Conteúdo de apoio

O uso do Git pode ser feito de várias formas. As IDEs e editores de código mais modernos, por exemplo, já trazem plugins e soluções nativas para utilização desse sistema de controle de versão. Nesse vídeo, no entanto, para apresentar os principais comandos independentemente do ambiente de desenvolvimento que você faz uso, optamos pelo terminal do sistema operacional.

Assim, após instalar o Git em sua máquina, os dois primeiros comandos a executar no prompt são:

```
git config --global user.name "Nome do usuario"
git config --global user.email e-mail@exemplo.com
```

O Git pede uma identificação, para que possa sinalizar o autor das mudanças realizadas no repositório. Logo após, no diretório em que você pretende criar ou já criou o projeto, execute:

```
git init
```

Isso é o que precisamos para inicializar o repositório e criar a pasta oculta .git. Agora, nosso diretório também é o que chamamos na linguagem Git de working area, o diretório de trabalho.

Qualquer arquivo criado/modificado neste diretório será monitorado pelo Git. Para que possamos adicionar esse arquivo em nosso repositório, primeiro precisamos adicioná-lo ao stage, o que é feito com o comando:

```
git add NomeDoArquivo.extensão
```

Dessa forma o arquivo estará referenciado nesse espaço intermediário. A partir disso, para confirmar as mudanças e adicionar o novo arquivo ao repositório, precisamos executar:

```
git commit -m “Mensagem descritiva das mudanças realizadas.”
```

É válido destacar a importância da mensagem especificado junto com o commit. É recomendado que a partir dela o programador saiba tudo o que foi feito naquela versão.

Outro conceito importante do Git é o conceito de branch. É aconselhado criar um branch sempre que você desejar testar alguma nova API/Biblioteca/Framework ou começar a criar uma nova funcionalidade no projeto. Quando inicializamos o repositório, automaticamente ele cria o branch master. Esse é o branch principal do projeto. Quando estamos trabalhando em equipe, principalmente, o ideal é que esse branch receba apenas versões do código com as funcionalidades já testadas e avaliadas.

Para começar a criar uma nova funcionalidade, como já mencionado, o ideal é criar um novo branch. Ao fazer isso, lembre-se de dar um nome autoexplicativo. Para criar o branch, execute:

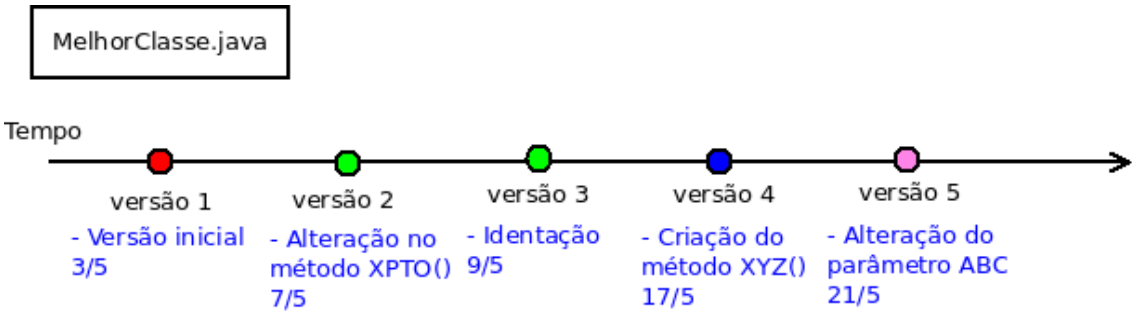
```
git branch nomeDoBranch
```

Ao fazer isso, é como se criássemos uma cópia de todos os arquivos do nosso projeto. A diferença é que essa cópia é gerenciada pelo Git. Portanto, você não verá arquivos duplicados. A partir disso, para acessar o novo branch, o nosso novo ramo, execute no terminal:

```
git checkout nomeDoBranch
```

Logo após, todas as mudanças realizadas a partir de agora estarão acessíveis apenas nesse branch. Portanto, se você criar um ou mais arquivos, ao voltar para o branch master (com o comando git checkout master) esses arquivos não serão encontrados, pois nesse ramo esses arquivos não existem.

Conhecer o histórico de um determinado arquivo é muito importante, pois fornece preciosas informações para o entendimento do arquivo em si. Contudo, o maior benefício do controle de versão é a colaboração. A **Figura 3** ilustra as alterações em um arquivo de código-fonte feitas por quatro programadores, cada um representado por uma cor diferente. Com esse histórico é possível saber quem foi o responsável pela criação de cada versão, quais mudanças ocorreram em cada versão, o motivo das alterações e, o recurso mais importante e poderoso no controle de versão, unir todas as contribuições de todos os indivíduos quando trabalharam em versões paralelas do arquivo em um determinado arquivo, contendo todas as alterações por meio do processo chamado `merge`. Graças a ele é possível ter diferentes pessoas alterando o mesmo arquivo ao mesmo tempo e, quando essas pessoas terminarem suas tarefas, une suas alterações em um arquivo que contemple as tarefas de ambos. Dessa forma, o trabalho se desenvolve muito mais rápido, pois não é mais preciso esperar que alguém que esteja trabalhando em um arquivo termine para que outra pessoa possa fazer suas alterações.

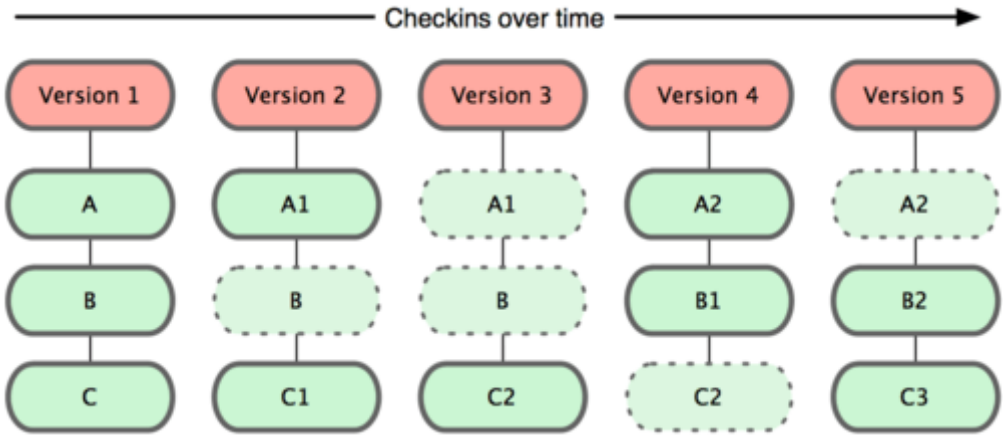


A grande diferença entre os sistemas centralizados e os sistemas distribuídos, como o Git, é em relação ao commit. Em sistemas centralizados, o commit simplesmente envia todas as alterações nos artefatos para o repositório central. Nos sistemas descentralizados o commit apenas cria uma imagem das alterações em um repositório local e, só após o comando o `push` (que será abordado no decorrer do artigo) é que as alterações nos artefatos são enviadas para o repositório remoto. Essa forma de trabalho descentralizada torna o Git muito mais rápido em relação as ferramentas centralizadas, além de promover o uso de branches locais para experimentos e commits frequentes, práticas comuns e desenvolvidas na disciplina de métodos ágeis.

Snapshots no Git

A maioria dos outros controladores de versão gerenciam diferenças entre arquivos, associando a cada um deles uma lista de mudanças. Mas no Git as coisas são bem distintas, pois ao invés de uma lista de mudanças associada a cada arquivo, ele usa snapshots para manter o histórico, ou seja, a cada commit o Git copia todos os arquivos do projeto e muda a referência dos mesmos marcando-os como sendo a versão atual. No caso de arquivos que não foram alterados, o Git apenas mantém um link para o arquivo sem copiá-lo novamente. Esse processo de funcionamento é ilustrado na Figura 4.

Nesse exemplo é possível observar o estado inicial onde todos os arquivos foram criados, Version 1. Em seguida, após alterações nos arquivos A e B, uma nova versão é criada, a Version 2, com os arquivos A1 e C1 copiados literalmente para esse snapshot e uma referência ao arquivo B, que não sofreu nenhuma alteração. Já na Version 3, quando apenas um arquivo é alterado (arquivo C2), apenas ele é copiado para o novo snapshot e os demais são links para os arquivos anteriores não alterados. As demais versões dessa linha do tempo seguem a mesma lógica.



Essa forma de lidar com as versões como se fossem snapshots traz muitas vantagens, especialmente em relação a criação de branches. Como no Git um branch é um ponteiro para algum commit, poderíamos criar facilmente um branch de qualquer uma das versões da **Figura 4**. O branch seria criado pelo Git simplesmente adicionando um ponteiro para algumas versões e, a partir de então, alterações e commits nesse novo branch seguiriam o mesmo processo de versões e snapshots a partir do ponto de criação desse novo branch em diante.

Trabalho local com Git

Outra característica única do Git é sua forma de trabalho local. Diferente de outros controladores de versão, a maior parte das operações do Git pode ser executada localmente, já que o Git mantém um histórico do projeto localmente. Isso faz do Git uma ferramenta extremamente rápida, que não sofre com a latência da rede para a execução da grande maioria de suas operações. As principais vantagens dessa característica são a rapidez nas operações (imagine ter que comparar diferentes versões de um arquivo de um ou dois meses atrás usando recursos locais versus recursos remotos) e a possibilidade de se trabalhar e ter a disposição quase todas as operações mesmo quando não se está conectado na rede.

Estados do Git

Um dos conceitos mais importantes do Git são os estados que podem ser aplicados em cada arquivo. São três estados fundamentais:

- **`Commited`** – arquivos armazenados na base local.
- **`Modified`** – arquivos que sofreram mudanças, mas não foram enviados para a base local (commit).
- **`Staged`** – arquivos modificados marcados para que façam parte do próximo commit.

A **Figura 5** ilustra a transição de estados de arquivos versionados pelo Git em seu fluxo de trabalho. Além dos estados, pode-se observar seções que estão relacionadas ao estado de cada arquivo. O Git os organiza no diretório de trabalho (working directory), área de preparação (staging area, também chamada de index) e repositório (git directory).

O repositório é o local onde são armazenados os metadados e o banco de objetos do projeto. Essa é a parte copiada quando um projeto é clonado.

O diretório de trabalho é um checkout de alguma versão do projeto. Os arquivos desse diretório são obtidos a partir do banco de dados comprimido do Git no repositório e disponibilizados no disco, possibilitando sua utilização e edição.

A área de preparação é, na verdade, um arquivo chamado de index dentro do repositório que especifica o conteúdo do próximo commit.

Em linhas gerais, o fluxo de trabalho com o Git pode ser descrito da seguinte forma:

1. Faz-se o checkout de um projeto do repositório, torna-se o diretório de trabalho.
2. Todas as alterações (correções, desenvolvimento, etc.) são realizadas no diretório de trabalho. As alterações vão para a área de preparação.
3. Quando o commit é feito, todos os arquivos da área de preparação são armazenados no repositório.

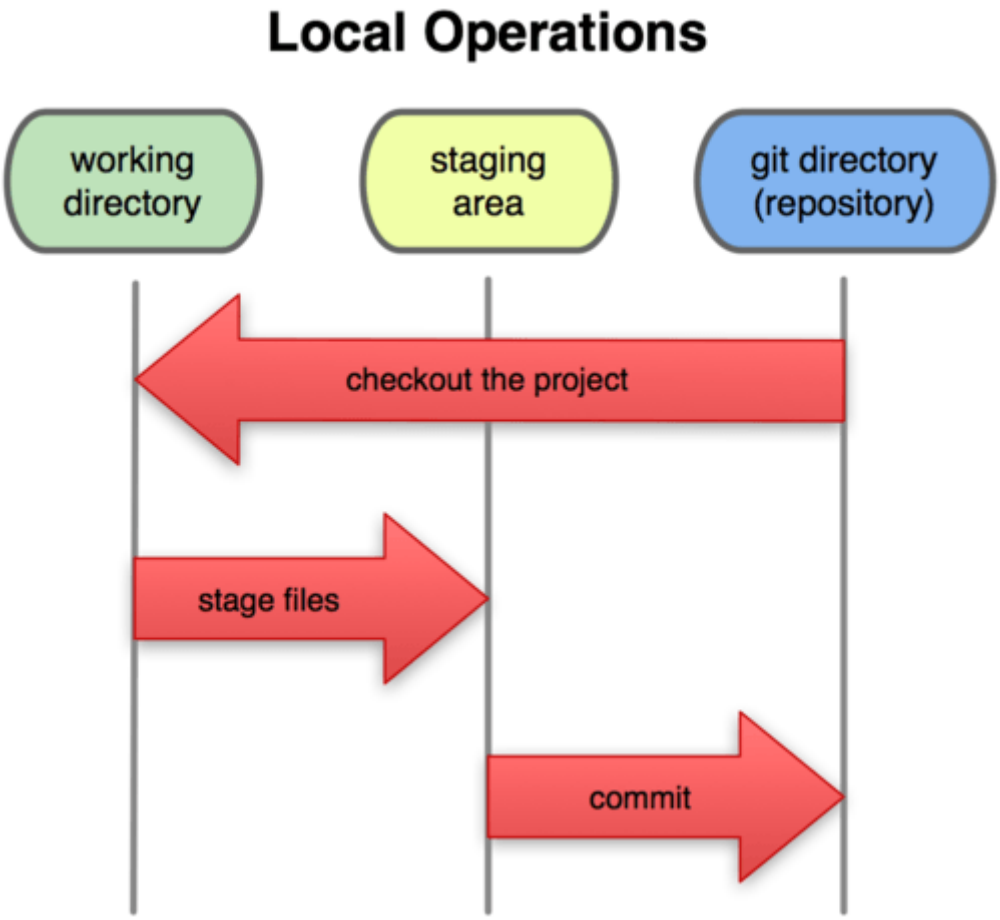


Figura 5. Transição de estados de arquivos versionados pelo Git

O entendimento dos estados, da forma de trabalho local e dos snapshots são fundamentais para se utilizar o Git de forma fácil e correta, pois essas três características são a base que define o versionamento de controle distribuído e o próprio Git.

Os próximos tópicos se concentram no uso prático do Git conforme o fluxo de trabalho apresentado na **Figura 5**, demonstrando os principais comandos para lidar com versionamento. Veja na seção **Links** o link para fazer o download, instalar e configurar o Git. Partiremos da ideia que o mesmo está pronto em sua máquina.

Como configurar Git

O comando `git config` permite configurar diversas características do Git. A maioria delas tem um valor default que se encaixa na grande maioria dos casos. Entretanto, ao menos o usuário deve ser configurado para que o Git associe, por exemplo, seu nome e e-mail em seus commits. Para fazer isso, basta executar os comandos

```
git config --global user.name <name>

git config --global user.email <email>.
```

Criando um repositório no Git

O primeiro passo para se trabalhar com controle de versão usando o Git é criar um repositório. Isso pode ser feito com o comando `git init` dentro do diretório que se deseja transformar em repositório, conforme exemplo a seguir:

```
[user@localhost repository]$ git init

Initialized empty Git repository in /home/user/repository/.git/
```

Esse comando inicializa um diretório em um repositório vazio. O diretório oculto criado `.git` contém todas as configurações e dados do repositório em questão, tais como links para repositórios remotos, banco de dados com os arquivos versionados, etc. Uma vez criado o repositório, todo o fluxo de trabalho apresentado anteriormente se torna possível.

Rastreando arquivos no Git

O repositório recém-criado ainda não contém nenhum arquivo para ser versionado (rastreado). Com o comando `git add` pode-se adicionar todos os arquivos do repositório no index (área de preparação) para o próximo commit e, quando finalmente o referido comando for executado, os arquivos passarão a ser rastreados.

O index é um snapshot das alterações do diretório de trabalho corrente e dita o conteúdo do próximo commit, por isso, após qualquer alteração é necessário adicioná-la no index, caso contrário a mesma não será gravada no repositório após commit. Esse comando adiciona todas as alterações nos diretórios abaixo do repositório, mas é possível especificar a adição de arquivos únicos informando o nome e o caminho do arquivo ao invés do ponto, como por exemplo,

```
git add/home/user/repository/MinhaClasse.java
```

Como exemplo, ao criar o arquivo MinhaClasse.java e executar o comando `git status` , que informa o atual estado do diretório de trabalho, obtém-se o resultado da **Listagem 1**.

Listagem 1. Estado atual do arquivo MinhaClasse.java

```
[user@localhost repository]$ git status

On branch master

Initial commit

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

          MinhaClasse.java

nothing added to commit but untracked files present (use "git add" to
track)
```

Isso significa que houve alterações no diretório de trabalho, mas essas mudanças não foram adicionadas no index e, portanto, nada será adicionado ao repositório em caso de

commit. Para fazer o commit desse arquivo é necessário, antes de tudo, adicioná-lo por meio do comando `git add`. Após adicionar o arquivo MinhaClasse.java o status agora é o mesmo apresentado na **Listagem 2**.

Listagem 2. Estado atual do arquivo MinhaClasse.java após o comando add

```
[user@localhost repository]$ git add MinhaClasse.java
[user@localhost repository]$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

     new file:   MinhaClasse.java
```

Agora há alterações no index prontas para serem adicionadas ao repositório. `MinhaClasse.java` se tornou um arquivo pronto para ser adicionado ao repositório e ter seu versionamento iniciado.

O próximo tópico detalhará o comando de commit, mas antes note a dica do Git para remover arquivos do index: `git rm --cached <file>`. Caso o arquivo seja removido do index, ele não será adicionado ao repositório.

Git Commit

O comando `git commit` armazena o atual conteúdo do index em um novo commit no repositório juntamente com uma mensagem definida pelo usuário descrevendo as alterações realizadas. Em linhas gerais, a partir do momento em que o commit é realizado, o arquivo passa a ter suas alterações rastreadas, sendo possível visualizar as mesmas ao longo do tempo, comparar diferentes versões e até voltar para versões anteriores.

Para fazer o commit do arquivo MinhaClasse.java descrevendo as alterações com uma mensagem indicando se tratar do primeiro commit, é preciso executar o `comando git commit -m "primeiro commit"`. Na **Listagem 3** é apresentado o resultado.

Listagem 3. Resultado do comando commit

```
[user@localhost repository]$ git commit -m "primeiro commit"
[master (root-commit) 9026f18] primeiro commit
 1 file changed, 3 insertions(+)
 create mode 100644 MinhaClasse.java
```

Nesse caso o commit foi executado com sucesso e um arquivo foi alterado (MinhaClasse.java que foi adicionado ao repositório), já que houve três inserções (este arquivo contém três linhas e, cada linha inserida entra nessa contagem). A partir de agora esse arquivo passa a ser rastreável e está no repositório local do Git, onde é possível consultar todo o seu histórico.

Histórico: Git commit history

Um dos usos mais importantes para controladores de versão é possibilitar o controle das versões, mostrando o histórico de evolução de um arquivo ou de um conjunto de arquivos por commits. O comando `git log <file>` possibilita visualizar o histórico dos commits por arquivo e o comando `git log` mostra todos eles. Após alteração e commit da classe MinhaClasse.java, na qual foi adicionada uma variável, o comando `git log MinhaClasse.java` mostra o seguinte histórico, apresentado a **Listagem 4**.

Listagem 4. Histórico após o comando git

```
[user@localhost repository]$ git log MinhaClasse.java
commit 10a1f9cb0a490a012dcb625c0571a1307aa41624
Author: Gabriel Amorim <gabriel@email.com>
Date:   Tue May 13 17:42:52 2017 -0300

    //adição de constante na MinhaClasse.java

commit 9026f184ed12772e8480ca59056eb3c29223a808
Author: Gabriel Amorim <gabriel@email.com>
Date:   Tue May 13 17:14:02 2017 -0300

    primeiro commit
```

Note que cada commit é seguido por um código hash, que é gerado considerando o conteúdo de todo o commit. Dessa forma, cada um tem seu próprio hash e, consequentemente, é impossível perder qualquer alteração ou ter arquivos corrompidos sem que o Git seja capaz de detectar. Voltando ao histórico, veja que cada commit é apresentado mostrando o autor do mesmo, a data e um comentário descrevendo o seu motivo. Assim, é muito fácil consultar o histórico e a evolução de um arquivo ou de toda a base de código.

Para acessar o autor do log:

```
git log --author "Heloisa"
```

Procurar log por palavra-chave na mensagem de commit:

```
git log --grep "aplicação-juros"
```

Gitignore

Arquivo criado para excluir os aquivos que não quero incluir no commit:

```
notepad .gitignore

*.class

cat .gitignore
```

Comandos do Git ao se trabalhar em equipe

Os comandos apresentados nos tópicos anteriores são bastante úteis ao se trabalhar localmente e fazem parte dos comandos core do Git. No entanto, quando se trabalha em

uma equipe que mantém um repositório remoto para versionamento, são necessários novos comandos para atuar nesse cenário. Os próximos tópicos apresentam os comandos mais úteis para se trabalhar em equipe no Git.

Git Clone

O cenário mais comum é uma equipe ter um repositório remoto ao qual todos têm acesso para enviar seus commits. Nesse caso, antes de começar a desenvolver, é necessário baixar o repositório remoto e isso é feito com o comando `git clone <caminho_para_o_repositório>`. Este faz uma cópia do trabalho local de um repositório. O exemplo da **Listagem 5** faz o clone de um repositório do GitHub – serviço que mantém diversos repositórios online – com o comando `git clone https://github.com/gabrielamorim/Java_repository.git`.

Listagem 5. Clone de um repositório do GitHub

```
[user@localhost repository]$ git clone
https://github.com/gabrielamorim/Java_repository.git
Cloning into 'Java_repository'...
remote: Counting objects: 126, done.
remote: Total 126 (delta 0), reused 0 (delta 0), pack-reused 126
Receiving objects: 100% (126/126), 28.53 KiB | 0 bytes/s, done.
Resolving deltas: 100% (35/35), done.
Checking connectivity... done.
```

Após realizar o clone do repositório, pode-se realizar quaisquer alterações nos códigos baixados. Quando as alterações forem finalizadas, o comando `git status` mostrará as mesmas no diretório de trabalho, que devem ser adicionadas ao index com o comando `git add` e, finalmente deve-se fazer o commit com o comando `git commit -m "mensagem do commit"` para armazenar as alterações no repositório local e poder rastrear localmente o trabalho realizado.

Enviando e recebendo alterações no repositório remoto

As alterações realizadas e que forem adicionadas no repositório por meio do commit estarão apenas no repositório local. Como o trabalho é em equipe, as alterações locais devem ser enviadas para o repositório remoto para que todos os membros da equipe também tenham acesso e possam atualizar seus repositórios locais com as últimas alterações de outros membros da equipe.

Para isso utiliza-se o comando git push, que envia todos os commits do repositório local para o repositório remoto. O exemplo da **Listagem 6** mostra o resultado da execução do git push. Para enviar os commits para o repositório remoto é necessário fornecer as credencias de acesso.

```
git remote add index.html
https://github.com/HeloisaFelizardo/teste.git

git push index.html
```

Listagem 6. Resultado da execução do git push

```
[gamorim@localhost Java_repository]$ git push
Username for 'https://github.com': username
Password for 'https://novais.amorim@gmail.com@github.com':
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 491 bytes | 0 bytes/s, done.
Total 5 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
To https://github.com/gabrielamorim/Java_repository.git
   bf03c8d..35ffd16  master -> master
```

Da mesma forma que alterações são enviadas para o repositório remoto, em algum momento será necessário fazer o update do repositório local para baixar as últimas alterações disponíveis no repositório remoto. Isso é feito com o comando git pull, que simplesmente baixa todo o conteúdo atualizado do repositório remoto para o repositório local. Ao tentar executar o `git pull` no repositório `https://github.com/gabrielamorim/Java_repository.git`, obtém-se o resultado a seguir:

```
[user@localhost Java_repository]$ git pull
Already up-to-date.
```

Esse resultado significa que não há alterações no repositório remoto e que o local já está atualizado. Entretanto, caso houvessem alterações, elas seriam baixadas e o Git mostraria os arquivos alterados.

Git Branches e tags

Branches são ramificações, como uma cópia de um diretório de trabalho, criados para se desenvolver funcionalidades de forma isoladas uns das outros. Todo repositório Git se inicia com um branch padrão (master). O principal fluxo de trabalho consiste em criar novos branches para desenvolver uma funcionalidade e, quando finalizar o desenvolvimento, mesclar (merge) o branch criado com o principal.

Um branch pode ser criado com o comando

```
git checkout -b <nome_do_novo_branch>.
```

O exemplo da **Listagem 7** mostra a criação de um novo branch.

Listagem 7. Criação de um novo branch

```
[user@localhost repository]$ git branch
* master
[user@localhost repository]$ git checkout -b funcionalidade_xyz
Switched to a new branch 'funcionalidade_xyz'
[user@localhost repository]$ git branch
* funcionalidade_xyz
  master
```

Primeiro foi executado o comando `git branch` para identificar o branch atual. Nesse caso, o único existente era o master, marcado com um asterisco para indicar o branch corrente. O comando `git checkout -b funcionalidade_xyz` criou um novo branch chamado funcionalidade_xyz. O Git então mudou para o novo branch e, uma nova

execução do comando `git branch` mostrou a existência de dois branches, sendo o atual (aquele em que as alterações serão aplicadas) como sendo a funcionalidade_xyz.

Ao fazer uma alteração em algum arquivo desse novo branch, como o MinhaClasse.java, todas as mudanças serão adicionadas a esse branch.

Para voltar ao branch master utilize o comando `git checkout master`. Ao verificar o arquivo MinhaClasse.java do branch master, pode-se ver que não há nenhuma alteração, pois as mudanças foram feitas no funcionalidade_xyz. Agora é preciso fazer o merge dos dois branches, que é feito com o comando `git merge <nome_do_branch>`. Este mescla as alterações realizadas no branch onde ocorreram as alterações com o branch corrente que, no caso, é o principal. Ao avaliar o arquivo MinhaClasse.java agora, é possível ver que as alterações do `funcionalidade_xyz` agora estão no branch master. O Git sabe exatamente como fazer o merge caso não haja conflitos, no entanto, quando houver divergências que ele não saiba como resolver, o mesmo exibirá as diferenças entre os branches e solicitar o merge manual, como mostra a **Listagem 8.**

Listagem 8. Diferenças entre os branches

```
[user@localhost repository]$ git merge funcionalidade_xyz
Updating 10a1f9c..a98668a
Fast-forward
 MinhaClasse.java | 1 +
 1 file changed, 1 insertion(+)
 create mode 160000 Java_repository
```

Os branches auxiliam no desenvolvimento paralelo de funcionalidades, não impactando no código do sistema em produção ou a versão mais próxima de produção, que geralmente é o que está no branch master. O merge de um branch paralelo com o principal geralmente é devido a inclusão de uma funcionalidade ou correção de bug, então convém criar um ponto de release, ou seja, uma marcação indicando a versão do software a partir daquele commit, chamada de tag, e sua criação é feita com o comando `git tag <nome_da_tag> <identificador_do_commit>`, como no exemplo a seguir:

```
[user@localhost repository]$ git tag 1.0.0
a98668ab880ba9059cf587de851b989f09c71fbb
```

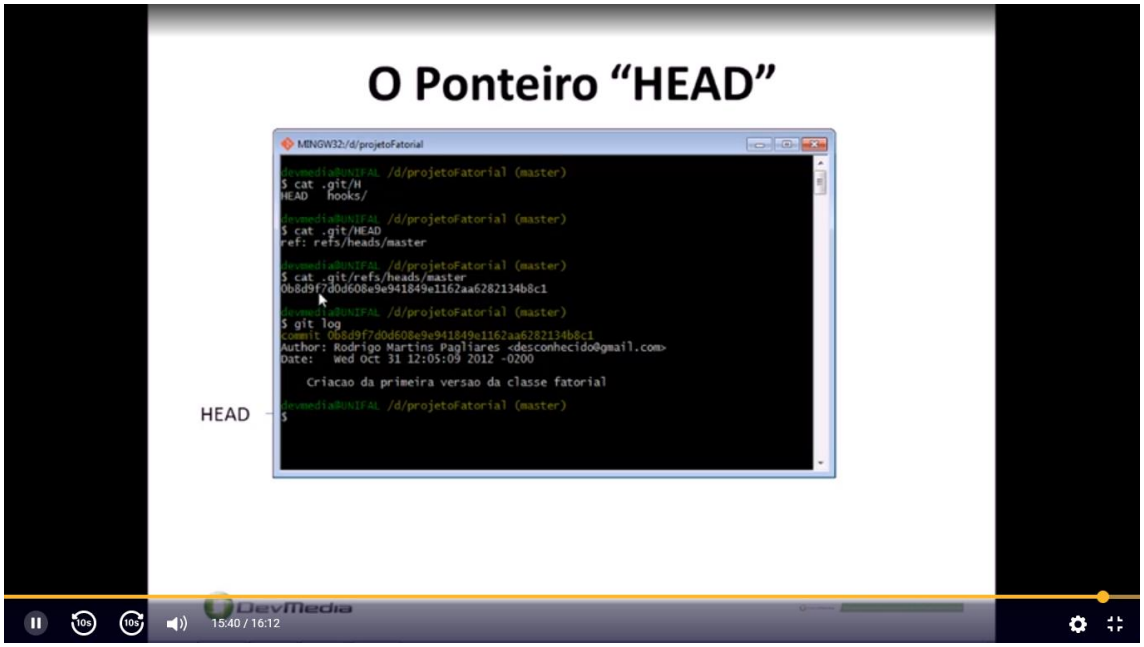
O nome da tag é o identificador da mesma para consultas, que geralmente é o número de versão seguindo a convenção da equipe. O identificador do commit é o hash do commit que originou a criação dessa tag, que pode ser obtido por meio do histórico.

O fluxo de trabalho das ferramentas controladoras de versão tendem a ser mal interpretadas, especialmente quando acontece a migração de uma ferramenta para outra, fazendo com que os desenvolvedores continuem a utilizar a nova ferramenta com os mesmos conceitos da antiga. Isso acontece especialmente com o Git, que tem um fluxo de trabalho um pouco diferente das ferramentas mais utilizadas até então, fazendo o Git ser taxado até mesmo de muito complexo.

Os recursos apresentados nesse artigo são muito úteis para facilitar e melhorar o controle de versão e possibilitam o uso básico e corriqueiro da ferramenta de forma

correta. Entender o Git corretamente é o primeiro passo de um longo caminho para o aperfeiçoamento do ambiente de desenvolvimento.

Comando Cat



Exemplo 1:

- Inicializar o Git;
- Configurar usuario
- Configurar e-mail;
- Criar branch 'ProjetoLoja';
- Definir o novo branch como padrão.

```
git init
git config --global user.name "Fulano da silva"
git config --global user.email e-mail@exemplo.com
git branch ProjetoLoja
git checkout ProjetoLoja
```

Exemplo 2:

- Inicializa o Git;
- Verifica se houve alguma alteração na working area;
- Adiciona o arquivo file.js ao stage;
- Commita as alterações no repositório local.

```
git init
git status
git add file.js
git commit -m "criação do file.js"
```

Atualizando informações

Complete os campos abaixo de forma que o primeiro comando acesse um branch já existente chamado 'novidades' e o segundo comando adicione a ele todas as alterações do branch 'master':

```
git checkout novidades

git merge master
```

Clonando repositório

Seguindo a ordem dos itens da lista, complete as lacunas com os comandos necessários para cumprir todos os objetivos:

- Clonar o repositório https://github.com/meusuario/meu_site.git;
- Checar por alterações na área de trabalho;
- Adicionar o item 'footer.php' ao stage;
- Publicar as alterações do stage no repositório local com a mensagem "footer".

```
git clone https://github.com/meusuario/meu_site.git
git status
git add footer.php
git commit -m "footer"
```

Ações do desenvolvedor

Após verificar se havia alterações no projeto, o desenvolvedor adicionou todos os arquivos do diretório atual ao stage e em seguida publicou no repositório local.

```
git status
git add .
git commit -m "publicado"
```

Cumprindo tarefas

Um desenvolvedor chegou na equipe para trabalhar em um novo projeto. Para isso, foi solicitado que ele fizesse algumas tarefas. Complete os campos abaixo com os comandos que o desenvolvedor utilizou para cumprir essas tarefas:

- Criar sua área de trabalho (working area) / inicializar o git
- Clonar o novo projeto https://github.com/meusuario/novo_projeto.git
- Criar um branch chamado 'ProjetoNovoDev'

```
git init

git clone https://github.com/meusuario/novo_projeto.git

git branch ProjetoNovoDev
```

Tarefas em ordem

- Clonar o repositório https://github.com/meusuario/meu_projeto.git;
- Criar um ramo chamado 'MeuProjetoDev';
- Selecionar o novo ramo criado;

- Verificar se houve alteração na working area;
- Adicionar todas as alterações encontradas no diretório atual ao stage;
- Publicar as alterações no repositório local com a mensagem 'release'.

```
git clone https://github.com/meusuario/meu_projeto.git

git branch MeuProjetoDev

git checkout MeuProjetoDev

git status

git add .

git commit -m "release"
```

Observe os comandos

Observe os comandos executados abaixo e selecione a alternativa que lista corretamente o que está sendo feito na ordem em que aparecem:

```
git checkout -b NovoBranch
git status
git add profile.php
git commit -m "novo perfil"
```

- Cria um novo branch chamado 'NovoBranch' e o define como padrão

- Checa se houve alguma alteração na working area do projeto

- Adiciona o arquivo 'profile.php' ao stage

- Publica as alterações do stage no repositório local

Alcançando objetivos

Após publicar uma grande alteração no projeto, foi solicitado ao desenvolvedor marcar essa alteração como a versão '3.0.0'. Porém, para isso ele precisava saber o hash do último commit do arquivo 'home.php'.

OBS.: considere o valor a98668ab880ba9059cf587de851b989f09c71fbb como a hash retornada.

```
git log home.php
git tag 3.0.0 a98668ab880ba9059cf587de851b989f09c71fbb
```

Instruções

- Baixa todas as novidades existentes no repositório externo configurado previamente;
- Verifica se houve alterações na working area do projeto;
- Adiciona todas as alterações do diretório atual ao stage;
- Publica as alterações no repositório local;
- Envia todas as alterações commitadas para o repositório externo.

```
git pull
git status
```

```
git add .
git commit -m "nova home"
git push
```

Ordem de execução

Em qual das alternativas abaixo os comandos utilizados para baixar dados do repositório externo configurado previamente,
adicionar todas as alterações ao stage, commitar para o servidor local e enviar para o repositório externo são executados, respectivamente?

```
git pull
git add .
git commit -m "commit"
git push
```

Clonando repositório para o repositório local e enviando para o repositório remoto

Dentro da pasta do projeto:

```
git init
git clone https://github.com/RexShack/rsg_stable.git
git status
git add .
git commit -m "comitando"
git remote add origin https://github.com/HeloisaFelizardo/teste-lua.git
git push -u origin main
git remote set-url origin https://github.com/HeloisaFelizardo/teste-lua.git
git remote -v
git push -u origin main
```

Instruções da lista

- Selecione o branch 'master';
- Publique as alterações do branch 'LojaDev' no branch principal;
- Defina uma marcação de valor '4.0.0' para o commit de hash a98268ab380ga9059cf587de851b989f09c71fbb.

```
git checkout máster
git merge LojaDev
git tag 4.0.0 a98268ab380ga9059cf587de851b989f09c71fbb
```

Removendo arquivos

Durante o processo de publicação das alterações no repositório, foi solicitado ao desenvolvedor que não publicasse as modificações feitas no arquivo 'relatorio.php'.
Qual comando o desenvolvedor utilizou para remover o arquivo do stage?

```
git rm --cached relatorio.php
```

Branch
Utilizando apenas um comando Git crie um branch chamado 'desenvolvimento' e defina ele como padrão:

```
git checkout -b desenvolvimento
```

Configurando dados
Qual das alternativas abaixo configura corretamente um nome e um e-mail de usuário no Git?

```
git config --global user.name "Nome do usuario"
git config --global user.email e-mail@exemplo.com
```

Unificando ramos
Um desenvolvedor precisava publicar alterações commitadas previamente em um branch chamado 'dev' no ramo principal. Para isso, ele precisava acessar o branch 'master' já existente e executar mais um comando.

```
git checkout master
git merge dev
```

Comandos corretos
- Baixar os dados do repositório remoto configurado previamente;
- Criar um novo branch;
- Definir o branch criado como padrão.

```
git pull
git branch novoPerfil
git checkout novoPerfil
```

Como visualizar as alterações feitas entre duas versões de um determinado arquivo
Git diff

- Verifica se existem diferenças entre os branchs master e development

- Seleciona o branch master

- Mescla o conteúdo do branch development com o branch master

```
git diff master..development
git checkout master
git merge development
```

Baixando dados

Um desenvolvedor precisa configurar um repositório de nome 'repo' e baixar as todas as novidades dele para sua working area.
Qual das alternativas abaixo executa essas tarefas em ordem?

```
git remote add repo https://github.com/devuser/development.git
git fetch repo
```

Ordem dos comandos

Um desenvolvedor precisava, a pedido do seu gerente, commitar algumas alterações feitas por um colega no branch onde foi feita a última alteração. Ao acessar o histórico de alterações, ele descobriu que o branch se chamava 'dev'. Depois de selecionar o branch desejado, o desenvolvedor verificou através de um comando que a única alteração disponível era no arquivo 'index.html'.
Depois disso, ele enviou o arquivo para o stage e commitou com a mensagem "home page".
Analisando os eventos descritos no enunciado, complete os campos abaixo com os comandos que foram executados pelo desenvolvedor na ordem em que ocorreram:

git log

git checkout dev

git status

git add index.html

git commit -m "home page"