

Criação de *branches* e *commits* com o Git

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Reconhecer o uso da ferramenta Git para versionamento de código-fonte.
- Descrever as vantagens do Git em relação aos sistemas de versionamento distribuídos ou centralizados e sua estrutura de funcionamento de *branches* e *staging area*.
- Demonstrar a criação de repositórios e a manipulação de arquivos no sistema de versionamento do Git.

Introdução

Para tornar o processo de controle de versão de *software* eficiente, contemplando o fluxo de trabalho das equipes de desenvolvimento, como as alterações realizadas no código-fonte ao longo do projeto, as identificações das versões lançadas, a estrutura de múltiplas frentes paralelas de execução de testes, a homologação, o controle de qualidade e a execução em produção, a ferramenta de gerenciamento de controle de versão deve prover mecanismos que atendam a esses requisitos de forma dinâmica, simples e independente para as pessoas envolvidas no projeto. O conhecimento a respeito da forma como é realizado o controle de versão de *software* por meio de uma ferramenta de gerenciamento é cada vez mais importante no mercado, visto que grande parte das equipes de projetos de *software* na internet, comunidades responsáveis por um determinado repositório público de aplicativo ou mesmo grupos de trabalho colaborativo têm como prática comum a utilização de um sistema de versionamento, a fim de garantir uma gestão mais eficiente das diversas alterações realizadas no código e nos arquivos.

Neste capítulo, você estudará sobre o Git, uma das ferramentas de controle de versionamento mais populares entre os desenvolvedores de sistema. Além disso, conhecerá as principais vantagens do formato de controle de versão do Git em relação às outras ferramentas correspondentes, bem como a estrutura em que os repositórios são dispostos, o formato de *branches* e a manipulação dos arquivos na denominada *staging area*. Por fim, verá os principais modos de criação e gerenciamento dos repositórios e arquivos de projeto dentro do sistema do Git.

1 Conhecendo a ferramenta Git

Criado em 2005, o Git é uma ferramenta de controle de versão de código-fonte disponibilizada em formato de código aberto, utilizada pela grande maioria das equipes de projetos de desenvolvimento de sistemas na atualidade. Conforme Silverman (2013), o Git utiliza um formato de estrutura de projeto com base no modo como a ferramenta é utilizada na prática.

Por meio do uso do Git, é possível criar o histórico de edições realizadas no código-fonte de um projeto, facilitando o processo de consulta do *status* de um arquivo editado e seu conteúdo em um determinado ponto no tempo. Além disso, essa ferramenta facilita a edição paralela entre arquivos, de forma simultânea, entre os vários desenvolvedores em um mesmo projeto, por meio do controle de fluxo de novas funcionalidades com ferramentas para análise e resolução de conflitos das modificações realizadas. Com isso, é possível desenvolver projetos nos quais uma série de pessoas podem contribuir simultaneamente nos mesmos arquivos, da forma mais segura e confiável possível, o que permite que esses conteúdos possam existir sem o risco de suas alterações serem sobrescritas.



Saiba mais

Criador do *kernel* Linux, o finlandês Linus Torvalds também é o responsável pela ferramenta Git (SANTACROCE, 2015). A ferramenta de controle de versão surgiu a partir de uma necessidade da comunidade de desenvolvedores do sistema operacional Linux de manter o compartilhamento das mudanças de código-fonte do *kernel*. A mudança para Git se deu após a ferramenta de sistema de controle de versão utilizada até aquele momento pelo projeto tornar-se paga (CHACON; STRAUB, 2014).

Sempre que os desenvolvedores criam um novo projeto, os arquivos relacionados ao código-base de um sistema são constantemente modificados, mesmo após o lançamento do projeto, por meio de atualização de versões, correção de falhas, adição de novas ferramentas, entre outros aspectos. Conforme Chacon e Straub (2014), em comparação com outros controladores de versão, o Git é categorizado como um **sistema de controle de versão distribuído**, pelo fato de permitir alterações independentes de versões, sem a necessidade de um servidor central para o armazenamento do banco de dados de controle.

O Git tem como principal objetivo auxiliar no acompanhamento das mudanças feitas nos arquivos ao longo do tempo, identificando o registro de quem efetuou uma alteração. Dentre as características que o controle de versionamento com o Git possibilita, destacam-se a possibilidade de restauração de porções de código removido ou previamente modificado, a manutenção da organização do código-fonte entre a equipe de desenvolvimento, a criação de históricos de funcionalidades e o *backup* do código utilizado em si.

O Quadro 1, a seguir, apresenta alguns dos principais termos associados ao Git, os quais são necessários para entender o funcionamento da ferramenta.

Quadro 1. Principais termos do Git

Termo	Descrição
Repositório	O repositório é o diretório onde os arquivos do projeto ficam armazenados , também denominado diretório Git , o qual pode estar presente de forma local ou remota (em outro servidor; CHACON; STRAUB, 2014).
<i>Commit</i>	O <i>commit</i> é o ponto no histórico do projeto que indica um conjunto de modificações realizadas em um ou mais arquivos do projeto naquele momento (CHACON; STRAUB, 2014). Além disso, uma descrição é utilizada para identificar as alterações realizadas nesse determinado ponto.
Diretório de trabalho	Diretório onde os arquivos de uma determinada versão do repositório do projeto ficam disponíveis para acesso e manipulação por parte dos usuários (CHACON; STRAUB, 2014). Esse conteúdo é, na verdade, uma simples cópia dos arquivos contidos no diretório Git .

Uma das principais características de descentralização do controle de versão com o Git é a capacidade de gerenciamento utilizando, em sua maioria, os recursos locais da máquina onde está hospedado o repositório do projeto. Com isso, é possível ter uma velocidade de operação maior nas operações de modificação e consulta de histórico dos arquivos do projeto, ou até mesmo manter a manipulação dos arquivos em um ambiente sem conectividade de rede.

Em um conceito de sistema de controle de versão distribuído, cada máquina na qual estão hospedados os arquivos do projeto se torna independente, com sua própria estrutura de repositório. Segundo Humble e Farley (2014), a grande diferença de um sistema distribuído para outros formatos de controle de versão é a real utilização concorrente dos repositórios por múltiplos usuários. Entretanto, existe a possibilidade de troca entre os arquivos de repositórios diferentes, utilizando-se um servidor remoto para centralizar o conteúdo das máquinas nas quais estão armazenados os arquivos do projeto. Conforme Santacrose (2015), as máquinas passam a se comunicar com o servidor para a atualização dos arquivos e a versão do projeto por meio de operações conhecidas como *push* (realizar o envio dos arquivos locais para a versão do projeto no servidor remoto) e *pull* (baixar os arquivos do servidor na máquina local e atualizar com os arquivos existentes).



Saiba mais

Um dos métodos mais conhecidos para a centralização de repositórios Git em um ambiente remoto é a utilização do serviço em nuvem conhecido como **GitHub**. Essa plataforma atua como um servidor remoto de controle de versão e hospedagem de repositórios de projetos, com os quais é possível interagir por meio do Git.

Segundo Chacon e Straub (2014), para garantir a integridade dos arquivos nos repositórios de projetos, o Git utiliza o conceito de *checksum*, por meio do qual um *hash* do tipo SHA-1 (i.e., uma sequência de 40 caracteres em formato hexadecimal) é sempre gerado para identificar a estrutura de um determinado arquivo no banco de dados da ferramenta de controle de versão. Assim, ao realizar qualquer modificação no arquivo presente no repositório, o Git modificará o valor da sequência de *hash* armazenado.

Os *status* dos arquivos do projeto durante o gerenciamento de versão com o Git podem ser atribuídos em três formas principais, por meio de uma sequência lógica:

- Em um primeiro momento, os desenvolvedores realizam as alterações nos arquivos dentro do repositório do projeto. Com isso, o Git identifica as alterações nos arquivos, porém os conteúdos modificados ainda não sofreram o processo de *commit* no banco de dados de controle de versão, sendo marcados com o *status modified*.
- Em seguida, os arquivos modificados podem ser marcados no Git, a fim de que sejam preparados para fazer parte do próximo processo de *commit* de versão, sendo marcados com o *status staged*.
- Por fim, os arquivos que foram preparados e marcados para serem adicionados são gravados, de forma definitiva, no banco de dados de controle de versão (localizado no diretório Git). Esse é o *status* relacionado diretamente com o *commit* dos arquivos no projeto, sendo marcado com o *status committed*.

O Git é uma ferramenta multiplataforma, com versão disponível para instalação nos principais sistemas operacionais do mercado, como Linux, Windows e Mac OS. Para a sua utilização, a ferramenta possui o gerenciamento de versão de código-fonte tanto por meio da linha de comando quanto pelo formato de aplicação via interface gráfica.



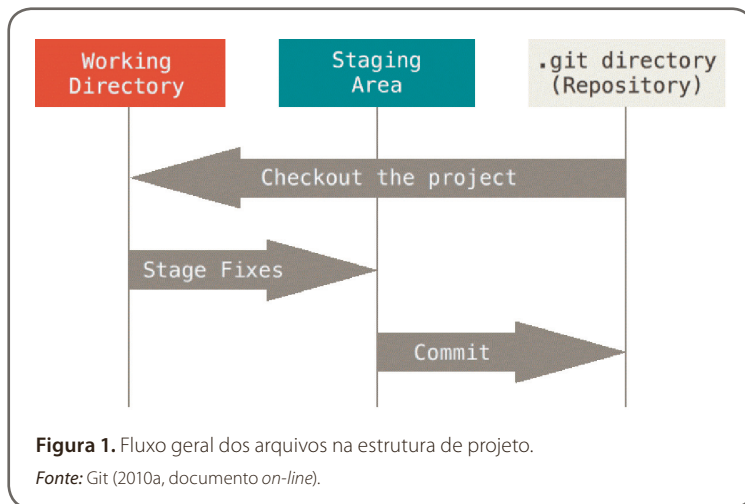
Fique atento

Para instalar o Git em uma determinada plataforma, recomenda-se seguir as instruções da documentação, disponíveis no *site* oficial do projeto.

2 Estrutura de versionamento do Git

Uma das principais diferenças do Git em relação às demais ferramentas de controle de versão é a sua estrutura de gerenciamento dos dados dos arquivos do projeto. Em geral, um sistema de versionamento de código-fonte trata somente das mudanças realizadas nos arquivos, armazenando as alterações que foram feitas, e não o arquivo inteiro. Uma das vantagens da estrutura de versionamento do Git é que a ferramenta trata os dados com um conceito denominado *snapshot*, ou seja, ao realizar um *commit*, é como se o sistema tirasse uma foto dos arquivos existentes, mantendo um valor de *hash* que faz referência à versão anterior. A eficiência no armazenamento se dá pelo fato de que, ao manter essa referência, caso não existam alterações realizadas nos arquivos do repositório, o Git não gravará o arquivo novamente, realizando apenas um apontamento para o conteúdo existente.

A Figura 1, a seguir, apresenta a forma como o Git trabalha com o fluxo geral dos *status* dos arquivos em sua estrutura de projeto (*modified*, *staged* e *committed*).



Como visto na Figura 1, o Git checará se existem alterações nos arquivos do projeto, os quais são editados pelos usuários no diretório de trabalho (*Working Directory*). Ao verificar os arquivos que foram identificados com o *status modified*, estes podem ser encaminhados e preparados para fazer parte do próximo *commit* do projeto. Para isso, os arquivos alterados no projeto são marcados o *status staged*, passando a ser listados em um controle interno do Git, denominado *Staging Area* (SANTACROCE, 2015). Essa área de controle é, basicamente, um arquivo de índice presente no diretório do repositório (diretório Git), no qual está presente a relação de arquivos do diretório de trabalho que estarão no próximo *commit*. Ao efetivar o processo de *commit*, os arquivos passam a ser identificados com o *status committed*, sendo armazenados no diretório Git.

A partir do momento em que o *commit* é realizado, os arquivos são incluídos no *snapshot* feito pelo Git. Todos os arquivos de projeto presentes no *snapshot* gerado são tratados pelo Git com um controle interno, que marca esses arquivos como rastreados. Conforme Chacon e Straub (2014), todos os outros arquivos do diretório de trabalho que não estão incluídos nesta última “foto” ou que não constam com o *status staged* são considerados pela ferramenta como arquivos não rastreados (*untracked*).



Saiba mais

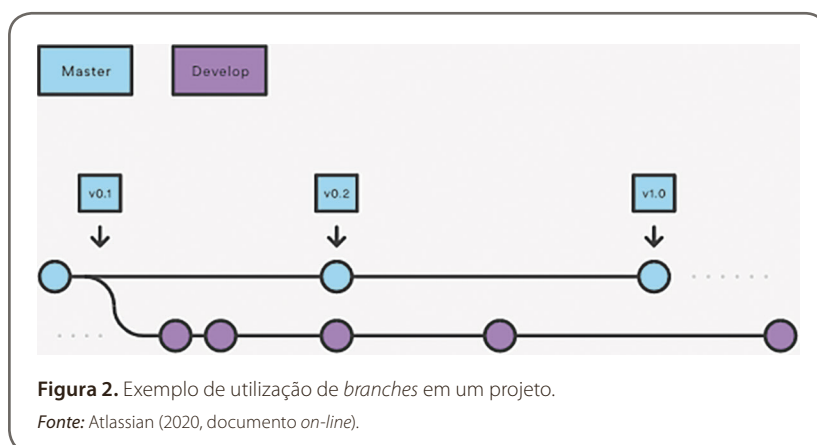
Segundo Chacon e Straub (2014), o Git trata como arquivo não rastreado todo novo arquivo criado no diretório de trabalho de um repositório existente. Além disso, quando um novo repositório de projeto é criado ou iniciado, todos os arquivos presentes no diretório de trabalho são considerados como não rastreados pela ferramenta.

O uso de *branches* no Git

As ferramentas de controle de versão de código-fonte para o gerenciamento de arquivos de um projeto possuem um conceito denominado *branch*. Segundo Silverman (2013), um ***branch*** é uma cópia de todo o conteúdo de uma versão específica do projeto criada para ser acessada por outras pessoas ou equipes, sem afetar a estrutura principal do repositório.

Assim, dentro do projeto, é criada uma ramificação idêntica do conteúdo dos arquivos do projeto, para que possam ser acessados, modificados e receber *commits* de forma independente do conteúdo original. Segundo Chacon e Straub (2014), todo *branch* dentro do Git funciona como um ponteiro de controle, apontando para o último *commit* realizado no conteúdo. A partir do momento em que for executado um novo *commit*, esse ponteiro moverá, de forma automática, a exibição do histórico de controle para esse novo registro.

Lembre-se de que todo *commit* realizado no Git cria um *snapshot* do conteúdo no momento em que foi armazenado. Ao se manipular os mesmos arquivos de um repositório em diferentes *branches*, cada modificação é realizada de forma isolada, embora seja feita no mesmo conteúdo. Com isso, durante um fluxo de trabalho do projeto, as alterações realizadas pela equipe em um *branch* adicional podem ser mescladas, posteriormente, com o *branch* principal do controle de versão do projeto. A Figura 2, a seguir, apresenta um exemplo simples de como uma estrutura de *branches* pode ser utilizada em um projeto.



O exemplo da Figura 2 mostra como o Git pode ser utilizado com diferentes fluxos de trabalho dentro de um mesmo projeto, por meio das ramificações dos históricos registrados das atividades com um determinado código. Neste caso, o *branch Master* é responsável pelos registros das versões específicas do projeto ao longo do tempo, contendo, assim, todos os *commits* realizados nos lançamentos oficiais do sistema em questão. Ao mesmo tempo, o *branch* identificado como *Develop* mostra o histórico dos *commits* realizados pela equipe de desenvolvimento do código ao longo do tempo, como, por exemplo, a adição de funcionalidades ao sistema ou a correção do código existente. No modelo citado, essa organização de *branches* possibilita que o time responsável pelo desenvolvimento tenha acesso para baixar e sincronizar todo o conteúdo do repositório, a fim de manter os arquivos rastreados dentro do *branch Develop*.



Fique atento

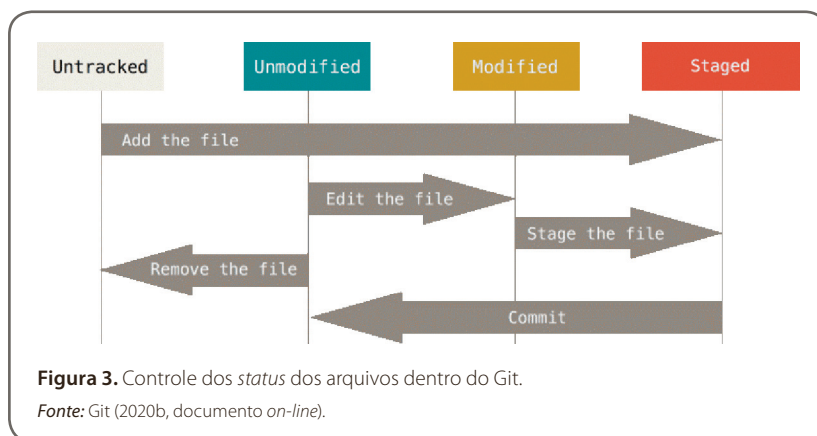
Por padrão, em qualquer repositório criado no Git, é adicionado um *branch* denominado *master*.

A forma como o Git trabalha torna mais simplificada a utilização de diferentes *branches* ao longo do projeto. Durante a utilização da ferramenta, o *branch* que está sendo utilizado pelo usuário para a realização dos *commits* é identificado com um ponteiro interno especial no Git, denominado HEAD. Desse modo, ao criar um novo *branch*, é possível identificar se ele está sendo utilizado para o registro dos *commits* naquele momento ao identificar a presença do ponteiro HEAD em seu *status*.

3 Utilizando o Git com os arquivos e repositórios

Após a instalação da ferramenta de controle de versão de código-fonte em um ambiente de trabalho por meio da linha de comando (ou por meio da interface gráfica, conforme a preferência de instalação), o Git pode realizar diversos métodos de gerenciamento de conteúdo, como a criação de repositórios e a manipulação de arquivos, *commits* e *branches*.

Antes de executar propriamente os comandos, faz-se necessário entender como os arquivos são manipulados no diretório de trabalho em relação ao fluxo de controle através do Git. A Figura 3, a seguir, apresenta um exemplo de como os arquivos do projeto e seus respectivos *status* dentro do controle de versão do Git são referenciados.



Conforme observado na Figura 3, os arquivos não rastreados dentro do diretório de trabalho (p. ex., arquivos novos no diretório ou que não estavam no último *commit*) podem ser adicionados como preparados para o próximo *commit*, recebendo o *status staged*. Ao realizar o *commit*, os arquivos são armazenados no Git, passando a ser monitorados em relação às modificações em seu conteúdo. Se o arquivo rastreado tiver novas alterações, o seu *status* fica como *modified*, sendo necessária a adição desse arquivo como preparado para o próximo *commit*; se não houver nenhuma modificação, o arquivo fica disponível no diretório de trabalho, inclusive para ser removido do conteúdo do projeto.

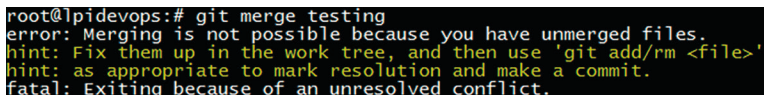
Com a divisão de um projeto no Git entre equipes diferentes, uma das vantagens do controle de versão dos arquivos é a utilização do conceito de **mesclagem de conteúdo** (definido como *merging*). Por exemplo, um determinado código-fonte pode ser alterado pela equipe de desenvolvimento em um *branch* antes de entrar em produção. Para que o arquivo alterado seja carregado no *branch* responsável por deixar o código em produção, a equipe responsável pelo *branch* de produção realiza uma mesclagem com o conteúdo alterado no *branch* de desenvolvimento, mantendo o arquivo final com as últimas alterações.

Em virtude da possibilidade de os usuários atuarem em *branches* diferentes para modificações em paralelo nos arquivos de um projeto, pode ocorrer uma alteração concorrente em um mesmo conteúdo. Por exemplo, um desenvolvedor adiciona uma linha de código em um arquivo `index.html` no *branch master*, porém outro desenvolvedor, realizando modificações no *branch testing*, apaga uma linha de código no mesmo arquivo do repositório. O Git tentará identificar e realizar de forma automática a resolução de conflitos entre as versões ao executar o comando para mesclagem entre um *branch* e outro, o `git merge`. No entanto, podem ocorrer situações em que esse conflito não é resolvido diretamente, sendo necessário identificar e corrigir manualmente as modificações.

Na linha de comando, o exemplo de conflito no arquivo `index.html` poderia ser diagnosticado ao se tentar utilizar o comando `git merge` a partir do *branch master* para mesclar as alterações realizadas no conteúdo do *branch testing*:

```
# git merge testing
```

A Figura 4, a seguir, apresenta a mensagem de erro será mostrada.



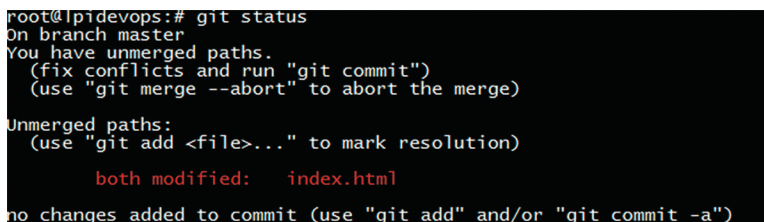
```
root@lpidevops:~# git merge testing
error: Merging is not possible because you have unmerged files.
hint: Fix them up in the work tree, and then use 'git add/rm <file>'
hint: as appropriate to mark resolution and make a commit.
fatal: Exiting because of an unresolved conflict.
```

Figura 4. Saída do comando `git merge`.

Nessa situação, é possível verificar o *status* dos arquivos para detectar qual conteúdo está sendo afetado pelo conflito de versões entre os *branches*. Para isso, pode-se utilizar o comando para a verificação do *status*, o `git status`, da seguinte forma:

```
# git status
```

A Figura 5, a seguir, apresenta a mensagem sobre o conflito entre dois arquivos modificados (em ambos os *branches*).

A terminal window with a black background and white text. The text shows the output of the 'git status' command. It indicates that the user is on the 'master' branch and has unmerged paths. It lists 'index.html' as a file that has been modified in both branches. It provides instructions on how to resolve the conflict using 'git add', 'git commit', or 'git merge --abort'.

```
root@lpidevops:~# git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   index.html

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

Figura 5. Mensagem de detecção de conflito entre conteúdo.

Neste caso, faz-se necessário intervir manualmente no conteúdo, identificando onde está o conflito, a fim de gerar a correção adequada. Para isso, é possível utilizar o comando `git diff` no arquivo indicado (neste caso, o `index.html`) e analisar quais informações estão diferentes entre as versões modificadas:

```
# git diff index.html
```

A Figura 6, a seguir, apresenta a saída do comando por meio do qual o Git insere uma marcação de detecção de conflito para as versões em cada *branch*, representada pelos caracteres “<<<<<<<<” e “>>>>>>>>”. As alterações detectadas são separadas pelos caracteres “=====”.

```

root@lpidevops:~# git diff index.html
diff --cc index.html
index c29f1c7,15980b0..0000000
--- a/index.html
+++ b/index.html
@@@ -1,5 -1,4 +1,8 @@@
<html>
++<<<<<< HEAD
+  <head>Novo Site</head>
++=====
++>>>>>> testing
  <body>
  <h1>Teste de HTML</h1>
  </body>

```

Figura 6. Detecção de conflito com o `git diff`.

Ao lado dos caracteres “<<<<<<”, é identificado o ponteiro *HEAD*, referente ao *branch* atual (neste caso, o *master*). Todo o conteúdo HTML (HyperText Markup Language; ou Linguagem de Marcação de Hipertexto, em português) do arquivo mostrado até os caracteres “=====” são as informações a mais (em verde) que a versão do `index.html` nesse *branch* traz em relação à versão mesclada com o *branch testing*. Lembre-se de que, no exemplo citado, no *branch testing*, o desenvolvedor havia removido uma linha (provavelmente a indicada no código HTML com o título para o *site*). A partir dos caracteres “>>>>>>”, é indicado o conteúdo no *branch testing* (em branco) que é idêntico ao encontrado no mesmo arquivo no *branch master*.

Com a identificação da informação que está em conflito, o desenvolvedor pode editar manualmente o arquivo `index.html` no *branch master* e verificar se a linha citada deve ser mantida ou removida para resolver o problema da mesclagem.



Fique atento

Ao editar novamente o arquivo para a correção manual de conflitos no Git após definir o conteúdo da mesclagem, faz-se necessário remover os caracteres de marcação (“<<<<<<”, “=====” e “>>>>>>”).

Comandos básicos do Git

A primeira coisa a ser feita com a ferramenta é iniciar o uso de um repositório Git. Para isso, faz-se necessário definir o diretório de trabalho, na máquina local, para a utilização dos arquivos.



Fique atento

Os comandos internos do Git funcionam em qualquer uma das plataformas suportadas pela ferramenta. Nos exemplos adotados neste capítulo, serão feitas referências à execução do Git em um ambiente com Linux.

Para iniciar um novo repositório vazio, utiliza-se o comando `git init`. Em um diretório de trabalho definido na máquina em `/home/user/projeto`, deve-se executar o seguinte comando:

```
# cd /home/user/projeto
# git init
```

A Figura 7, a seguir, apresenta o resultado do comando.

```
root@lpidevops:/home/user/projeto# git init
Initialized empty Git repository in /home/user/projeto/.git/
```

Figura 7. Saída do comando `git init`.

Essa sequência de comandos criará o repositório Git dentro do diretório de trabalho, com toda a estrutura para iniciar o controle de versão. O diretório Git será identificado como um subdiretório, denominado `.git`, dentro de `/home/user/projeto` (no caso do exemplo citado).

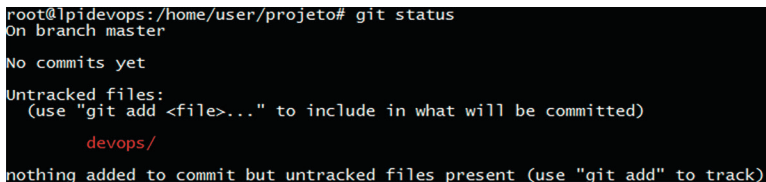
Em vez de iniciar um novo repositório vazio, outra forma seria baixar a cópia de um repositório existente em um servidor remoto. Se essa for a opção escolhida, o comando a ser utilizado no diretório de trabalho criado é o `git clone`, indicando a URL remota do repositório. Com isso, a estrutura do repositório remoto (arquivos do projeto e o diretório de controle `.git`) será copiada para a máquina local. Para isso, é só executar o seguinte comando:

```
# git clone <url remota do projeto>
```

Assim, tanto por meio da criação de uma nova estrutura de repositório com o `git init` como por meio da cópia de um projeto existente com o `git clone`, o diretório de trabalho possui uma estrutura para versionamento dos arquivos existentes em seu conteúdo. Para verificar os *status* dos arquivos do diretório de trabalho dentro do Git, pode-se utilizar o comando `git status` da seguinte forma:

```
# git status
```

A Figura 8, a seguir, apresenta o resultado do comando com o *status*.



```
root@lpidevops:/home/user/projeto# git status
On branch master

No commits yet

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

    devops/

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

Figura 8. Saída do comando `git status`.

Esse comando exibirá todos os arquivos não rastreados, modificados e preparados para o próximo *commit*. Para adicionar novos arquivos criados ou editados no diretório de trabalho para serem rastreados pelo Git, utiliza-se o comando `git add`, indicando o(s) nome(s) do(s) arquivo(s) ou diretório(s) a serem preparados para o próximo *commit*. Assim, é só executar o seguinte comando:

```
# git add README.txt
# git add *.html
```

Para organizar a visualização das versões, é interessante utilizar marcações e identificadores ao longo do histórico de *commits*, adicionando uma *tag*. Conforme Santacroce (2015), as *tags* fazem referência a alterações que podem ser consultadas futuramente, servindo como uma documentação das versões disponibilizadas ao longo do projeto. O comando `git tag` é responsável por adicionar ou consultar as *tags* utilizadas ao longo dos *commits*. Para consultar as *tags* existentes, basta executar o seguinte comando:

```
# git tag
```

Para criar uma nova *tag*, o comando citado deve ser acompanhado com as opções `-a` (criar uma nova *tag* de versão) e `-m` (comentário adicionado à *tag* a ser criada):

```
# git tag -a v1.0 -m "Versão 1.0 do sistema"
```

Os arquivos referenciados com o `git add` estão marcados como preparados para o próximo *commit* dentro do projeto (*status staged*). Com isso, pode-se utilizar o comando `git commit` para armazenar os arquivos no diretório Git. Para isso, é só executar o seguinte comando:

```
# git commit -m "Primeiro commit do projeto"
```

A opção `-m` indica um comentário a ser realizado durante a realização do *commit*. Isso é importante para o registro do histórico das modificações ao longo do projeto, pois indica o que foi realizado naquele momento.



Fique atento

Pode ocorrer uma situação em que um arquivo tenha sido preparado via comando `git add` para fazer parte do próximo *commit*, porém, antes da execução do comando `git commit`, ele tenha sido novamente modificado. Assim, o *status* do arquivo para o Git estará como *modified*, sendo necessário realizar novamente o comando `git add` com o mesmo arquivo, a fim de que as novas alterações sejam armazenadas no próximo *commit*.

Se um arquivo for removido do diretório de trabalho, no *status* do diretório Git, ele irá aparecer como um conteúdo a ser excluído do repositório. Para que o arquivo seja removido do repositório no próximo *commit*, utiliza-se o comando `git rm`, indicando o(s) nome(s) do(s) arquivo(s) a ser(em) removido(s). Para isso, basta executar o seguinte comando:

```
# git rm robots.txt
```

Assim, no próximo `git commit`, o arquivo referenciado será removido do repositório do projeto. Além disso, é possível retirar o *status staged* de arquivos preparados para o próximo *commit*, como, por exemplo, em uma situação em que um arquivo modificado no diretório de trabalho foi adicionado de forma incorreta via `git add` para fazer parte do próximo *commit*. Para isso, é só utilizar o comando `git reset`, indicando o ponteiro de controle do Git para o *branch* atual e o nome do arquivo que será necessário retirar da *staging area*.

No caso de um arquivo `index.html` que foi adicionado acidentalmente à *staging area* via `git add`, ele deve ser removido via `git reset` da lista de arquivos preparados para o próximo *commit*. Para isso, basta utilizar o seguinte comando:

```
# git reset HEAD index.html
```



Referências

ATLASSIAN. *Gitflow workflow*. [2020]. Disponível em: <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>. Acesso em: 27 ago. 2020.

CHACON, S.; STRAUB, B. *Pro Git: everything you need to know about Git*. 2nd ed. New York: Apress, 2014.

GIT. *1.3 getting started: what is Git?* [2020a]. Disponível em: <https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F>. Acesso em: 27 ago. 2020.

GIT. *2.2 Git basics: recording changes to the repository*. [2020b]. Disponível em: <https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository>. Acesso em: 27 ago. 2020.

HUMBLE, J.; FARLEY, D. *Entrega contínua: como entregar software de forma rápida e confiável*. Porto Alegre: Bookman, 2014.

SILVERMAN, E. *Git: guia prático*. São Paulo: Novatec, 2013.

SANTACROCE, F. *Git essentials*. Birmingham: Packt Publishing, 2015.

Leituras recomendadas

NAREBSKI, J. *Mastering Git*. Birmingham: Packt Publishing, 2016.

PIPINELLIS, A. *GitHub essentials*. Birmingham: Packt Publishing, 2015.



Fique atento

Os *links* para *sites* da *web* fornecidos neste capítulo foram todos testados, e seu funcionamento foi comprovado no momento da publicação do material. No entanto, a rede é extremamente dinâmica; suas páginas estão constantemente mudando de local e conteúdo. Assim, os editores declaram não ter qualquer responsabilidade sobre qualidade, precisão ou integridade das informações referidas em tais *links*.