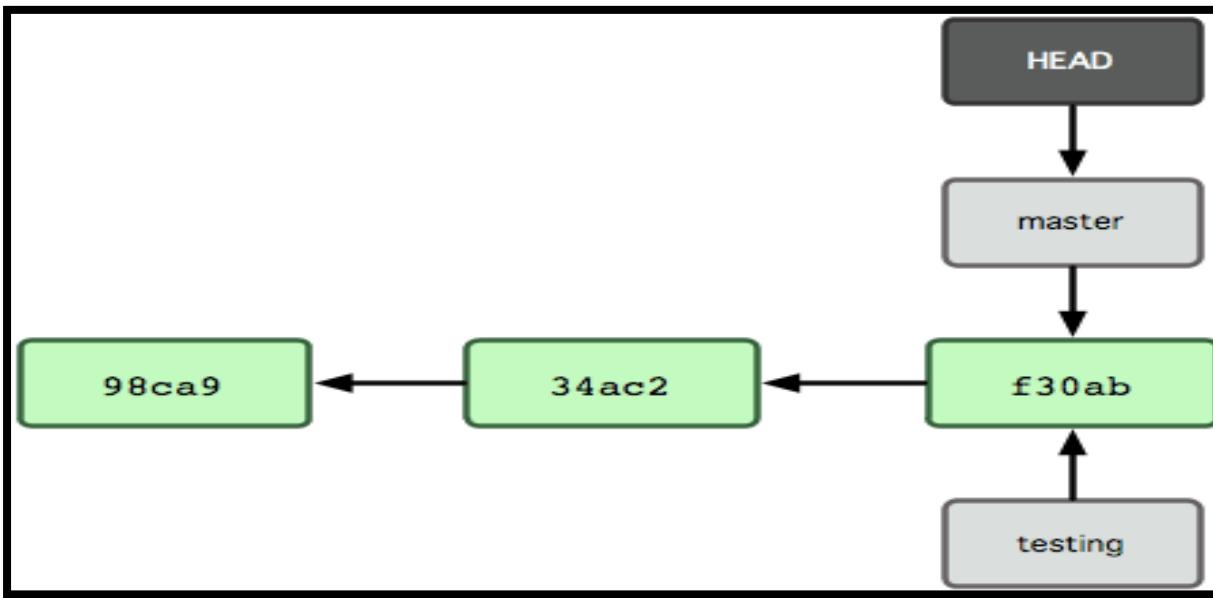
Como o git sabe o branch em que você está atualmente? O git mantém um ponteiro “deslizante” chamado HEAD.

Para mudar para um branch existente, você executa o comando git checkout que move o ponteiro HEAD entre as branches.

```
$ git checkout <nome da branch>
```

Para deletar um branch:

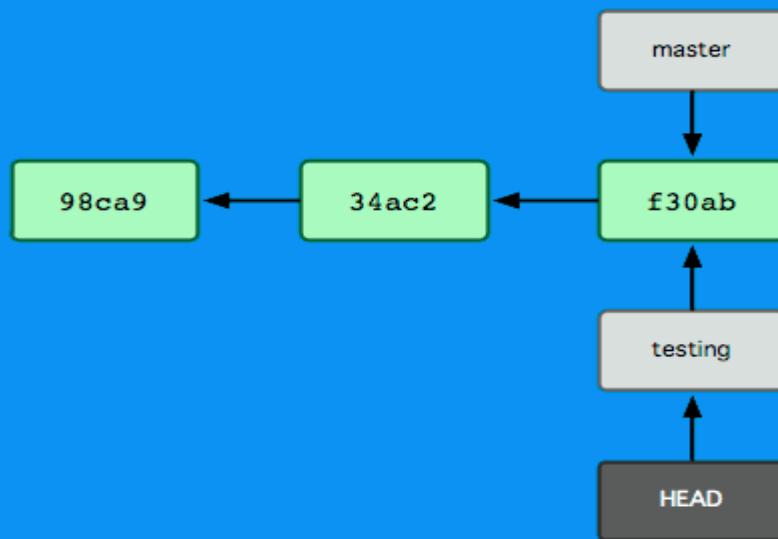
```
$ git branch -d <nome>
```



EXERCÍCIO

Crie uma cópia do projeto. Crie um branch para um desenvolvimento de teste chamada “teste” e verifique as branches existentes. Mude o ponteiro para a branch teste e verifique os branches novamente, o status e o log (com a opção --oneline). Verifique o ponteiro HEAD.

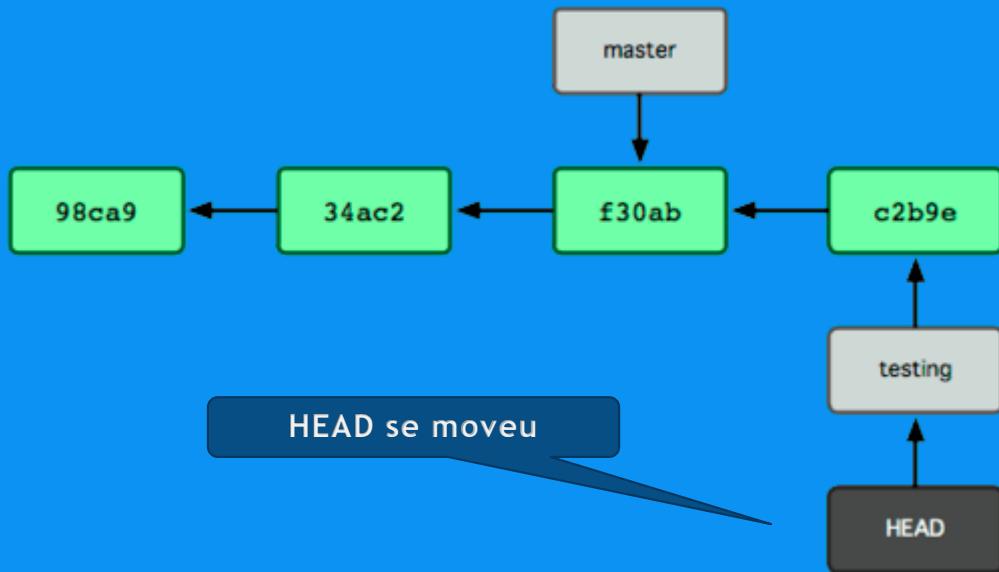
1. \$ git branch teste
2. \$ git branch
3. \$ git checkout teste
4. \$ git branch (verifique o asterisco)
5. \$ git status (on branch teste)
6. \$ git log --oneline



EXERCÍCIO

Escolha um arquivo e faça uma alteração qualquer nele para “evidenciar” o seu teste. Crie uma marcação “anotada” para o branch de teste e verifique o log com a opção --oneline e o arquivo.

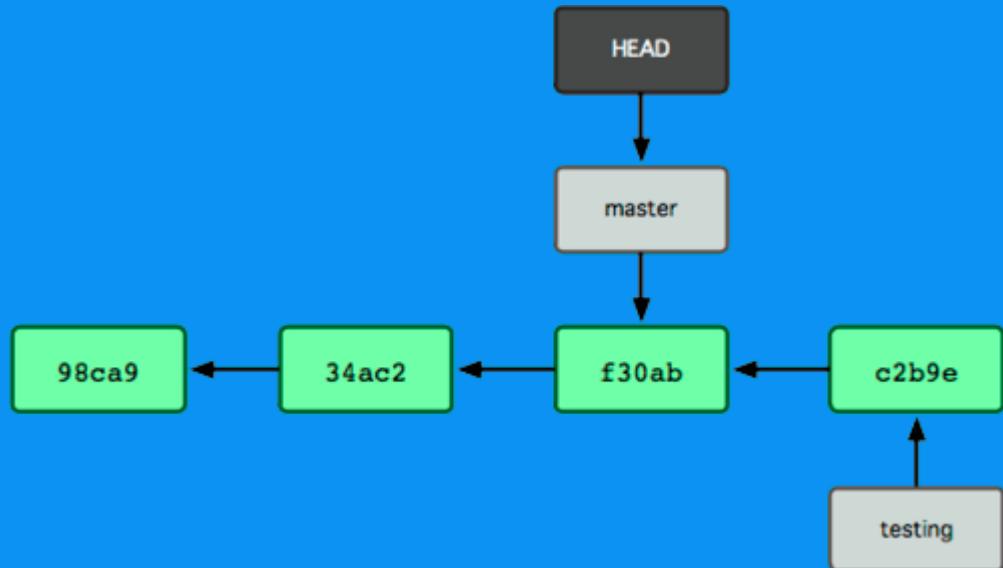
1. \$ vi <arquivo>
2. \$ git commit -am “meu-teste”
3. \$ git tag -a v2.0.1 “teste1”
4. \$ git log --oneline
5. \$ cat <arquivo>



EXERCÍCIO

Volte para o branch master, verifique o seu log --oneline e seu arquivo de teste.

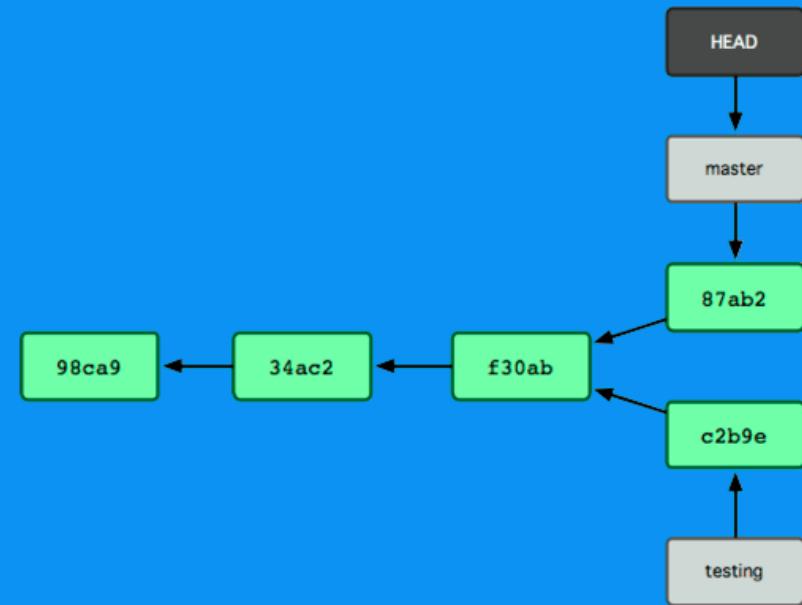
1. \$ git checkout master
2. \$ git log --oneline
3. \$ cat <arquivo>



EXERCÍCIO

Agora na master, realize uma alteração no mesmo arquivo e faça um commit. Crie a marcação 2.1, verifique o log --oneline e o arquivo.

1. \$ vi <arquivo>
2. \$ git commit -am “entrega 2.1”
3. \$ git tag -a v2.1 “nova entrega”
4. \$ git log
5. \$ cat <arquivo>



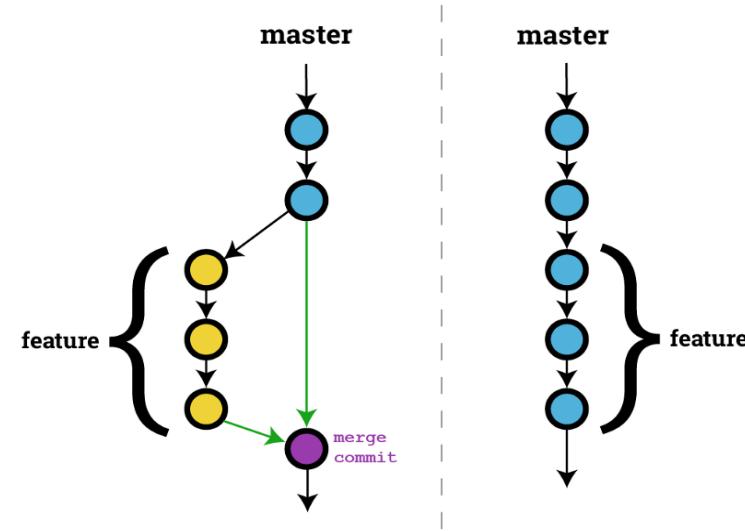
MERGES

Merge é a fusão entre dois ramos de desenvolvimento de alguma solução para o seu projeto.

A fusão pode ocorrer sem conflitos ou com conflitos.

Para comandar uma fusão ou merge é necessário ponteirar para o ramo desejado de receber o merge e realizar o comando:

```
$ git merge <ramo a ser integrado>
```



FAST FORWARD

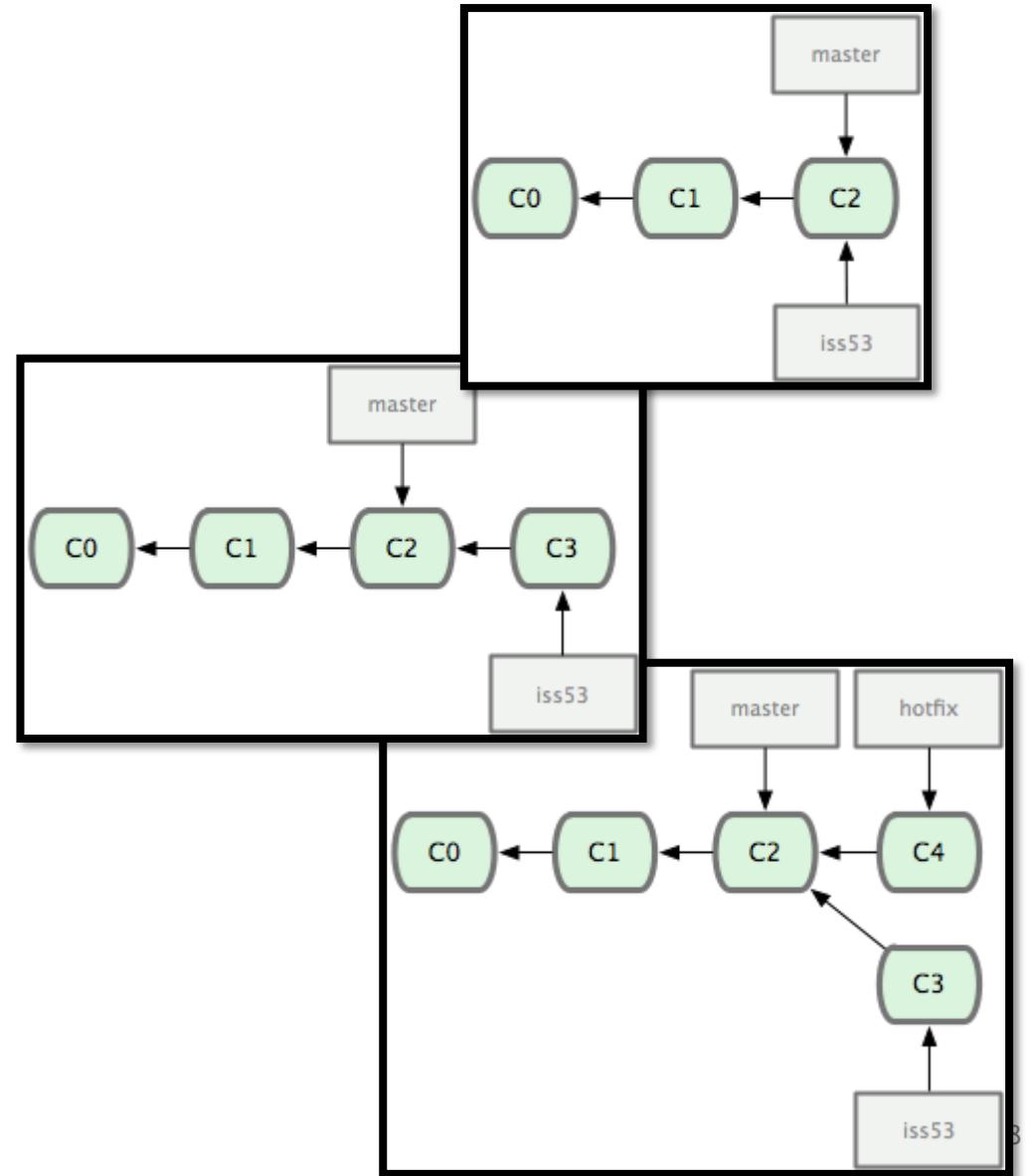
Você trabalha no seu projeto e já tem alguns commits. Entra uma necessidade ou uma issue #53 e você precisa criar um branch novo para trabalhar nesta demanda. Neste caso:

```
$ git checkout -b iss53
```

Você continua trabalhando e faz um commit. Ao fazer isso o HEAD da branch iss53 irá avançar.

Nesse momento você recebe uma ligação dizendo que **existe um problema com o web site** e você deve resolvê-lo imediatamente.

Em seguida, você tem uma **correção para fazer**. Vamos criar um branch para a correção (hotfix) para trabalhar até a conclusão.



FAST FORWARD

O fix é bem simples e você pode realizar um merge **SEM** conflitos com a master.

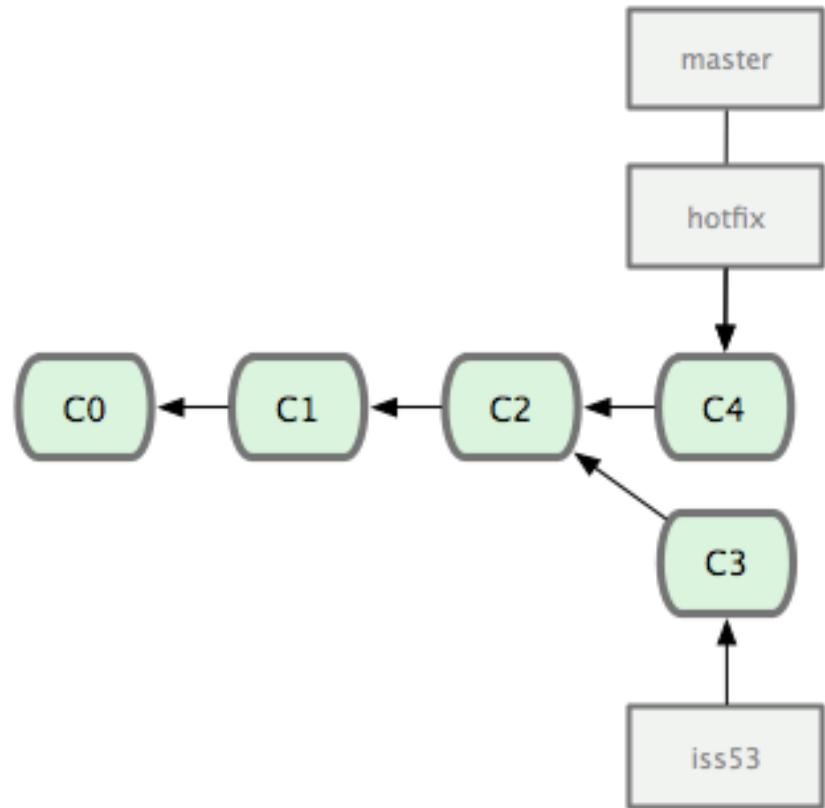
Para isso você precisa voltar a master e realizar o comando merge:

```
$ git checkout master
```

```
$ git merge hotfix
```

Quando você tenta fazer o merge de um commit com outro que **apontam para o mesmo commit anterior (C2)**, este pode acontecer seguindo o fluxo da **linha de tempo** do primeiro. Neste caso, o git simplifica as coisas movendo o ponteiro adiante porque **não existe modificações divergentes ou conflitos** para fazer o merge.

Isso é chamado de “fast forward”.



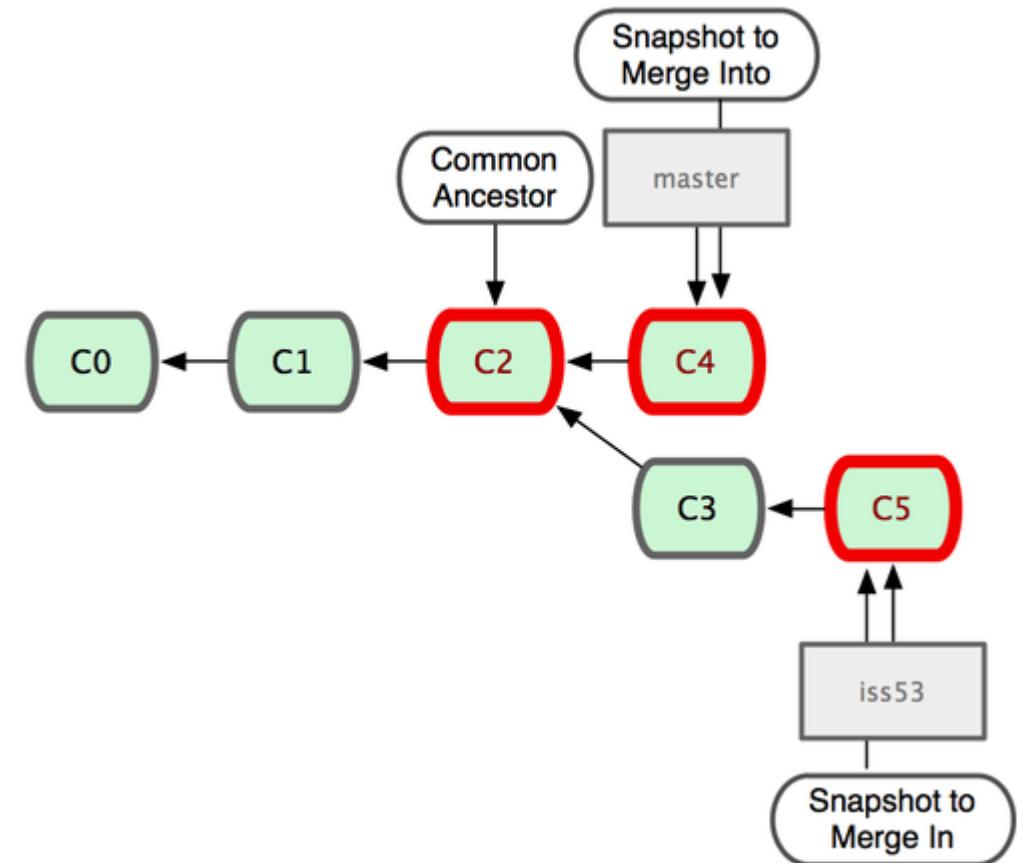
THREE WAY

Se o branch da issue 53 realizou um novo commit (C5), haverá conflito no merge.

Na **estratégia recursiva** (teoricamente conhecida como **three-way**), o git verifica o **HEAD**, que neste caso aponta os últimos commits de cada branch um parental DIFERENTE, ou seja, são dois pais diferentes:

Note que aqui há dois commits-pai (C2) E (C3).

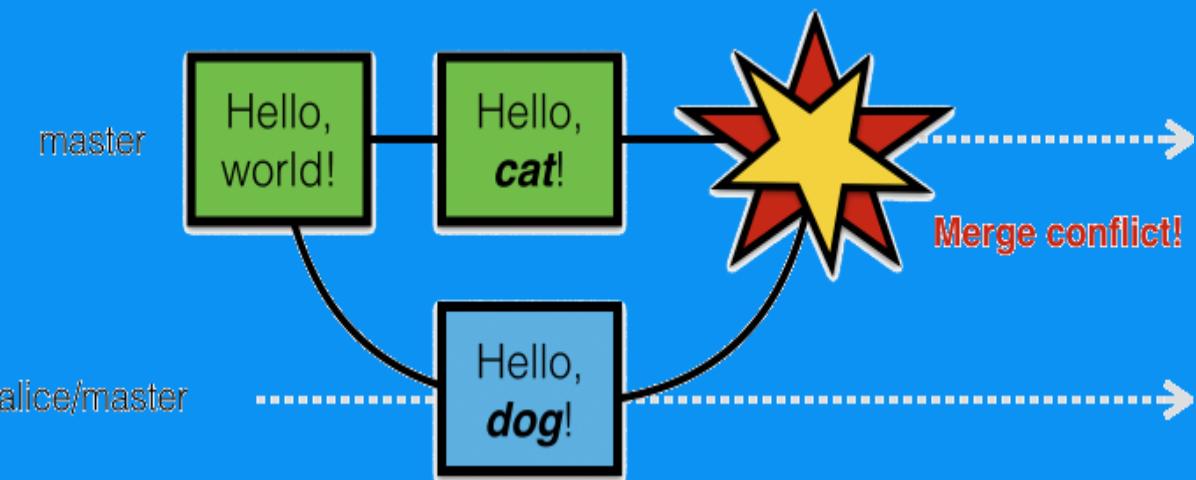
Caso não haja conflitos (como alterações na mesma linha), aplica-se as mudanças introduzidas pelos dois branches em um novo commit ou, caso haja, apresenta os conflitos para serem resolvidos.



EXERCÍCIO

Após a finalização do branch de teste, integre este branch com o branch de desenvolvimento principal (master). Perceba que o comando merge é seguido pela branch que você deseja que seja integrada.

1. \$ git checkout master
(caso não esteja na master)
2. \$ git merge teste



EXERCÍCIO

Escolha uma decisão para solução do conflito. Vá até o seu editor de preferência e edite o arquivo com o conteúdo desejado (limpe o que não for necessário). Verifique o log com as opções –oneline e --graph

1. Resolva o conflito (entre as marcações HEAD e teste)
2. \$ git commit -am “conflito resolvido”
3. \$ git branch
4. \$ git log

Obs.: A branch master que tinha 7 commits passou a ter 9 commits (meu teste e conflito resolvido) e a branch teste continua com os mesmos commits de antes). A branch teste não é mais necessária.

1. \$ git branch -d teste

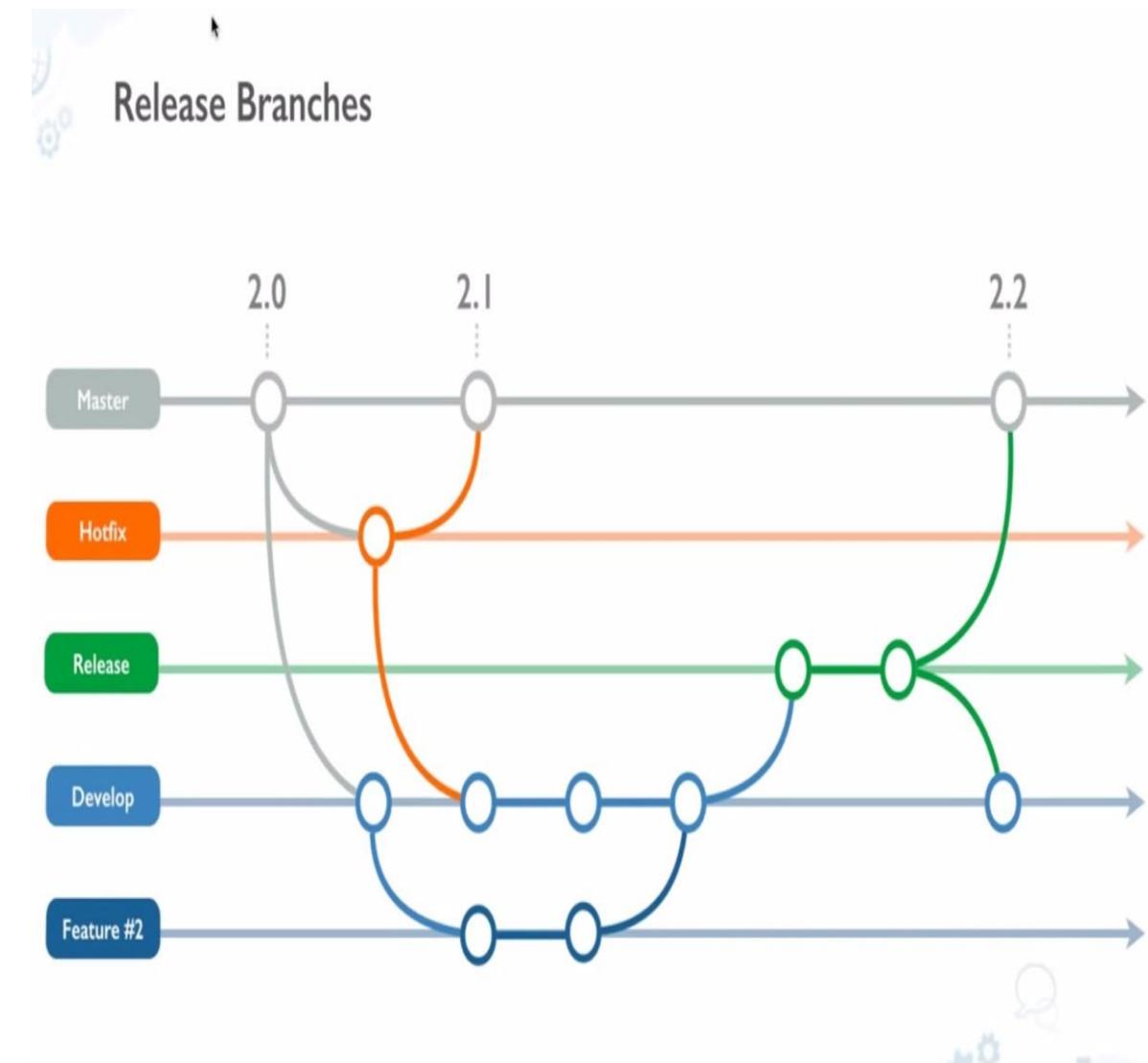


FLUXOS DE TRABALHO

Agora que você sabe o básico sobre criação e merge de branches, o que você pode ou deve fazer com eles? Resposta: fluxos de trabalho e entregas.

Os **branches longos** adotam a abordagem de ter somente código completamente estável em seus branches. A master é um tipo de branch longo e deve manter apenas a baseline atualizada do seu sistema.

branches paralelos chamados de release e develop são feitos para trabalhar evoluções do seu sistema e também são branches longos, normalmente os releases são versões candidatas do seu sistema e que fecham seu ciclo de vida quando “mergeados” com o branch master.



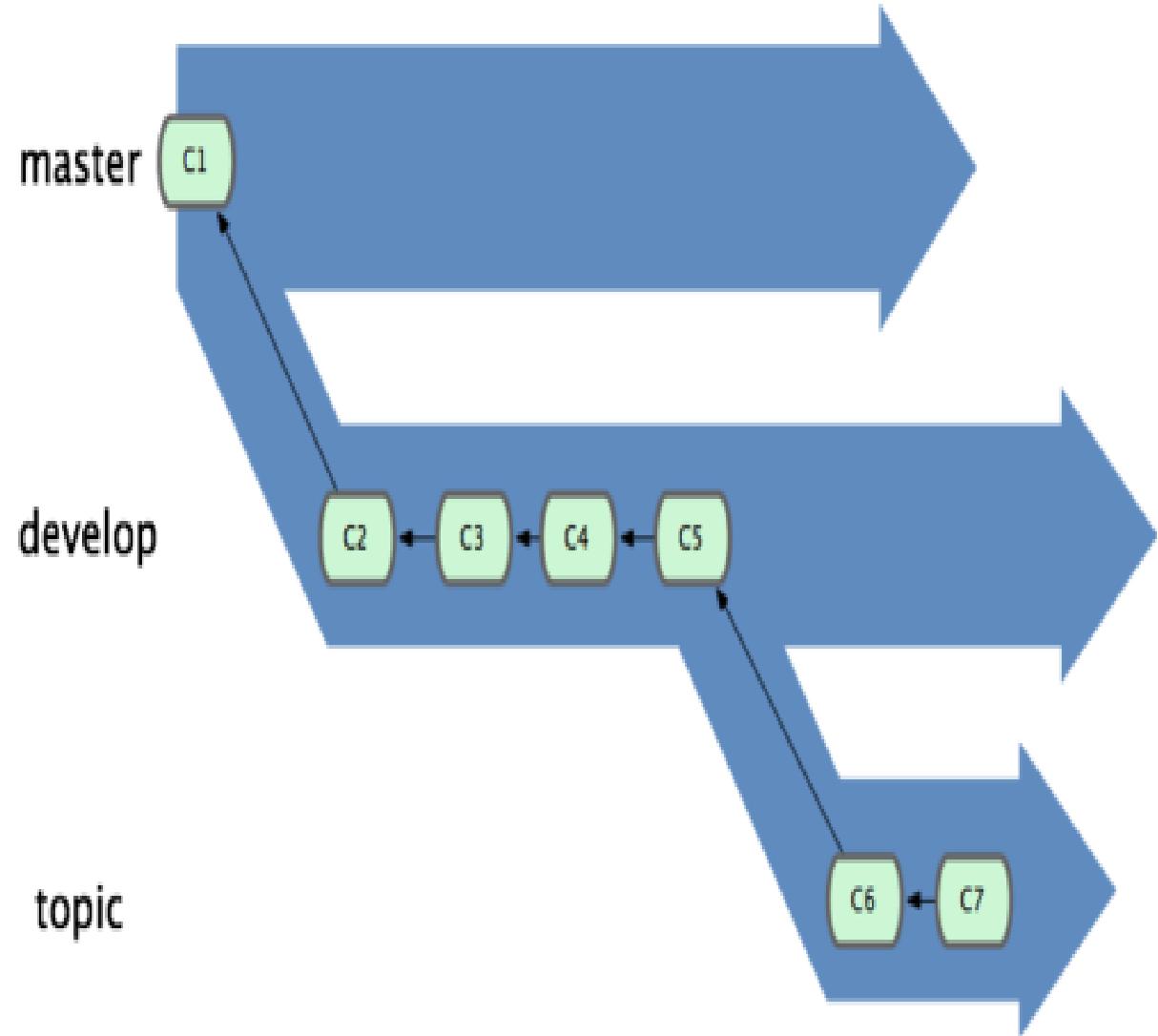
BRANCHES: CONTÊINERS DE TRABALHO

Normalmente é mais fácil pensar neles como um contêiner de trabalho, onde conjuntos de commits são promovidos a um contêiner mais estável quando eles são completamente testados.

Você pode continuar fazendo isso em vários níveis de estabilidade.

A ideia é que seus branches estejam em vários níveis de estabilidade; quando eles atingem um nível mais estável, é feito o merge no branch acima deles.

Repetindo, ter muitos branches de longa duração não é necessário, mas geralmente é útil, especialmente quando você está lidando com projetos muito grandes ou complexos.

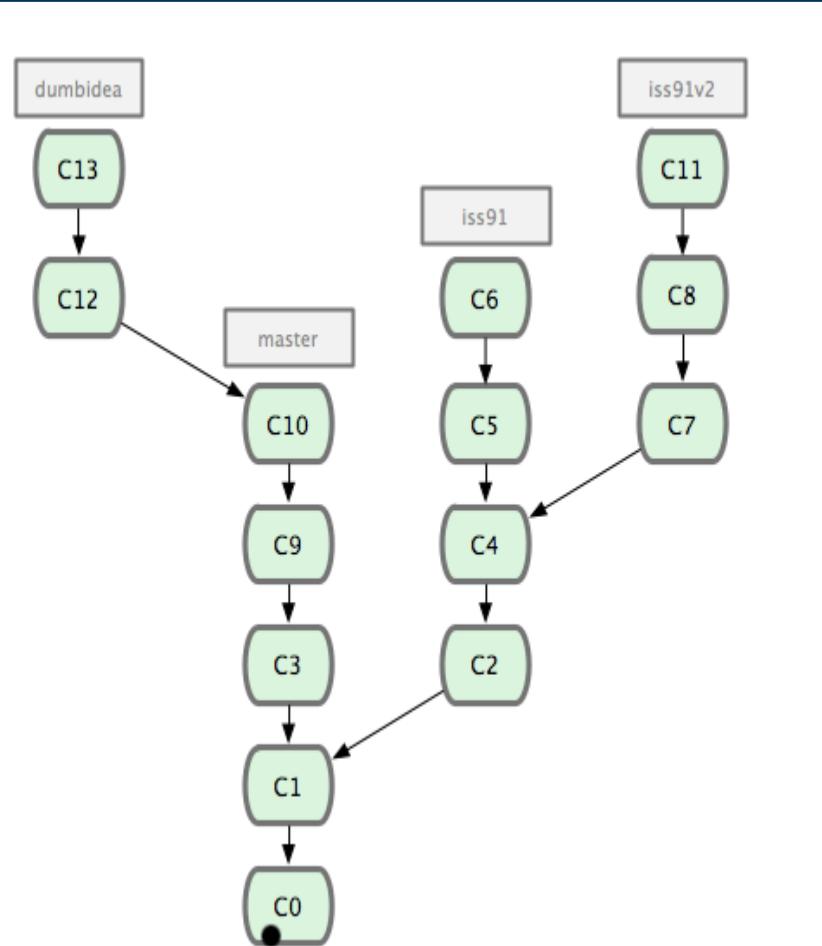


BRANCHES DE APOIO

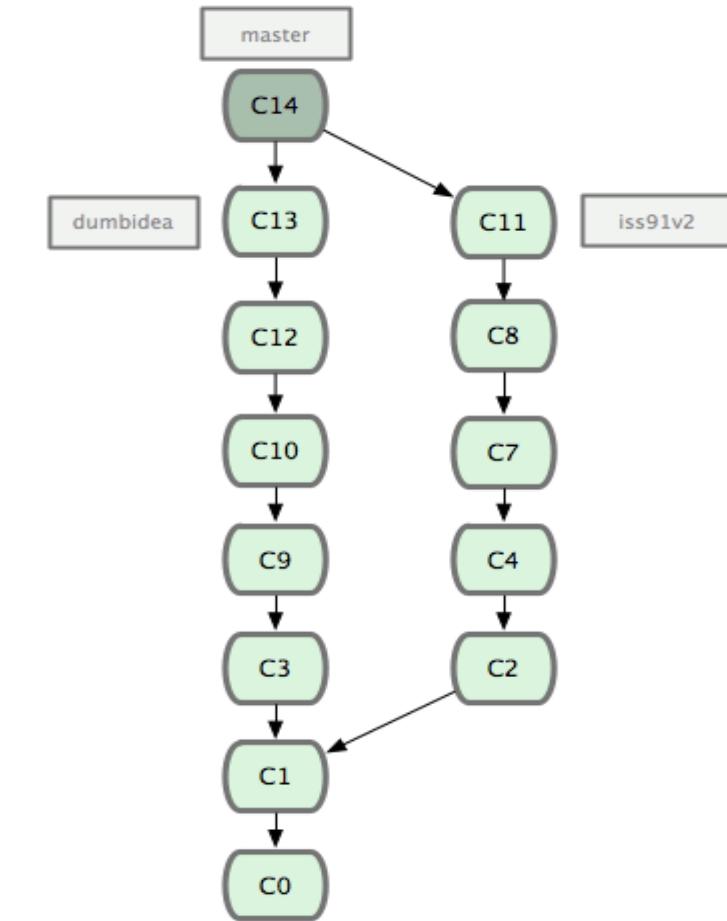
Branches de apoio são úteis em projetos de qualquer tamanho.

São branches de curta duração que você cria e usa para correção, ajustes, teste, uma funcionalidade simples ou um trabalho relacionado, como uma issue qualquer.

Se nas ferramentas de CVS isso era algo operacionalmente custoso, no git é comum criar, trabalhar, mesclar e apagar branches muitas vezes ao dia.



Seu histórico de commits com múltiplos branches tópicos



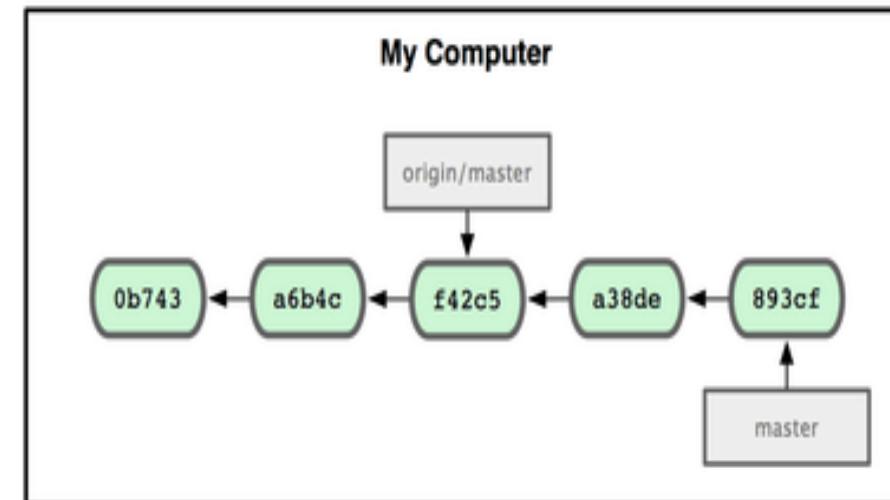
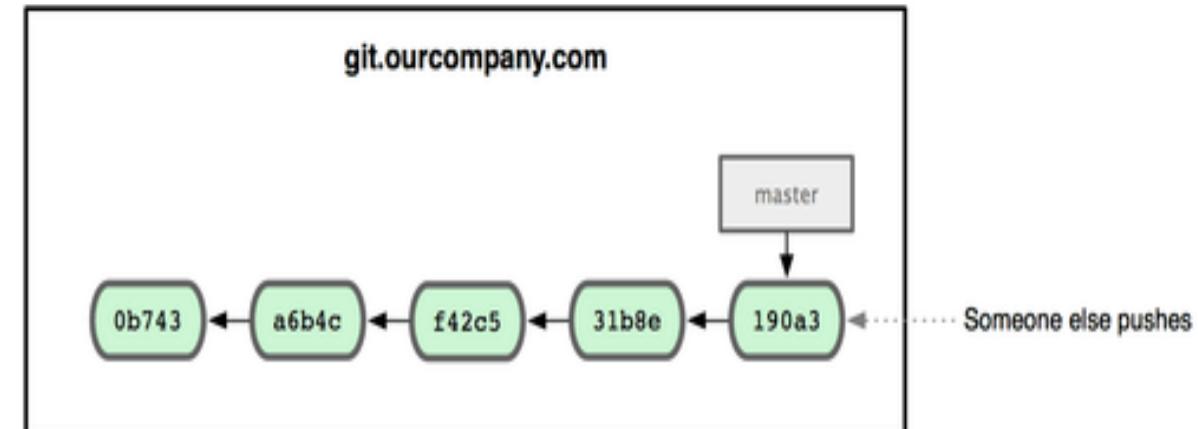
Seu histórico depois de fazer o merge de dumbidea e iss91v2

BRANCHES REMOTOS

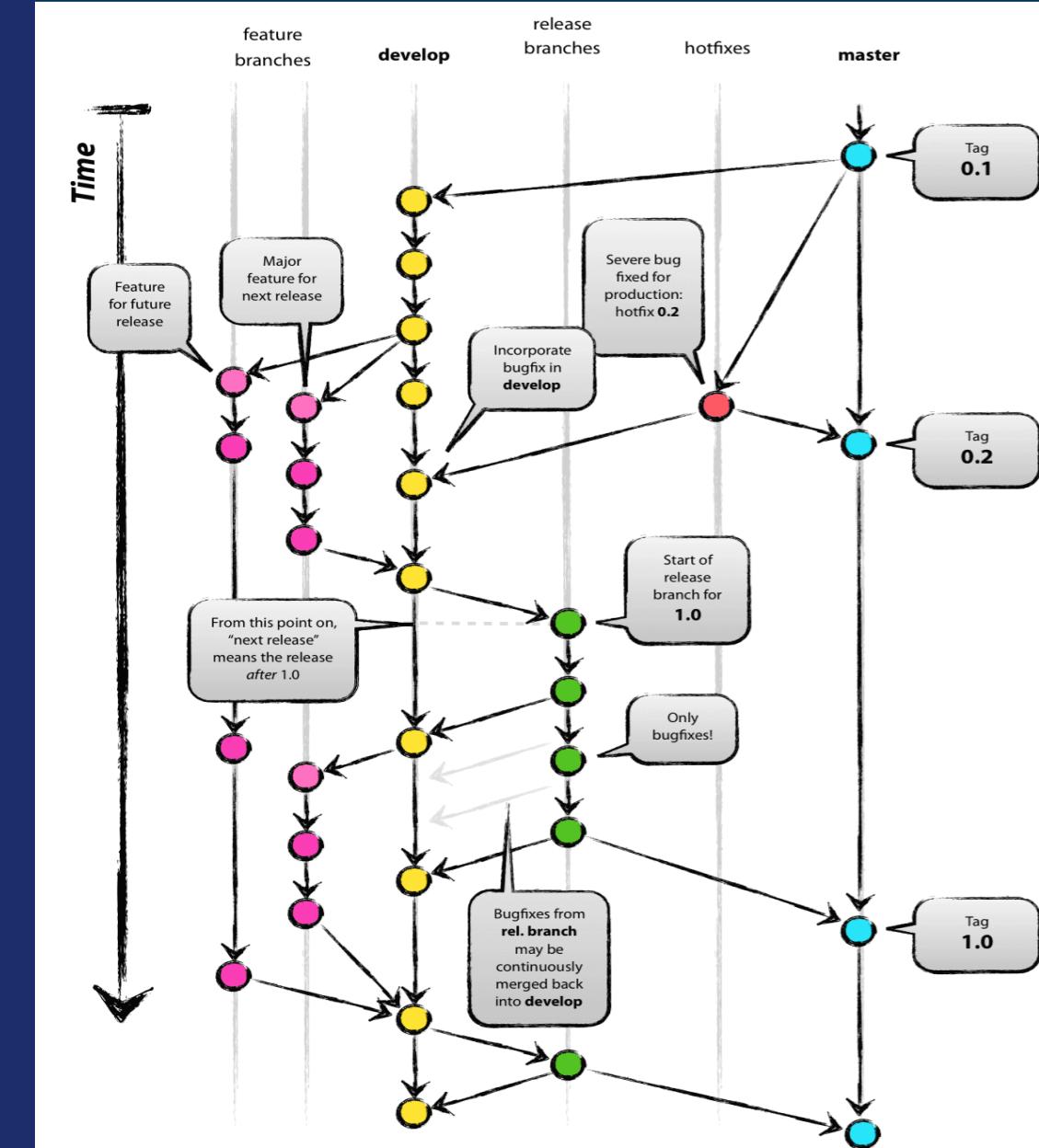
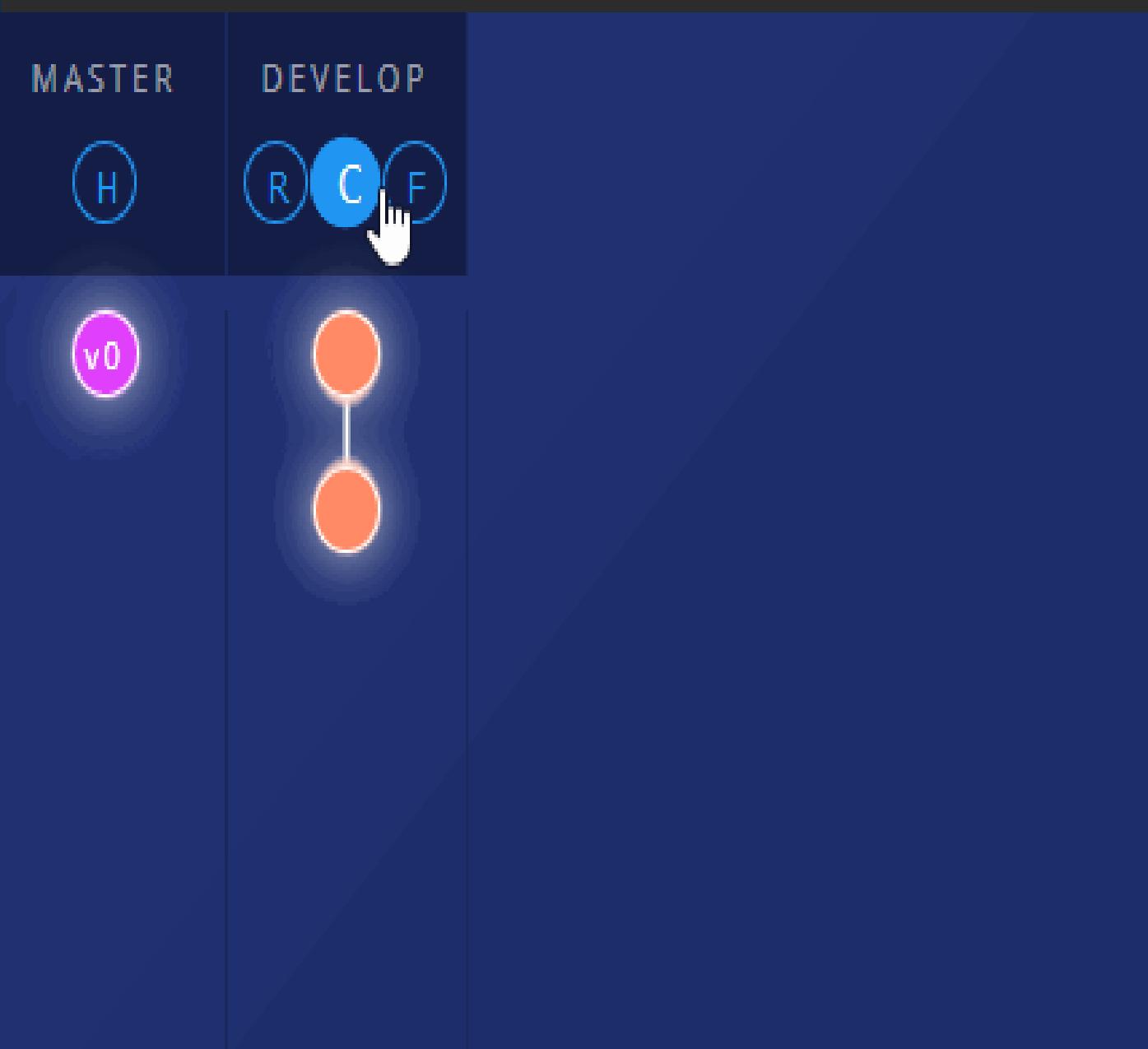
Branches remotos são **referências** ao estado de seus branches no seu repositório remoto. Branches remotos agem como marcadores para lembrá-lo onde estavam seus branches no seu repositório remoto na última vez que você se conectou a eles.

Digamos que você tem um servidor git na sua rede e você decide cloná-lo. O git automaticamente dá o nome **origin** para ele, baixa todo o seu conteúdo, cria uma referência para onde o branch master dele está, dá o nome **origin/master** para ele localmente; e você **não** pode movê-lo.

O git também dá seu próprio branch master como ponto de partida no mesmo local onde o branch master remoto está, a partir de onde você pode trabalhar.



GIT BRANCHING MODEL



REBASE

O commit não deve ser usado a qualquer momento, como um backup, mas principalmente como uma forma de garantir o histórico cronológico do projeto (project history).

Entre commits e merges, o histórico do seu projeto é construído.

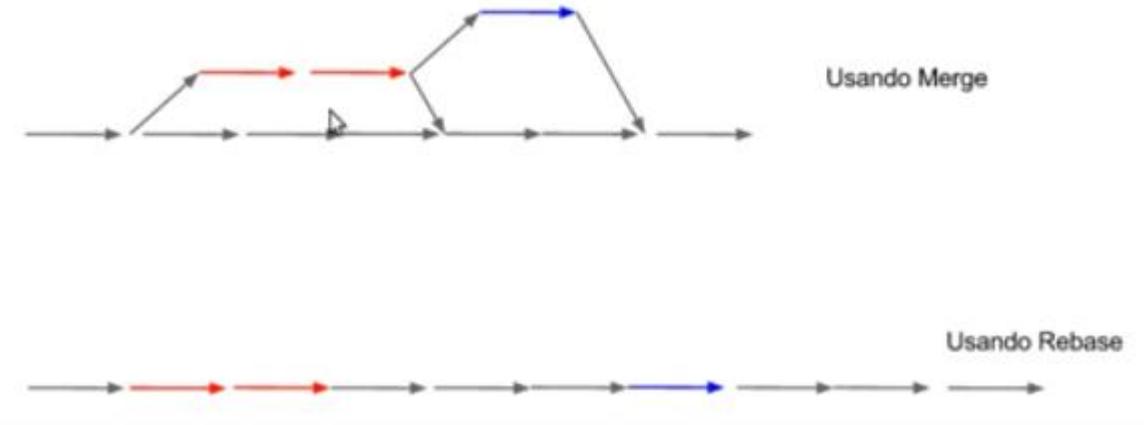
Mas com o passar do tempo e a dinâmica do processo de desenvolvimento é muito difícil manter um histórico ideal, limpo e organizado.

O rebase é um comando que tem um objetivo: reescrever o seu histórico e deixá-lo mais “linear”, mais inteligível do que um processo normal de merges.

\$ git rebase nome

<https://git-scm.com/book/pt-br/v1/Ramifica%C3%A7%C3%A3o-Branching-no-Git-Rebasing>

<https://git-scm.com/book/pt-br/v1/Ferramentas-do-Git-Reescrevendo-o-Hist%C3%B3rico>



GIT STASH

Pode acontecer de estar trabalhando em um problema e ter que largá-lo para focar em um bugfix.

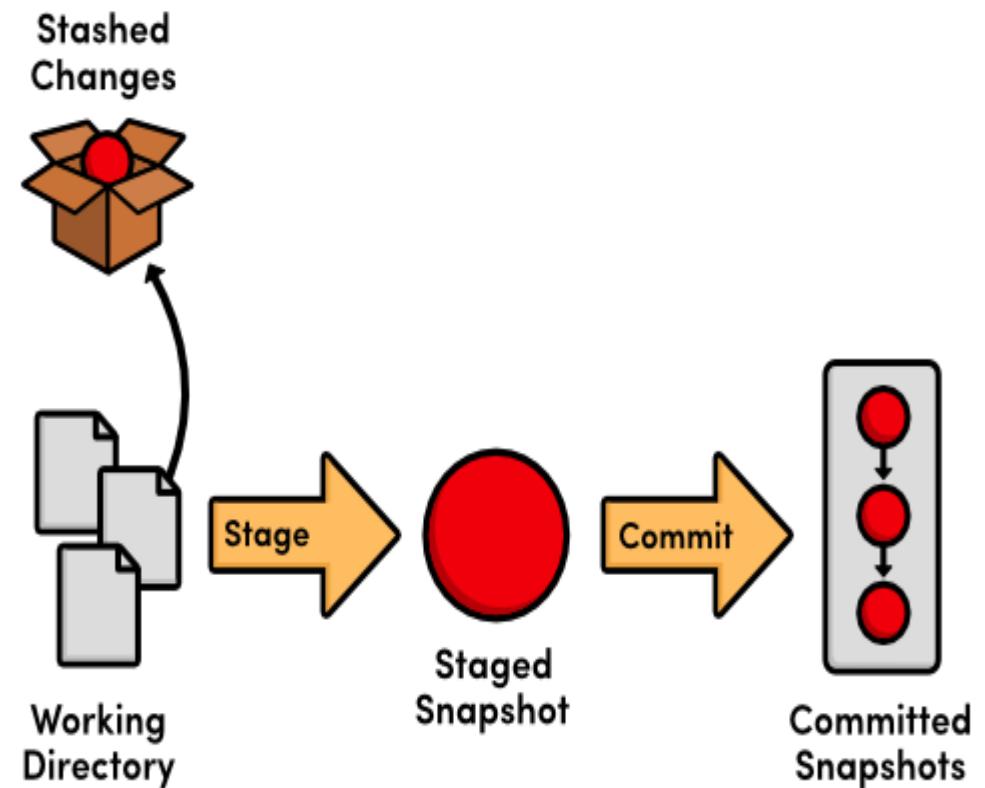
O problema é que o seu trabalho pode não estar em um estado ideal para realizar o commit na hora em que o bug aparece. Ao invés de fazer o commit só na metade do seu trabalho, o ideal seria você salvar o trabalho em progresso (wip) e assim, você já estaria livre para trabalhar no problema de maior prioridade.

E é exatamente nessa parte que o git stash se torna útil. Ele guarda arquivos da sua área de trabalho para que possam ser recuperados mais tarde.

\$ git stash

\$ git stash <arquivo>

<https://git-scm.com/book/pt-br/v1/Ferramentas-do-Git-Fazendo-Stash>



\$ git stash list

```
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
```

TRABALHANDO
COM REMOTOS

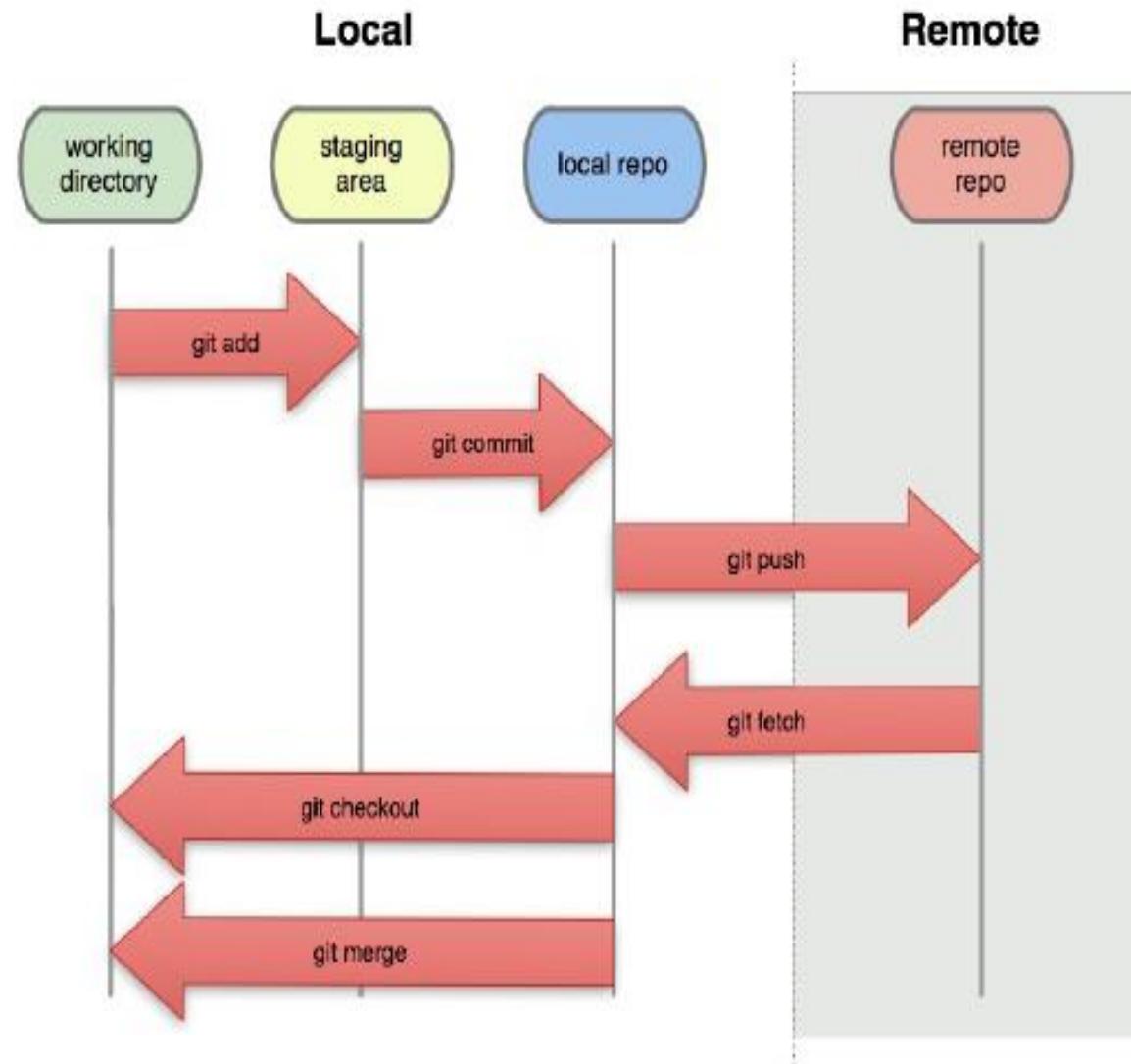
TRABALHANDO COM REMOTOS

Para ser capaz de colaborar com qualquer projeto no git, você precisa saber como gerenciar seus repositórios remotos.

Repositórios remotos são versões do seu projeto que estão hospedados na Internet ou intranet.

Para ver quais servidores remotos você configurou, você pode executar o comando `git remote`. Ele lista o nome de cada remoto que você especificou. Se você tiver clonado seu repositório, você deve pelo menos ver um chamado `origin` — esse é o nome padrão que o Git dá ao servidor de onde você fez o clone.

```
$ git remote  
$ git remote -v  
$ git remote show <nome>
```



EXERCÍCIO

Verifique que repositórios remotos pertencem ao seu projeto

1. \$ git remote
2. \$ git remote -v
3. \$ git remote show <nome>



GitLab

O **GitLab** é um gerenciador de repositório de software baseado em git, com suporte a:

- Wiki,
- gerenciamento de tarefas e
- CI/CD

GitLab é similar ao [GitHub](#), mas o GitLab permite que os desenvolvedores armazenem o código em seus próprios servidores, ao invés de servidores de terceiros.

Ele é software livre, distribuído pela Licença MIT. Está disponível como um pacote Omnibus, assim como um instalador simplificado provido pela [Bitnami](#) e pela Digital Ocean.

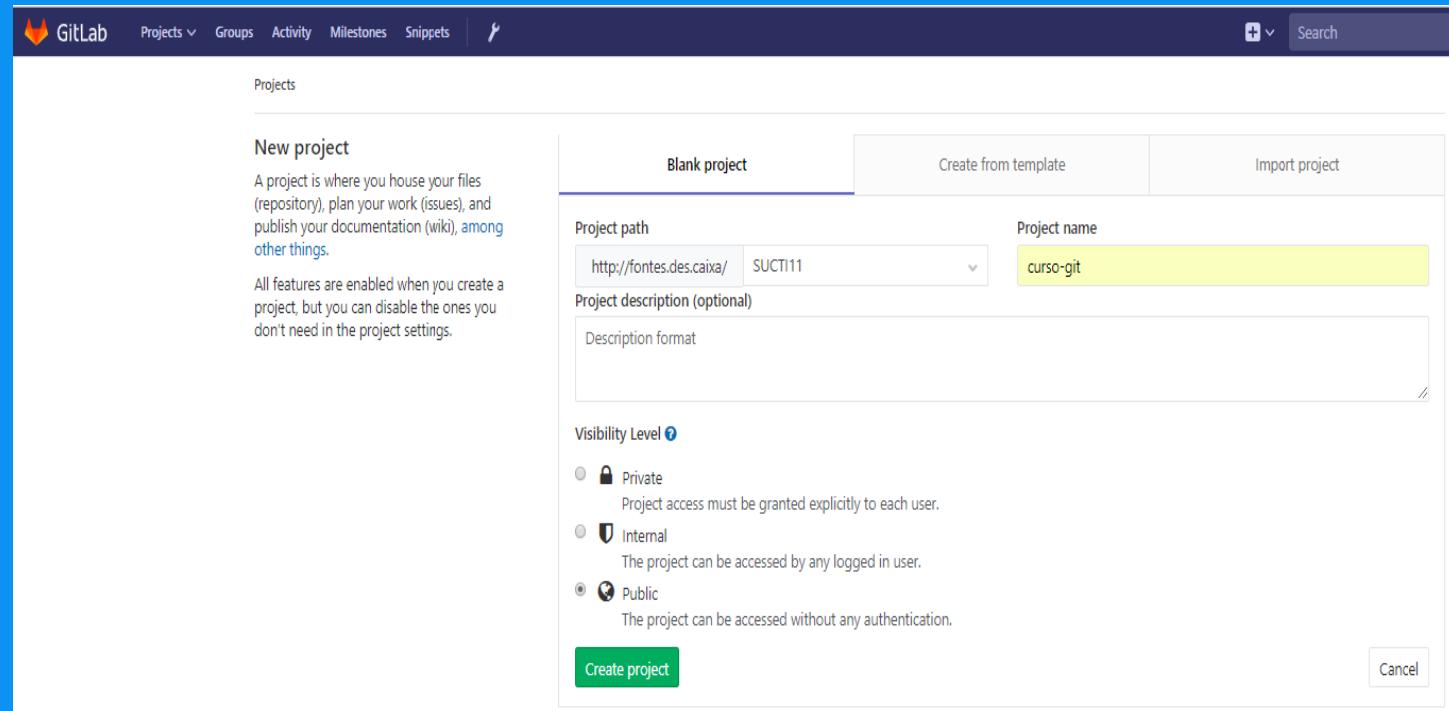
Permissões:

<https://docs.gitlab.com/ee/user/permissions.html>



EXERCÍCIO

Faça o login no GitLab, entre no seu grupo e crie um projeto (mantenedor) exclusivo para o treinamento (será deletado depois).



The screenshot shows the GitLab interface for creating a new project. At the top, there's a navigation bar with links for Projects, Groups, Activity, Milestones, Snippets, and a search bar. Below the navigation is a section titled 'Projects' with a sub-section for 'New project'. It explains what a project is and lists enabled features. A 'Blank project' tab is selected. In the 'Project path' field, the URL 'http://fontes.des.caixa/ SUCTI11' is entered. The 'Project name' field contains 'curso-git', which is highlighted with a yellow background. Below these fields are sections for 'Project description (optional)' and 'Description format'. Under 'Visibility Level', the 'Public' option is selected. At the bottom right of the form are 'Create project' and 'Cancel' buttons.



COPIE A URL

A screenshot of a GitLab project page for 'curso-git'. The page shows basic project details like 'Public' status, 'Add license', and a 'Copy URL to clipboard' button highlighted with a red arrow. Below the URL are buttons for 'Add Changelog', 'Add Contribution guide', 'Enable Auto DevOps', 'Add Kubernetes cluster', and 'Set up CI/CD'. The commit history shows a single commit by 'manuelnunespereira' titled 'criando arquivo'. The file list includes 'README.md' and 'arquivo.txt'. A large red box at the bottom right contains the text 'Copie a <url> do seu projeto!'.

GitLab

Projects Groups Activity Milestones Snippets

Manuel Nunes Pereira > curso-git > Details

C curso-git

Project Details Activity Security Dashboard Cycle Analytics

Repository

Issues 0

Merge Requests 0

CI / CD

Operations

Registry

Packages

Wiki

Snippets

Settings

C curso-git Public Add license

escreva o que quiser

Project ID: 9437026

0 ⭐ Star 0 Fork HTTPS <https://gitlab.com/Manup/curso-git> Copy URL to clipboard

Readme Files (92 KB) Commit (1) Branch (1) Tags (0) Security Dashboard

Add Changelog Add Contribution guide Enable Auto DevOps Add Kubernetes cluster Set up CI/CD

master curso-git / +

criando arquivo
Manuel Nunes Pereira authored 10 seconds ago

5607a7e4

Name Last commit Last update

README.md Initial commit 8 minutes ago

arquivo.txt criando arquivo just now

README.md

curso-git

escreva o que quiser

Copie a <url> do seu projeto!

COMANDO PUSH

Quando o seu PATCH estiver pronto para ser compartilhado, você pode encaminhar para o repositório remoto a sua COLABORAÇÃO.

O comando para isso é simples: `git push <nome do remoto> <branch>`.

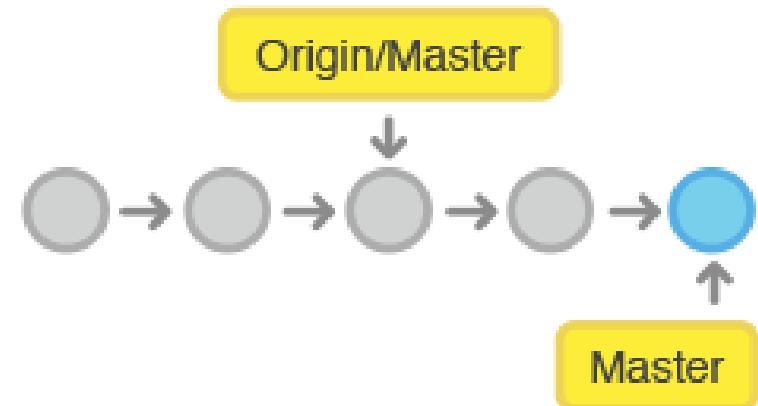
Por padrão, quando a branch não tem um nome definido ela adota a branch master

`$ git push <repo> <branch>`

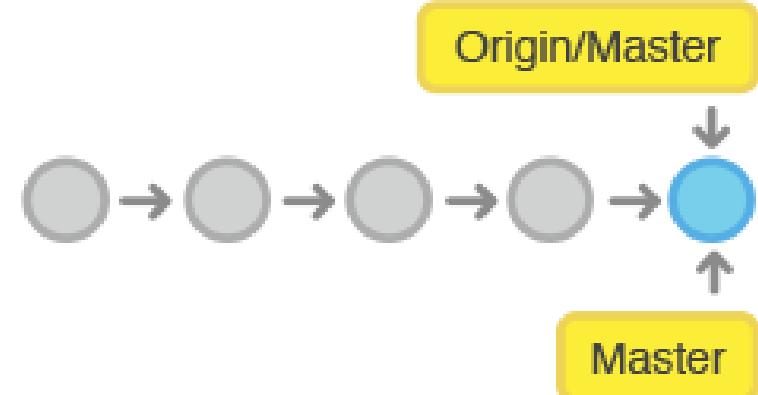
Lembre-se que você precisa ter permissionamento para enviar arquivos para o repositório remoto.

Ajuste as suas configurações de perfil, caso necessário.

Before Pushing



After Pushing



C curso-git

Projeto

Detalhes

Atividade

Painel de controle de segurança

Análise de Ciclo

Issues 0

Merge Requests 0

CI / CD

Operações

Registro

Pacotes

Wiki

Snippets

Git global setup

```
git config --global user.name "Manuel Nunes Pereira"  
git config --global user.email "manuel.nupe@gmail.com"
```

Create a new repository

```
git clone https://gitlab.com/Manupe61/curso-git.git  
cd curso-git  
touch README.md  
git add README.md  
git commit -m "add README"  
git push -u origin master
```

Existing folder

```
cd existing_folder  
git init  
git remote add origin https://gitlab.com/Manupe61/curso-git.git  
git add .  
git commit -m "Initial commit"  
git push -u origin master
```

Existing Git repository

```
cd existing_repo  
git remote rename origin old-origin  
git remote add origin https://gitlab.com/Manupe61/curso-git.git  
git push -u origin --all  
git push -u origin --tags
```

« Collapse sidebar

Remove project

EXERCÍCIO

Crie uma pasta chamada curso, acesse-a via git e faça o primeiro bloco de comandos para fazer as configurações globais necessárias.

Git global setup

```
git config --global user.name "Manuel Nunes Pereira"  
git config --global user.email "manuel.nupe@gmail.com"
```



EXERCÍCIO

Faça agora o segundo bloco de comandos para criação do novo repositório git dentro da pasta curso. Vá na opção repositório (projeto) e verifique.

Create a new repository

```
git clone https://gitlab.com/Manupe61/curso-git.git  
cd curso-git  
touch README.md  
git add README.md  
git commit -m "add README"  
git push -u origin master
```

Manuel Nunes Pereira > curso-git > Repositório

master / +



add README

Manuel Nunes Pereira fez commit 2 minutos atrás

Nome

Último commit

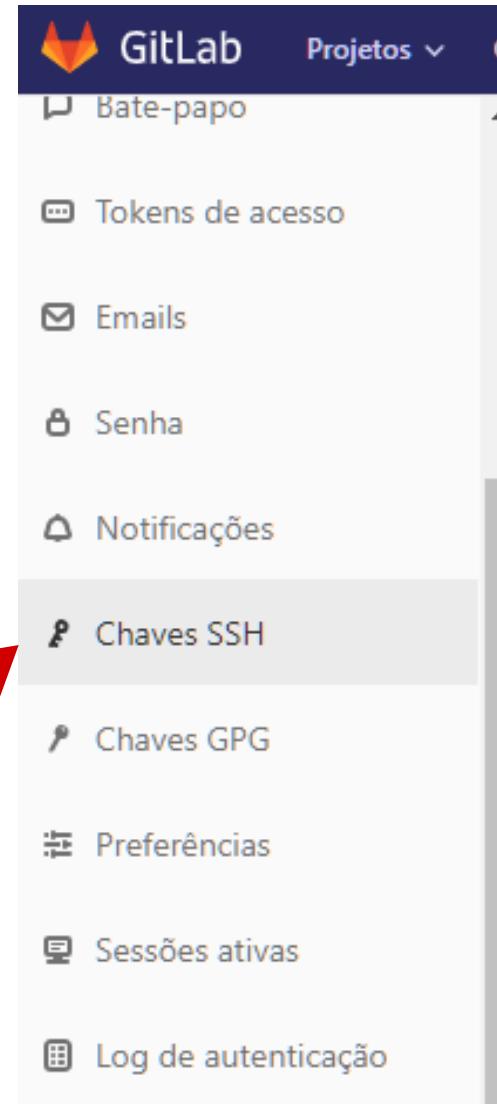
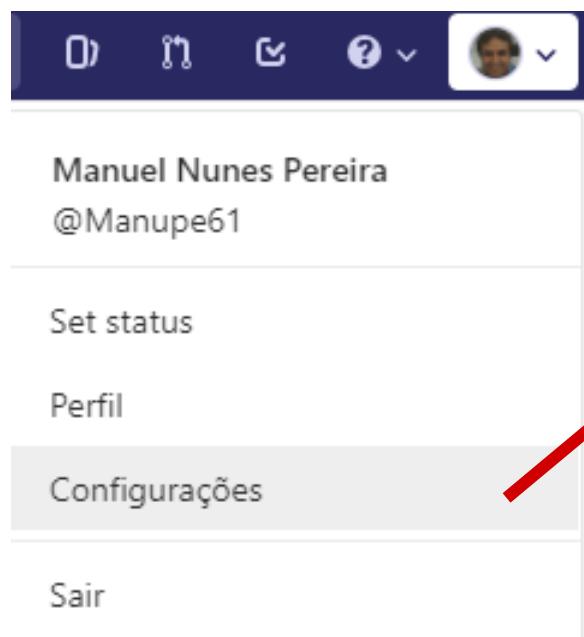
README.md

add README



CHAVE SSH

Para proporcionar mais segurança de conexão e controle no repositório remoto, você pode adicionar uma chave criptográfica do tipo SSH, onde a chave pública fica no servidor e a chave privada no seu computador.



Generating a new SSH key pair

Before creating an SSH key pair, make sure to read about the [different types of keys](#) to understand their differences.

To create a new SSH key pair:

1. Open a terminal on Linux or macOS, or Git Bash / WSL on Windows.

2. Generate a new ED25519 SSH key pair:

```
ssh-keygen -t ed25519 -C "email@example.com"
```

Adding an SSH key to your GitLab account

1. Copy your **public** SSH key to the clipboard by using one of the

macOS:

```
pbcopy < ~/.ssh/id_ed25519.pub
```

WSL / GNU/Linux (requires the xclip package):

```
xclip -sel clip < ~/.ssh/id_ed25519.pub
```

Git Bash on Windows:

```
cat ~/.ssh/id_ed25519.pub | clip
```

Configurações do Usuário > SSH Keys

SSH Keys

SSH keys allow you to establish a secure connection between your computer and GitLab.

Add an SSH key

Para adicionar uma chave SSH, você precisa [gerar uma ou usar uma chave existente](#).

Key

Cole a sua chave SSH pública, que geralmente é encontrada no arquivo '`~/.ssh/id_rsa.pub`' e começa com '`ssh-rsa`'. Não use a sua chave SSH privada.

Geralmente se inicia com "ssh-rsa ..."

Título

por exemplo, Chave do meu MacBook

Nomeie sua chave individual por meio de um título

[Adicionar chave](#)

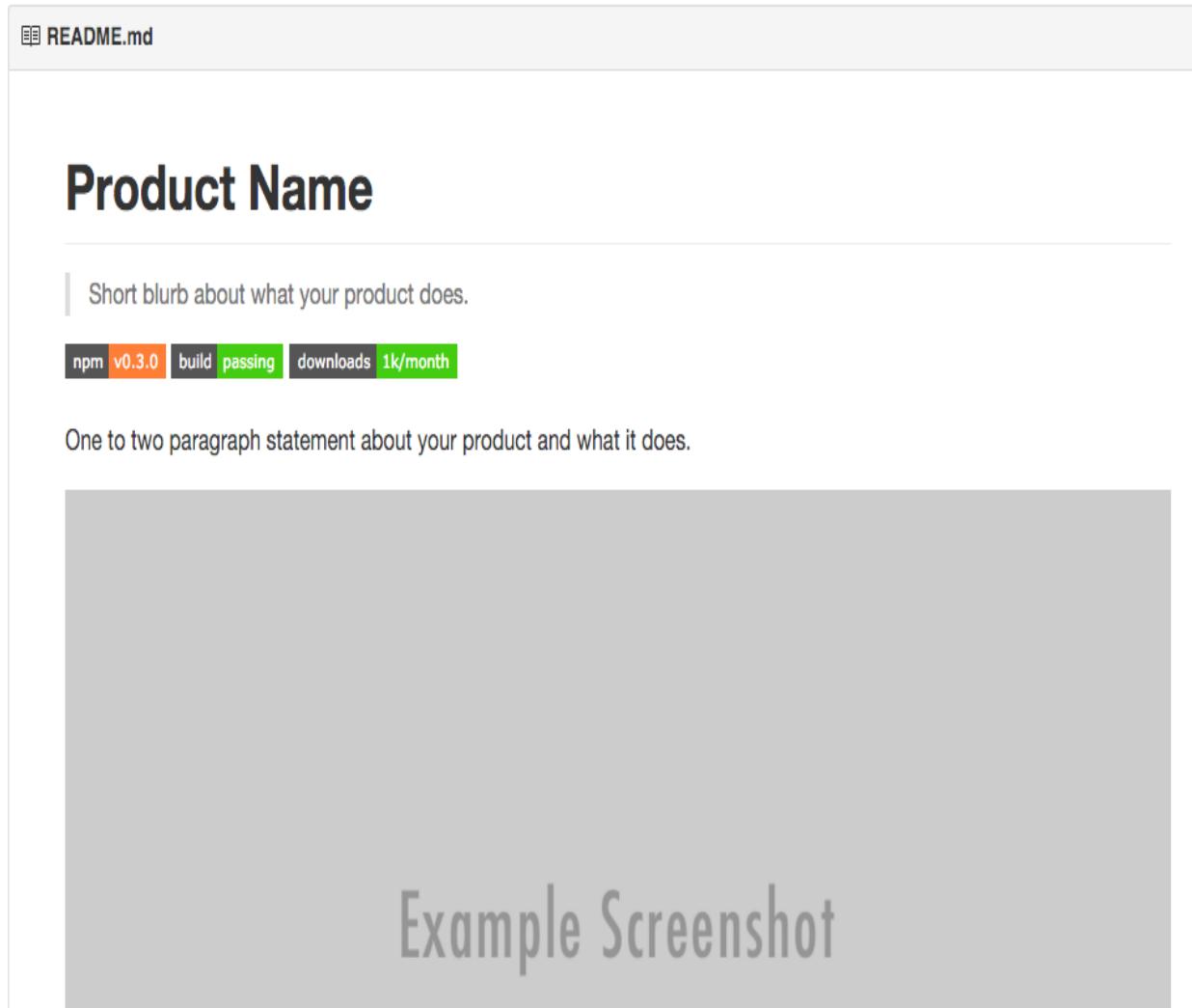
README.md

Lembre-se que você pode criar um arquivo `readme.md` tanto localmente como no repositório remoto.

Atente para o fato de que seu arquivo pode ser visto no seu repositório remoto e procure utilizar uma notação adequada, neste caso a notação `MarkDown` (MD).

`Markdown` é uma ferramenta de conversão de `text-to-HTML` que possibilita a marcação de títulos, listas, tabelas, etc., de forma muito limpa, legível e precisa.

Ainda que seja uma “conversão” ele não é a marcação `HTML`, como colocar colocar atributos nos elementos (`class`, `id`, `title`, etc.)





MARKDOWN CHEATSHEET

themeSPECTRE

themespectre.com

| A HEADERS | LINKS | ... HORIZONTAL RULES | | | | | | | | | | | | |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|-----------------|------------|---------|------------|--------|--------------|----------|------------------|----------|--------------|---------------|-----------------------------------------------------------|
| <pre># Header 1 ## Header 2 ### Header 3</pre> | <p>[Google](http://www.google.com/)</p> <p>[Google](http://www.google.com/ "Google")</p> | <p>* * *</p> <p>---</p> <p>***</p> <p>---</p> | | | | | | | | | | | | |
| LISTS | IMAGES | BI EMPHASIS | | | | | | | | | | | | |
| <p>Undordered Ordered</p> <ul style="list-style-type: none"> * List Item 1 1. List Item 1 * List Item 2 1. List Item A * List Item A | <p>![Alt text](/path/to/image.jpg)</p> <p>![Alt text](/path/to/image.jpg "Title")</p> <p>[img1]: /path/to/img.jpg "Title" ![Alt text][img1]</p> | <p>*Emphasis* **Strong**</p> <p>_Emphasis_ __Strong__</p> <p>*Super*emphasis</p> <p>**Super**strong</p> | | | | | | | | | | | | |
| BLOCK QUOTES | { } ESCAPABLE CHARACTERS | </> INLINE CODE | | | | | | | | | | | | |
| <pre>> Lorem ipsum dolor sit amet > Lorem ipsum > >> Lorem ipsum</pre> | <table> <tbody> <tr> <td>\ Backslash</td> <td>() Parenthesis</td> </tr> <tr> <td>` Backtick</td> <td># Pound</td> </tr> <tr> <td>* Asterisk</td> <td>+ Plus</td> </tr> <tr> <td>_ Underscore</td> <td>- Hyphen</td> </tr> <tr> <td>{ } Curly braces</td> <td>. Period</td> </tr> <tr> <td>[] Brackets</td> <td>! Exclamation</td> </tr> </tbody> </table> | \ Backslash | () Parenthesis | ` Backtick | # Pound | * Asterisk | + Plus | _ Underscore | - Hyphen | { } Curly braces | . Period | [] Brackets | ! Exclamation | <p>Use `<div>` tags</p> <p>``echo `uname -a````</p> |
| \ Backslash | () Parenthesis | | | | | | | | | | | | | |
| ` Backtick | # Pound | | | | | | | | | | | | | |
| * Asterisk | + Plus | | | | | | | | | | | | | |
| _ Underscore | - Hyphen | | | | | | | | | | | | | |
| { } Curly braces | . Period | | | | | | | | | | | | | |
| [] Brackets | ! Exclamation | | | | | | | | | | | | | |
| CODE BLOCK | Normal text | #include <stdio.h> | | | | | | | | | | | | |

MANTER REPOSITÓRIOS REMOTOS

git remote

Para adicionar seu repositório remoto no git:

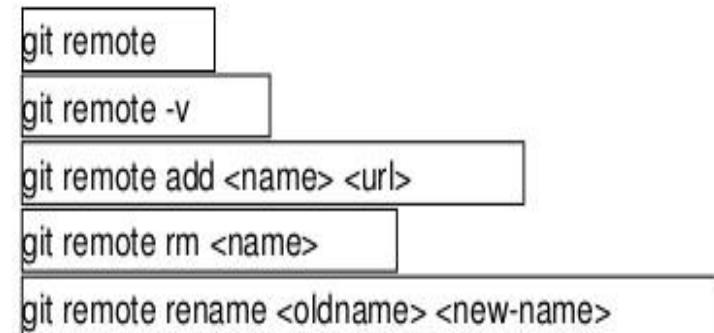
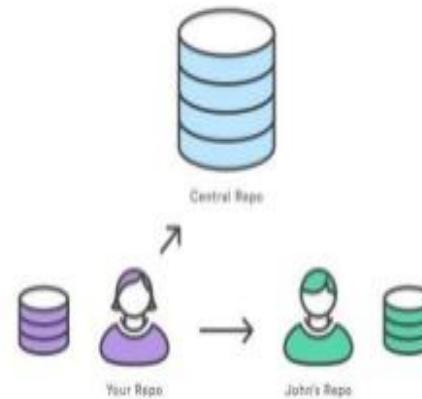
```
$ git remote add <nome> <url>
```

Para modificar um apelido de um remoto:

```
$ git remote rename <nome> <novo nome>
```

Se você quiser remover uma referência:

```
$ git remote rm <nome>
```



EXERCÍCIO

Vá ao endereço:

<https://github.com/dbader/readme-template/blob/master/portuguese/README.md>

Veja um modelo de README.md. Copie, cole e ajuste o seu arquivo local com informações que considere relevante. Atualize o projeto remoto com um nome “gitlab” e atualize.

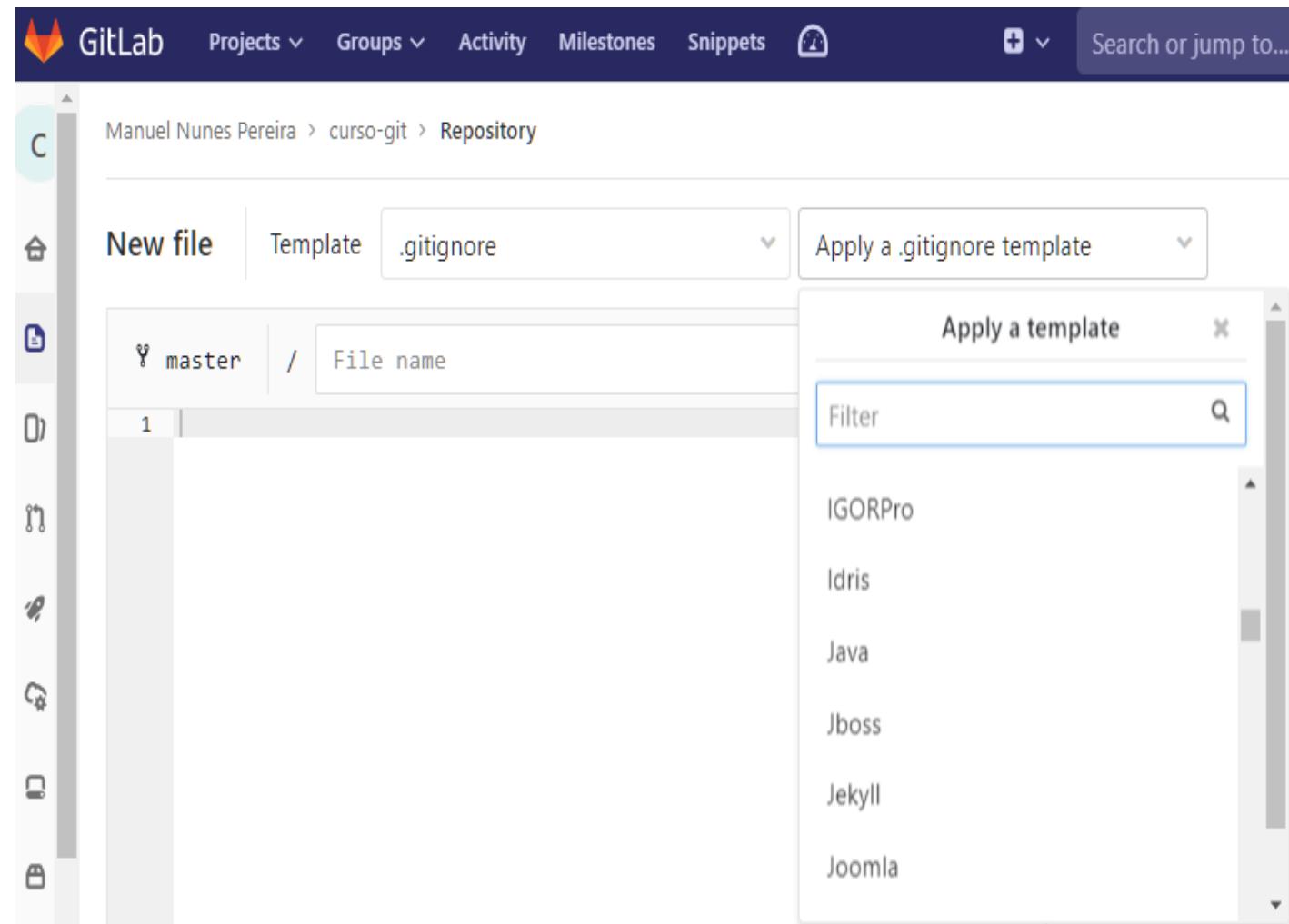
1. Altere arquivo README.md
2. \$ git commit –am “ajustando README”
3. \$ git remote rename origin gitlab
4. \$ git push gitlab master
5. Verifique REDME.md no seu repositório remoto.



.GITIGNORE NO GITLAB

Lembre-se que você pode criar um arquivo .gitignore tanto localmente como no repositório remoto.

Mas ao criar no GitLab ou GitHub, este te oferece modelos (templates) de construção que podem lhe ser úteis na hora da confecção do seu arquivo.



EXERCÍCIO

Inclua um arquivo do tipo `.gitignore`. Escolha o modelo que quiser do seu repositório remoto.

A screenshot of a GitLab repository page for 'curso-git'. The top navigation bar shows the repository name, ID (9756779), and basic statistics (0 stars, 0 forks). Below the navigation, there are links for 'Leia-me', 'Arquivos (82 KB)', 'Commit (1)', 'Branch (1)', 'Tags (0)', and 'Padrões'. A 'Novo arquivo' button is highlighted with a red box and a red arrow pointing from the previous image. The page also features an 'Auto DevOps' section with a cloud icon and a 'Novo arquivo' button.

A screenshot of a 'Choose a template type' dialog box in GitLab. The dialog is titled 'Choose a template type' and shows a list of options: '.gitignore' (which is selected and highlighted with a red box and a red arrow), '.gitlab-ci.yml', 'Dockerfile', and 'LICENSE'. The background shows the 'New file' interface with 'master' selected and a 'File' input field.



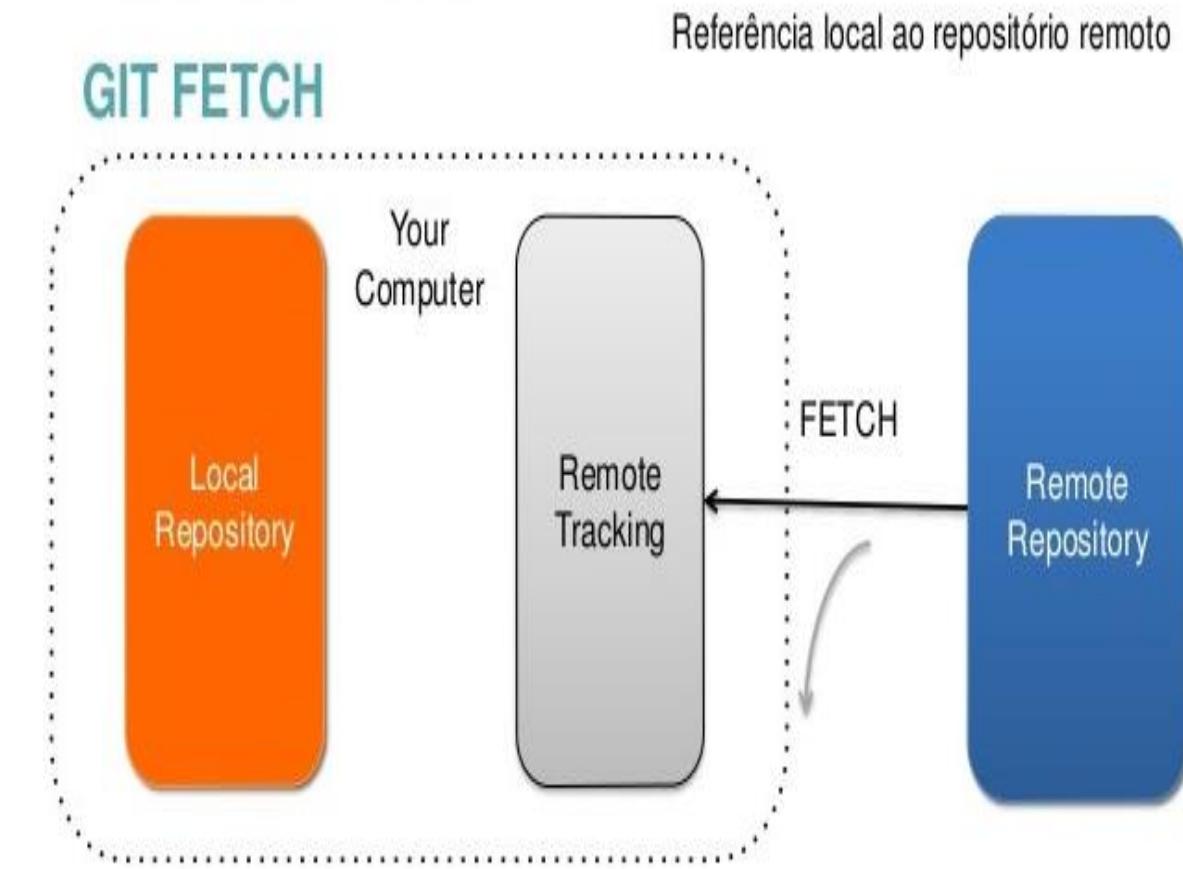
COMANDO FETCH

Se você clonar um repositório, o comando automaticamente adiciona o remoto com o nome origin.

Então, git fetch origin busca qualquer novo trabalho que foi enviado para esse servidor (ou outro qualquer) desde que você o clonou ou atualizou.

É importante notar que o comando fetch traz as referências para o seu repositório local — ele não faz o merge automaticamente com o seus dados ou modifica o que você está trabalhando atualmente.

\$ git fetch <nome>



O FETCH apenas atualiza a sua referência ao repositório remoto na área chamada de: "Remote Tracking"

EXERCÍCIO

Faça um comando fetch com o diretório remoto e verifique a pasta do seu projeto e o seu repositório local. Verifique o log.

1. \$ git fetch gitlab
2. Verifique a pasta do seu projeto e repositório local
3. \$ git log



ANÁLISE DO EXERCÍCIO

Com o comando fetch, o novo projeto adicionado ou atualizações, não fazem parte do seu diretório de trabalho.

As referências estão no arquivo FETCH_HEAD e nas pastas:

.git/refs/remote ou
.git/log/ref/remote

Dica:

Sempre que possível é recomendado utilizar o comando

\$ git remote show <nome>

```
Administrador@DESKTOP-037U9EO MINGW64 /c/cursos/Git/sitreina-git (master)
$ git fetch origin
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From git://github.com/Manupe61/sitreina-git
  4208c23..870a1c0  master      -> origin/master
```

```
Administrador@DESKTOP-037U9EO MINGW64 /c/cursos/Git/sitreina-git (master)
$ git remote show meu-remoto
* remote meu-remoto
  Fetch URL: https://github.com/Manupe61/sitreina-git.git
  Push  URL: https://github.com/Manupe61/sitreina-git.git
  HEAD branch: master
  Remote branch:
    master tracked
  Local ref configured for 'git push':
    master pushes to master (local out of date)
```

COMANDO PULL

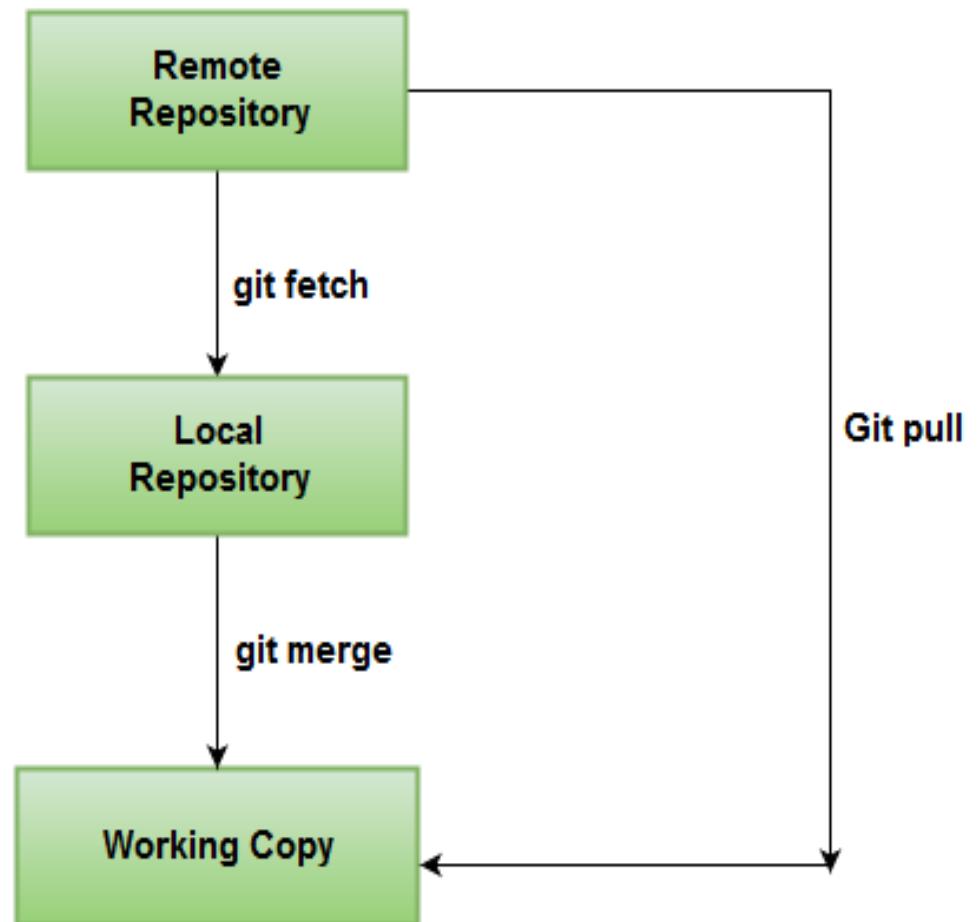
O git fetch baixa os HEADs com nomes ou tags de um ou mais repositórios (caso você tenha outro remote além do origin configurado).

Basicamente ele atualiza as referências locais com relações às remotas, mas não faz o merge com o branch local.

O git pull incorpora mudanças de um repositório remoto para o branch local.

Equivale a um git fetch seguido de git merge `FETCH_HEAD`.

\$ `git pull <nome> <branch>`



EXERCÍCIO

Faça um comando pull com o diretório remoto e verifique a pasta do seu projeto. Verifique o log.

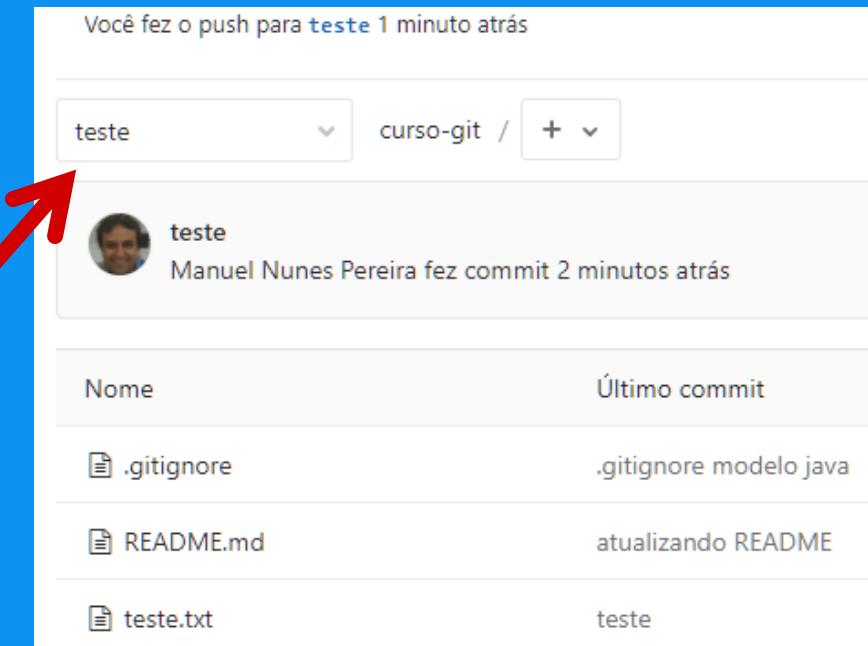
1. \$ git pull gitlab master
2. Verifique a pasta do seu projeto
3. \$ git log



EXERCÍCIO

Crie um arquivo de teste e crie uma branch de teste. Envia-o para o remoto numa branch de teste.

1. Criar arquivo teste com conteúdo teste
2. \$ git checkout -b "teste"
3. \$ git add .
4. \$ git commit -m "teste"
5. \$ git push gitlab teste
6. Verificar no remoto o branch "teste"



A screenshot of a GitLab repository interface. At the top, a message says "Você fez o push para teste 1 minuto atrás". Below it, the repository path is shown as "teste curso-git / +". A red arrow points from the text "Verificar no remoto o branch 'teste'" to the repository name "teste" in the path bar. The main area shows a commit by "Manuel Nunes Pereira fez commit 2 minutos atrás" with the message "teste". Below the commit, there's a table with file details:

| Nome | Último commit |
|------------|------------------------|
| .gitignore | .gitignore modelo java |
| README.md | atualizando README |
| teste.txt | teste |



PULL REQUEST / MERGE REQUEST

O merge request é um recurso do desenvolvedor de notificar aos outros membros colaboradores do time que ele concluiu uma atividade e o seu patch está pronto para ser “mergeado” a um branch superior.

É importante que haja notas de commit suficientes para informar as mudanças ou ajustes realizados ao integrador (master) para que ocorra a análise e realização do merge.

The screenshot shows the GitLab web interface for creating a new merge request. The top navigation bar includes links for 'Projetos', 'Grupos', 'Atividade', 'Marcos', 'Snippets', and a search bar. The left sidebar for the project 'curso-git' lists 'Projeto', 'Repositório', 'Issues' (0), 'Merge Requests' (0), 'CI / CD', and 'Operações'. The main content area is titled 'New Merge Request'. It has two main sections: 'Source branch' (set to 'Manupe61/curso-git') and 'Target branch' (set to 'master'). Below these sections, a message from 'Manuel Nunes Pereira' is shown, indicating they have edited the README file. A green button at the bottom says 'Compare branches and continue'.

New Merge Request

From teste into master

[Change branches](#)

Título

teste

Start the title with **WIP:** to prevent a **Work In Progress** merge request from being merged before it's ready.

Add [description templates](#) to help your contributors communicate effectively!

Description

[Write](#) [Preview](#)



detalhe com precisão o que precisa ser feito para obter a realização do merge pretendida

Markdown and quick actions are supported

[Attach a file](#)

Assignee

Manuel Nunes Pereira



Milestone

Milestone



Labels

enhancement



Approvers

Search for a user

This merge request must be approved by these users. You can override the project settings by setting your own [list of approvers](#).

Search for a group

This merge request must be approved by members of these groups. You can override the project settings by setting your own [list of approvers](#).

Approvers

There are no approvers

Approvals required

0

Tip: add a **CODEOWNERS** to automatically add approvers based on file paths and file types.

C curso-git

Manuel Nunes Pereira > curso-git > Merge Requests > !1

Open Opened 2 minutos atrás by  Manuel Nunes Pereira

Edit **Close merge requests**

teste

detalhe com precisão o que precisa ser feito para obter a realização do merge pretendida

 **Merge de teste**  para **master**
The source branch is 2 commits atrás the target branch

[Abrir na Web IDE](#) [Checkout branch](#) 

 Nenhuma aprovação necessária

 **Merge** Remove source branch [Modify commit message](#)

Você pode fazer merge manualmente usando o linha de comando

 0  0 

Pendente **Adicionar tarefa**

Responsável Alterar
Sem responsável - atribuir a si mesmo

Milestone Alterar
Nenhum

Acompanhamento de tempo 
Sem estimativa de tempo gasto

Etiquetas Alterar
enhancement

Bloquear merge request Alterar
 Desbloqueado

1 participante 

Notificações 

In case of fire



- ⌚ 1. git commit
- ⬆ 2. git push
- 🏃 3. leave building

ISSUES

ISSUES

O GitLab tem uma funcionalidade que são as issues.

Uma issue é um item de trabalho que pode representar uma demanda , uma ideia, uma tarefa ou um problema que precisa ser desenvolvido ou tratado.

A issue permite que você, sua equipe e seus colaboradores compartilhem, discutam propostas e tratem problemas antes e durante a implementação.

Uma issue pode ser uma boa prática para registrar ou documentar aspectos relevantes do desenvolvimento. Uma issue e sua evolução pode estar vinculado ao e-mail do colaborador.

The screenshot shows the GitLab web interface for a project named 'curso-git'. The left sidebar lists several project sections: Projeto (selected), Detalhes, Atividade, Painel de controle de segu..., Análise de Ciclo, Repositório, Issues (0), Merge Requests (0), CI / CD, and Operações. A red arrow points from the text above to the 'Issues' section in the sidebar. The main content area on the right displays a message: 'Você não conseguirá faz...' followed by a user profile for 'Manuel Nunes Pereira'. Below this, there's a section for 'curso-git' with a note 'escreva o que quiser' and an ID 'ID do Projeto: 94370'. There are also buttons for '0' (issues), 'Star', and 'C' (CI/CD). The right sidebar contains links for Lista, Painéis, Etiquetas, Balcão de Atendimento, and Milestones.

EXERCÍCIO

Baixe um template de HTML para edição, adicione ao seu repositório local, volte para o branch master e suba para o repositório remoto.

1. \$ git checkout master
2. \$ git status
3. \$ git add .
4. \$ git commit -m "html"
5. \$ git push gitlab master



The screenshot shows the homepage of OS TEMPLATES. At the top, there are navigation links for PREMIUM WEBSITE TEMPLATES, FREE WEBSITE TEMPLATES, FREE BASIC TEMPLATES (which is highlighted in red), FREE PSD TEMPLATES, and PAGE TEMPLATES. Below the navigation is a breadcrumb trail: You Are Here > Home > Free Basic Templates > Basic 86. The main content area features a large image of the 'Basic 86 Free HTML5 Template' with the text 'Basic 86 Free HTML5 Template'. To the right of the template image are three call-to-action buttons: 'VIEW THE TEMPLATES LICENCE' (red), 'WEBSITE TEMPLATE DEMO' (orange), and 'DOWNLOAD THIS TEMPLATE' (green). On the far right, there is an advertisement for 'elegant themes' featuring various WordPress theme previews.

<https://www.os-templates.com/free-basic-html5-templates/basic-86>

EXERCÍCIO

Crie uma issue de melhoria para ajuste no título da página.

Título: Alteração de Título

Descrição: Incluir o título “Primeiros passos no git” e na linha debaixo em h2 “Git Client & GitLab”

Dica: A nomenclatura das etiquetas padrão pode ser customizada.



New Issue

Title Add [description templates](#) to help your contributors communicate effectively!

Description

[Write](#) [Preview](#)

Incluir o título em `h1` “Primeiros passos no `git`” e na linha debaixo em `h2` “`Git Client & GitLab`”

Markdown and quick actions are supported [Attach a file](#)

This issue is confidential and should only be visible to team members with at least Reporter access.

Assignee Assign to me Due date

Milestone

Labels

[Submit issue](#)

Cancel

Labels can be applied to issues and merge requests to categorize them.
You can also set a label to make it a priority label.

[New label](#) [Generate a default set of labels](#)

VINCULAÇÃO DE ISSUES

Para relacionar um commit com uma issue, deve-se utilizar o símbolo **#** e o **número da issue** na mensagem do commit.

\$ git commit -m "bla bla bla#1"

Assim é possível garantir rastreabilidade no processo de atendimento ou resolução das issues.

Na página da issua aparece registro dos commits e branches de referência.

The screenshot shows a GitLab interface with three projects: UX, Frontend, and Platform. Each project has a list of issues:

- UX**:
 - Standardize the settings pages views #22210 (Deliverable, feature proposal, settings)
 - Do a better job of communicating when MR is blocked by a locked file. #29419 (Frontend, bug, merge requests)
 - Unable to see user to add him to repositories #29371 (Frontend, Platform, bug, reproduced on GitLab.com, user management)
 - No feedback when project limit is reached #28764 (Frontend, bug)
 - When "No one" is allowed to push, the manual
- Frontend**:
 - Import project by URL form error hides the url field #28349 (Accepting Merge Requests, bug)
 - Contribution calendar label is cut off #27839 (Accepting Merge Requests, bug, user profile, Relates to the user profile page)
 - Refine merge request widget #25424 (Deliverable, Discussion, backend, code review, coming soon, direction, merge requests, meta)
 - Create new branch and empty WIP merge request from issue page #28558 (Deliverable, Discussion, UX ready, docs-missing, feature proposal, idea-to-production, issues, merge requests)
- Platform**:
 - Improve consistency in the way we retrieve project & group in API endpoints #20728 (Deliverable, Discussion, api, technical debt)
 - Wiki Page History appears to direct to wrong link and 404s #29528 (Accepting Merge Requests, bug, wiki)
 - Removing projects is incredibly slow #27998 (Next Patch Release, availability, database, performance, reproduced on GitLab.com)
 - [API] /api/v3/projects/{projectId}/repository/commits/problem encoding non-ascii characters #28738 (api, bug, repository)

EXERCÍCIO

Realize os ajustes no arquivo index.html (linhas 13 e 14), commit com mensagem “ajustes no título”, vincule a issue e atualizar no remoto

Inclua o numero da issue criada

1. Ajustar arquivo index.html
2. \$ git commit –am “ajustes no título #1”
3. Verificar no remoto se a issue em referência faz menção ao commit



COMENTÁRIO E FECHAMENTO NAS ISSUES

Além de ser possível criar diálogos na própria issue, também é possível clicar na referência do próprio commit referenciado na issue.

Assim fica fácil se organizar quando é possível abrir o commit referenciado na issue e construir comentários diretamente neles!

Além de referenciar uma determinada issue em um commit, também é possível fechar uma issue.

Neste caso deve-se incluir o comando **closes** ou o comando **fixes**.

\$ git commit -m "bla bla bla #1"

The screenshot shows a Git commit interface. At the top, it displays a commit message: "ajustes no titulo #1". Below this, it shows the commit details: "Commit a1bf0124 authored 3 minutes ago by MANUEL NUNES PEREIRA" and a link to "Browse files" or "Options".

The commit message "ajustes no titulo #1" is shown in a box with a "parent 81f6c88f master" link.

Below the commit message, there is a file diff for "index.html". The diff shows changes made to the file:

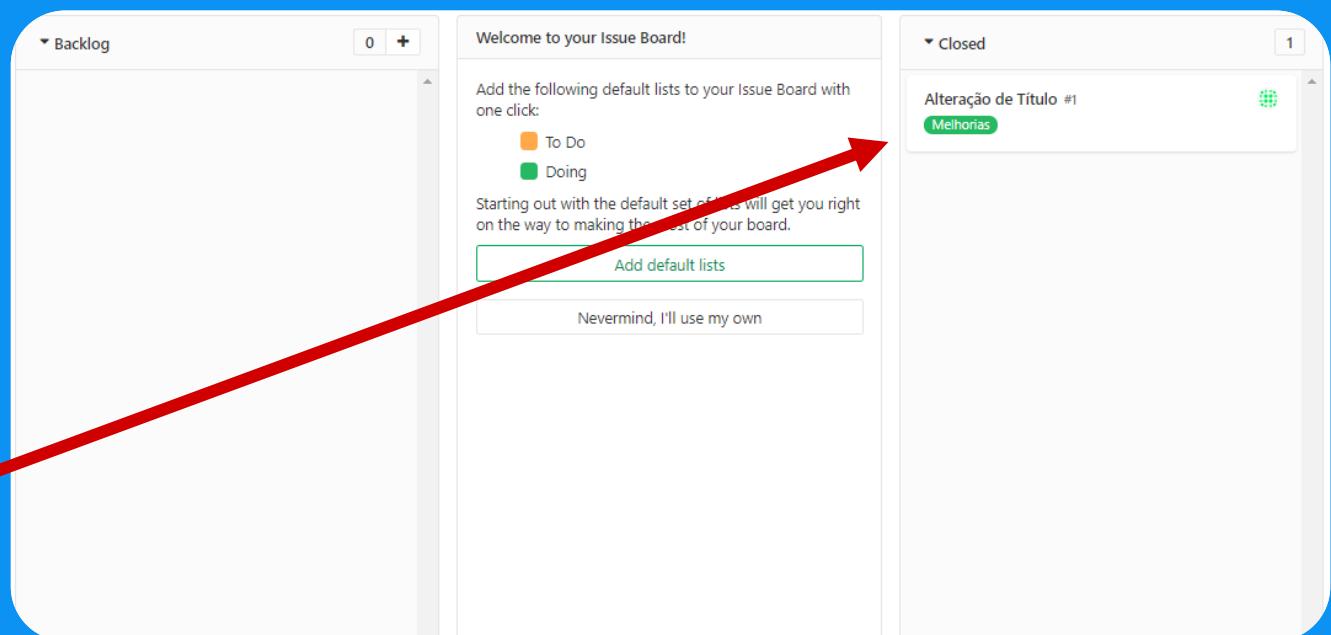
```
Showing 1 changed file ▾ with 2 additions and 2 deletions
Hide whitespace changes Inline Side-by-side
View file @ a1bf0124
index.html
@@ -10,8 +10,8 @@
 10 10 <div class="wrapper row1">
 11 11   <header id="header" class="clear">
 12 12     <div id="hgroup">
 13 -       <h1><a href="#">Basic 86</a></h1>
 14 -       <h2>Free HTML5 Website Template</h2>
 13 +       <h1><a href="#">Primeiros Passos no Git</a></h1>
 14 +       <h2>Git Client & GitLab</h2>
 15 15   </div>
 16 16   <nav>
 17 17     <ul>
```

At the bottom, there is a comment input field with a placeholder "Write a comment or drag your files here...". It includes a rich text editor toolbar and a note that "Markdown is supported". There is also a link to "Attach a file".

EXERCÍCIO

Realize novo ajuste no arquivo index.html (linhas 13) com a seguinte frase “Curso: Primeiros Passos no Git”, faça o commit com mensagem “ajustes realizados” e feche a issue.

1. Ajustar arquivo index.html
2. \$ git commit –am “ajustes realizados closes #1”
3. Verificar no remoto se a issue em referência foi encerrada.



MILESTONES

Milestones são como agregadores de issues.

Eles são um tipo especial de **etiqueta** que uma issue pode estar vinculada.

Eles podem representar uma funcionalidade, uma entrega, um conjunto de correções, um processo de teste ou uma outra atividade qualquer.

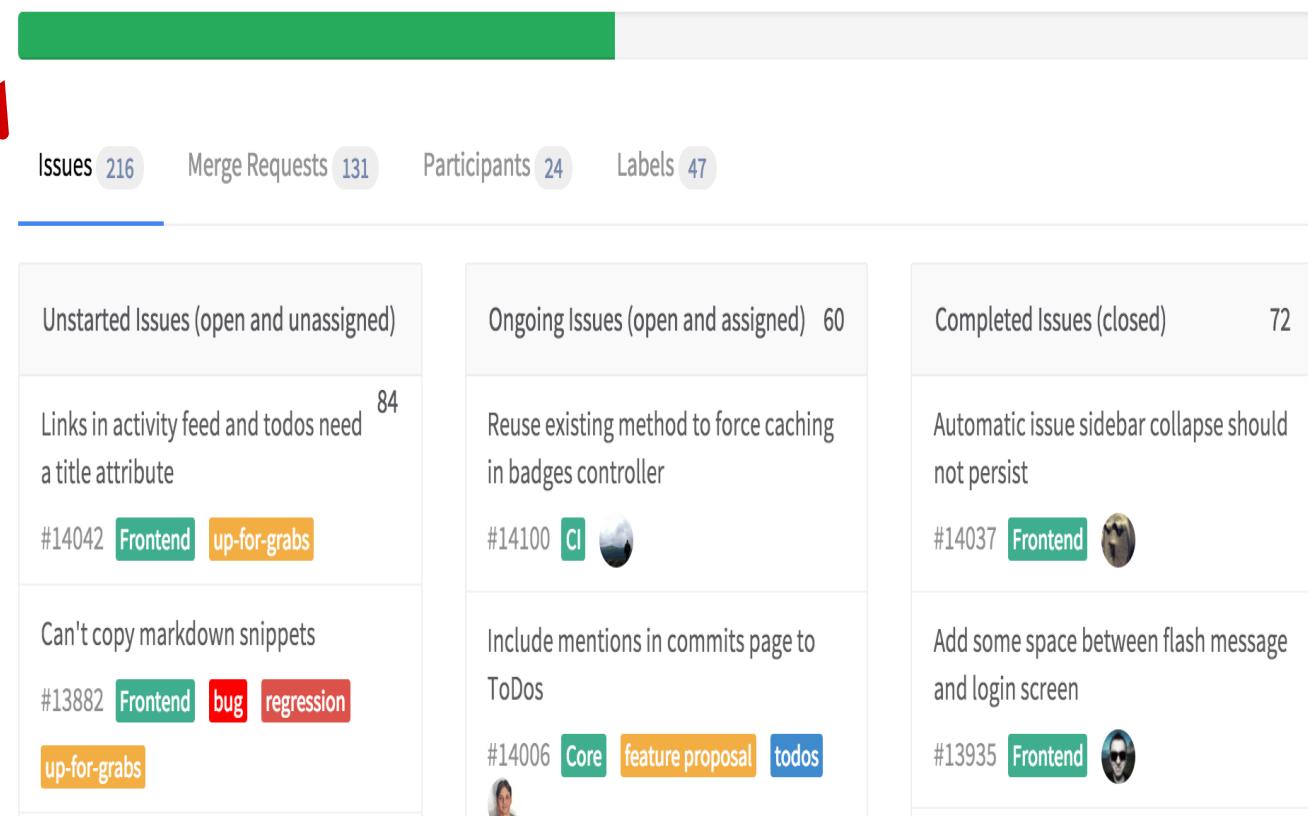
Um milestone cria uma **barra de progresso** que evolui conforme as issues vão sendo encerradas.

Progress

216 issues: 186 open and 161 closed (total weight: 107) 46% complete 14 days remaining

+ New Issue

Browse Issues



FORK

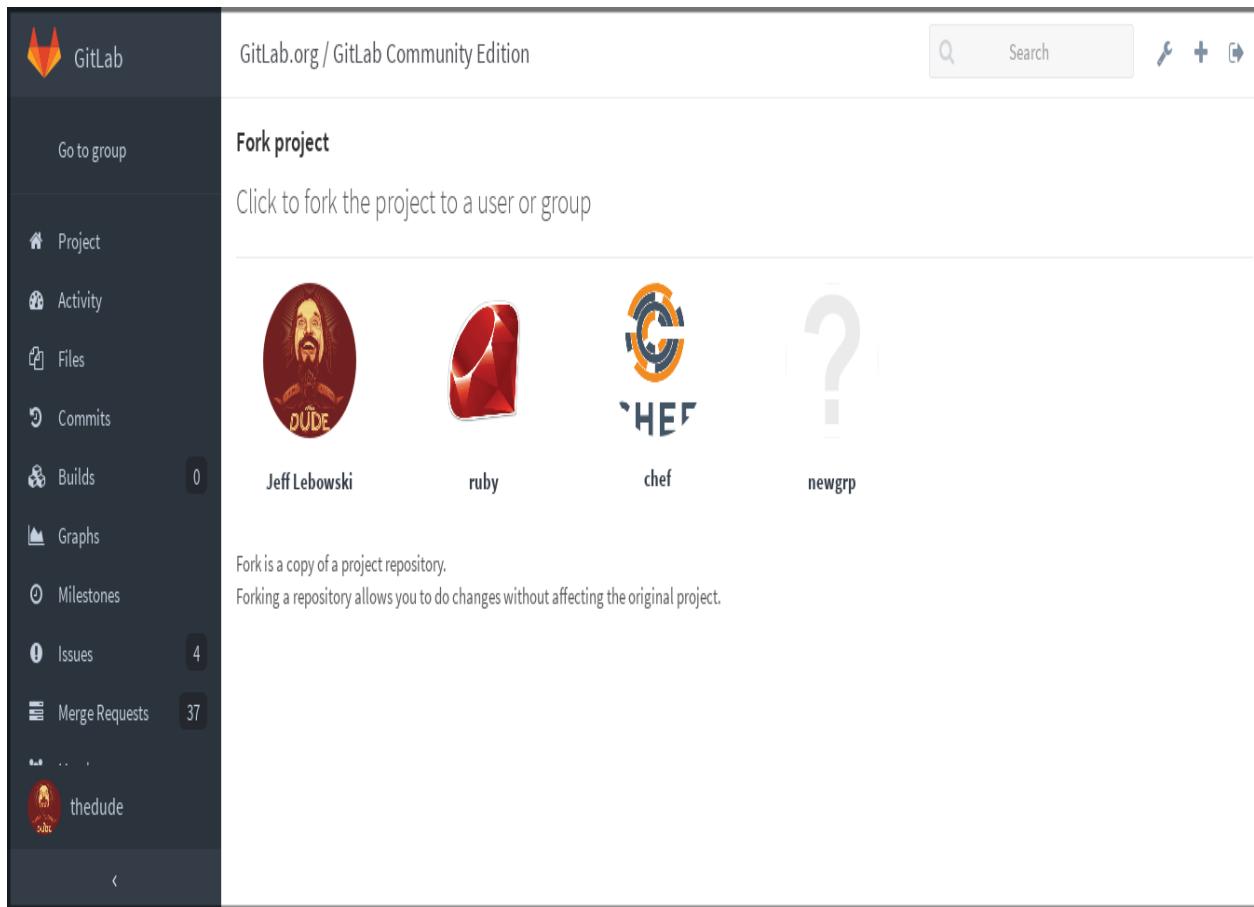
FORK

O fork é um recurso muito interessante que possibilita acessar um conjunto de códigos os quais não faz parte como desenvolvedor, mas que lhe permite copiar e experimentar alterações.

O repositório central desse projeto é comumente conhecido como “upstream”.

A cópia gerada pelo processo de “fork” será realizada no seu repositório remoto, como se fosse um novo projeto no seu grupo.

A cópia é **assíncrona** (atualizações independentes), mas é possível manter seu repositório **local** sincronizado (via add)

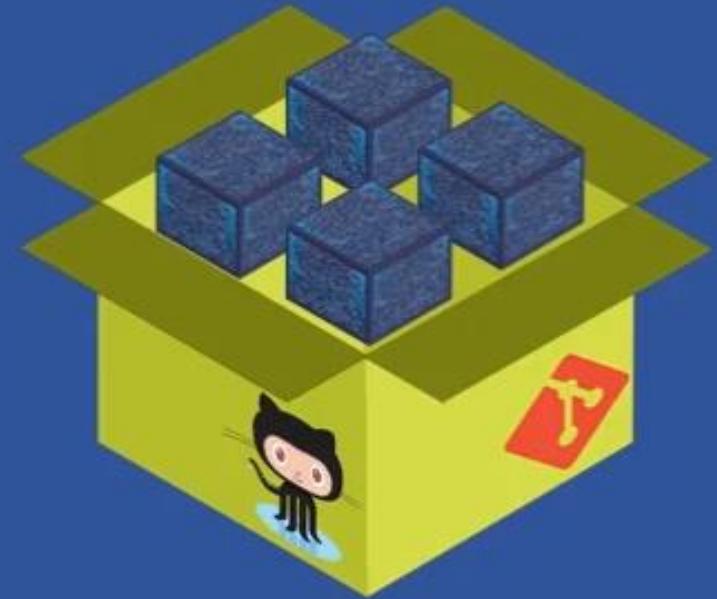


local



pull upstream

upstream



push origin



Sincronizando Repositórios

remoto



Install Git Client

git-scm.com/downloads
`apt-get install git`
toolbelt.heroku.com
sourcetreeapp.com

Identify yourself

```
git config --global user.name "name"  
git config --global user.email "email"  
  
View your Git settings  
git config --list
```

Create a local repository

```
git init
```

Version your project locally

Stage everything, a specific file or hunks

```
git add .      git add file  
git add -p    git add -p file
```

Commit all staged changes

```
git commit -m "commit message"
```

Modify the change last committed *

```
git commit --amend
```

Tracking changes

```
git status     git log  
git diff      git diff --staged  
git diff --word-diff
```

Branching & Merging

```
git branch new-branch  
git checkout branch  
git checkout -b new-branch
```

Merge specified branch to current branch

```
git merge branch
```

Modifying changes

Reset file to latest committed version

```
git checkout file
```

Remove a tracked file from staging

```
git reset HEAD file
```

Remove untracked file from staging

```
git rm -r --staged file
```

Reset all files from the last commit

```
git reset --hard
```

Stashing Changes

Save / restore working copy

```
git stash      git pop
```

Remote Repositories

Copy project from a remote repo

```
git clone remote-name repo-url
```

Update local repo from remote

```
git pull remote-name branch
```

Update remote with local commits

```
git push remote-name branch
```

Stash

changes briefly hidden away

Working Copy

source code & config files

Staging

changes awaiting commit

Local Repo

.git folder managing changes on your laptop

Remote Repo

Repos accessible via the Internet eg. Github, Heroku

Git Help

Git visual cheatsheet
bit.ly/git-visual-cheatsheet

Github Help
help.github.com

Git & Github tutorial
jr0cket.co.uk/git-workshop