Treinamento

Git

O que é?

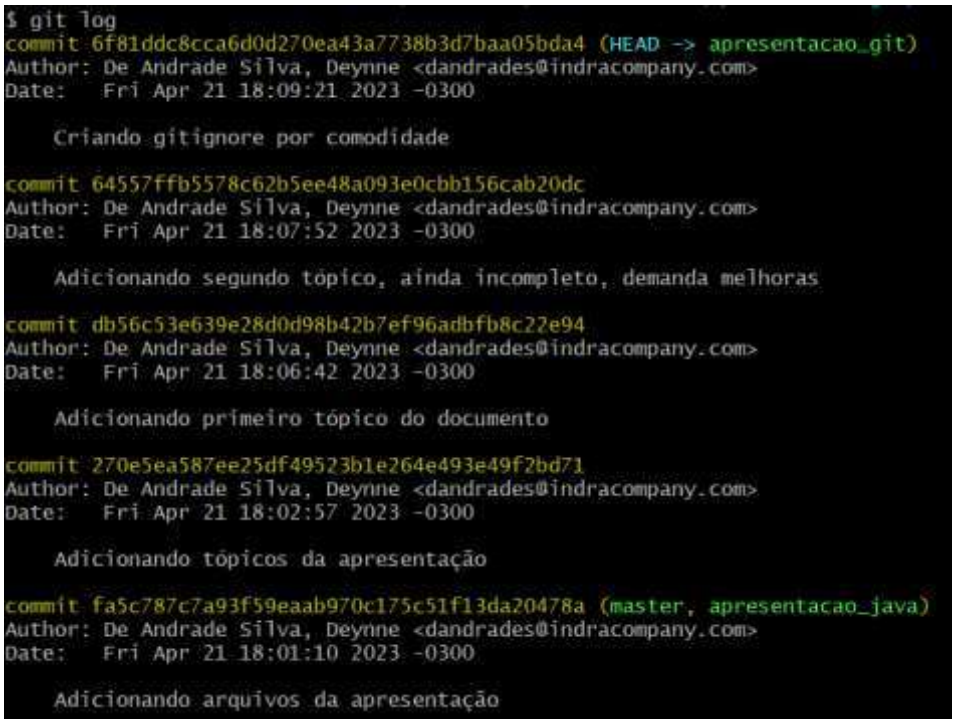
O Git foi inicialmente projetado e desenvolvido por Linus Torvalds para o desenvolvimento do kernel Linux, mas foi adotado por muitos outros projetos. Ele é um sistema de controle de versões grátis e de código aberto, sendo geralmente utilizado para desenvolvimento de softwares, mas não sendo limitado a isto. Cada repositório gerenciado pelo git tem, a partir do momento da implantação da ferramenta, um registro completo das alterações realizadas e permite revisões, reversões e ramificações conforme necessário.

Certo, mas o que ele faz exatamente?

Em resumo, o git permite que você salve “momentos” do projeto (também chamado de repositório) a cada alteração, vamos chamá-los de commits. Estes commits gravam o estado do projeto no momento daquela alteração e assim como registram o que foi alterado, quando foi alterado e quem realizou esta alteração. Mas ele não se limita a isto.

Ele permite a ramificação do projeto de modo que alterações feitas não afetam o trabalho original. Estes ramos ou branchs podem crescer por conta própria e, quando necessário, unirem-se ao ramo original, permitindo a mistura das informações.

Mas até agora tudo ainda parece muito abstrato, vamos então para um exemplo. Este texto, por exemplo, foi criado em um repositório git, registrando as mudanças conforme elas ocorreram. Podemos visualizar o histórico de alterações na seguinte imagem:



Também podemos verificar que, visando não “misturar” os tópicos do projeto foram criadas branchs para separá-los. Temos a branch “master”, como branch principal, mas também as branchs “apresentacao_git” e “apresentacao_java”.

O git nos permite mover por essa estrutura de commits com liberdade uma vez que caso o arquivo seja corrompido, a qualquer momento, se pode voltar para um commit anterior, antes de ele ser corrompido e salvar uma boa parte do trabalho. Além disto, qualquer alteração que eu faça na branch **apresentacao_git** não afeta a branch **apresentacao_java** de modo que não é necessário se preocupar que uma alteração na apresentação atrapalhe a outra. Por fim, ainda será possível unir ambos os trabalhos em um trabalho principal (branch máster).

E os Repositórios remotos?

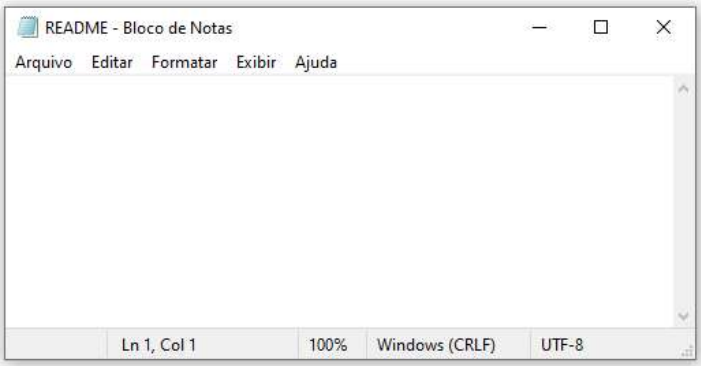
O git também está preparado para auxiliar o trabalho em grupo, mas percebe-se que há uma uma barreira para trabalhos neste modelo. Neste ponto consideramos os repositórios remotos. Ambientes em nuvem de armazenamento de projetos que funcionam com base no git. Alguns exemplos destes ambientes são: github, gitlab e bitbucket.

Os repositórios remotos funcionam como um git compartilhado onde todos os criadores podem adicionar e gerenciar seus trabalhos. O git será encarregado de “unir” estes trabalhos e ajustar, quando possível, as alterações para melhor compreender o trabalho geral. Deve-se ressaltar, porém que há uma separação entre repositório local e remoto para o git.

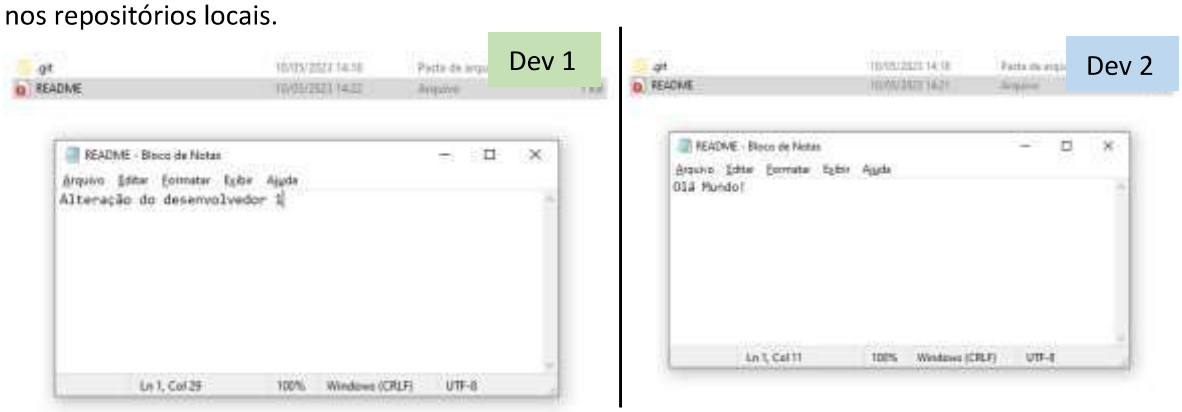
Uma alteração feita num repositório git local não necessariamente deve ser salva em um repositório remoto. É comum criar branchs pra testes e experimentação de funcionalidades que não irão para o projeto final, assim, o trabalho com repositórios remotos demanda que as alterações sejam salvas em commits não só no ambiente local, mas também que estes commits e branchs sejam ativamente salvos no repositório remoto.

Vamos utilizar um exemplo para especificar como é feito o trabalho com um repositório remoto.

Um repositório exemplo1 foi criado no github contendo apenas um arquivo de nome README, o qual estava vazio.



Então o repositório foi clonado (uma copia dele foi criada localmente) por dois desenvolvedores. Ambos os desenvolvedores realizaram uma alteração no arquivo e salvaram nos repositórios locais.



Um dos desenvolvedores decidiu que esta alteração deveria ser salva também no remoto, e assim o fez.

```
PS C:\Users\dandrades\Projetos\Treinamento\exemplo01> git add .
PS C:\Users\dandrades\Projetos\Treinamento\exemplo01> git commit -m "Adicionando 'Olá mundo'"
PS C:\Users\dandrades\Projetos\Treinamento\exemplo01> git push origin main
```

O segundo, por sua vez, quando tentou fazer o mesmo foi alertado de que o repositório remoto já havia sido atualizado e, assim, precisava trazer para seu repositório local a nova versão do projeto antes que pudesse adicionar suas próprias alterações ao repositório remoto.

```
PS C:\Users\dandrades\Projetos\Treinamento\exemplo01> git push origin main
To https://github.com/Deynne/exemplo01.git
 ! [rejected]        main -> main (non-fast-forward)
error: failed to push some refs to 'https://github.com/Deynne/exemplo01.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

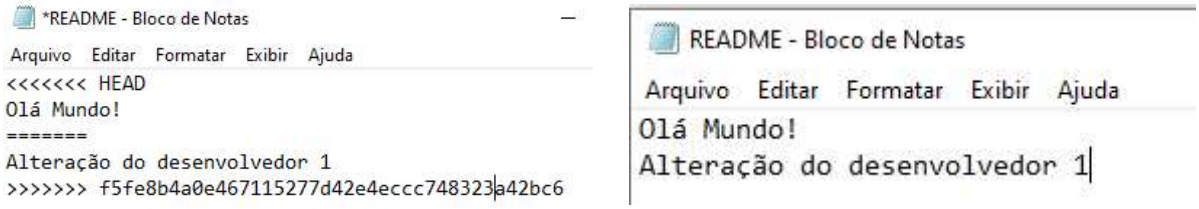
No processo de atualizar seu repositório local com os dados do repositório remoto o desenvolvedor 2 se deparou com um conflito, pois ambos os desenvolvedores adicionaram informações no mesmo arquivo.

```
PS C:\Users\dandrades\Projetos\Treinamento\exemplo01> git pull origin main
From https://github.com/Deynne/exemplo01
 * branch            main       -> FETCH_HEAD
Auto-merging README
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.
PS C:\Users\dandrades\Projetos\Treinamento\exemplo01>
```

Assim, o git indicou que ele precisava resolver os conflitos para poder seguir, sinalizando onde estes conflitos haviam ocorrido.

```
PS C:\Users\dandrades\Projetos\Treinamento\exemplo01> git diff
diff --cc README
index a6425e3,a1f19ac..0000000
--- a/README
+++ b/README
@@@ -1,1 -1,1 +1,5 @@@
- Alteração do desenvolvedor 1
++<<<<<<< HEAD
 +Olá Mundo!
++=======
++Alteração do desenvolvedor 1
++>>>>>>> f5fe8b4a0e467115277d42e4eccc748323a42bc6
PS C:\Users\dandrades\Projetos\Treinamento\exemplo01>
```

Ajustando o arquivo em questão o commit pôde ser realizado.




```
PS C:\Users\dandrades\Projetos\Treinamento\exemplo01> git add .
PS C:\Users\dandrades\Projetos\Treinamento\exemplo01> git commit -m "Merge mantendo as duas alterações"
[main af25332] Merge mantendo as duas alterações
PS C:\Users\dandrades\Projetos\Treinamento\exemplo01> git push origin main
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (6/6), 631 bytes | 315.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/Deynne/exemplo01.git
   f5fe8b4..af25332  main -> main
```

Perceba que um novo commit precisou ser feito indicando esta junção (merge). Um outro ponto importante é que a correção nem sempre preza por manter ambas as alterações, por vezes é necessário eliminar uma das mudanças para melhor coesão do trabalho final.

Está bem, mas como eu uso?

Com a compreensão do conceito do git pode-se abordar como utilizá-lo. Geralmente um comando git é estruturado no padrão “git <comando> [flags opcionais] [informações adicionais do comando]”. Por exemplo:

```
git commit -m “texto”
```

Este é o comando para realização de um commit e, por padrão pede um comentário. A flag **`-m`** é um indicador de que este comentário será dado junto ao próprio comando e o **“texto”** indica o texto deste comentário.

As flags do git são diversas e variam de comando para comando. Em geral podemos usar flag **`--help`** para obter uma descrição detalhada dos funcionamentos de qualquer comando. Ex:

```
git commit --help
```

Este comando geralmente abre uma pagina no browser com a descrição do comando selecionado.

Segue uma lista de alguns dos comandos do git:

1. `git init`: É o comando principal para iniciar um repositório local git. Ele cria um repositório vazio, sem branchs.
2. `git clone <url>`: É o comando para clonar o repositório remoto para o repositório local
3. `git status`: É o comando que permite verificar as alterações que o repositório sofreu até o momento. Todas as alterações listadas por este comando ainda não foram salvas em commits
4. `git add <regex de adição>`: Adiciona todas as alterações selecionadas para a lista de alterações que serão salvas em commits.
   `git add .` adiciona tudo
5. `git commit`: Salva as alterações e registra data de criação e criador do commit. Demanda que um comentário seja adicionado para a alteração realizada.
   `git commit -m "texto"` faz commit com mensagem no comando
6. git reset: reverte o git add (não gera perdas) ou pode "reverter" um commit (processo pode gerar perdas)
7. `git log`: mostra a lista de commits feitos até o momento para branch atual.
8. git branch: gerencia a visualização, criação e deleção as branchs do código
   `git branch` lista branchs
   `git branch -c <nome branch>` cria branchs
   `git branch -d <nome branch>` deleta branchs
9. `git checkout/switch <nome branch ou id commit>`: permite mudar de uma branch para outra. Também permite mudar para um commit.
10. `git merge <id da branch>`: Permite unir dados da branch indicada na branch atual.
11. `git stash`: permite adicionar ou remover alterações da "pilha de alterações" (Simula um commit).
    `git stash push` insere na pilha.
    `git stash pop` remove
12. `git remote`: permite gerenciar informações do repositório remoto.
    `git remote -v` exibe os repositórios remotos do projeto git e seus ids (geralmente o id é **origin**)
13. `git fetch <id remoto> <branch remota>`: Permite verificar o repositório remoto e identificar o que foi alterado. (não muda o repositório local)
14. `git pull <id remoto> <branch remota>`: Permite verificar o repositório remoto, identificar o que foi alterado e trazer para o local. (muda o repositório local)
15. `git push <id remoto> <branch remota>`: Salva o estado atual da branch atual no repositório remoto e na branch remota indicadas.
16. `git tag <nome da tag>`: permite o gerenciamento das tags do projeto

É possível obter uma “folha de cola” do git, no seguinte link: https://education.github.com/git-cheat-sheet-education.pdf

Ignores

Saber o que não salvar no git é tão importante quanto saber o que salvar. Em projetos de desenvolvimento é comum que arquivos de build e bibliotecas consumam uma grande parcela de espaço sejam alterados com frequência. Assim, não é viável ou cômodo armazenar estes dados num repositório git. Assim, podemos usar os ignores do git.

O git possui 3 ignores, o ignore de sistema, o ignore de repositório local e o ignore de repositório local e remoto. Todos os 3 funcionam da mesma forma, através de regex que indicam as pastas, arquivos e tipos de arquivo que o git não deve considerar no repositório. Ignores de níveis mais baixos tem precedência sobre os de níveis mais altos.

É importante ressaltar porém que, uma vez que um arquivo seja adicionado a um commit ele não será ignorado, independente do arquivo de ignore, o que pode gerar algumas complicações, então é importante que o arquivo de ignore seja um dos primeiros do projeto. Atualmente, costuma ser fácil de encontrar ignores prontos para a maioria das linguagens e ferramentas mais comumente utilizadas pela internet.

O ignore de sistema pode ser definindo alterando a variável core.excludesFile, do arquivo .gitconfig (encontrado em geral em ~/user/.gitconfig), com o caminho do arquivo de ignore (necessita que o arquivo seja criado). Já o ignore de repositório local se encontra no repositório como o arquivo .git/info/exclude, o qual pode ser editado para adicionar os ignores. E por fim, o git identifica todo arquivo .ignore como um arquivo de ignore para o projeto inteiro (repositório local e remoto).