O GIT DAS GALÁXIAS

UMA ODISSÉIA NA PROGRAMAÇÃO

MARCELO HADAD

01

GIT

“Comofas”

Introdução ao Git

Desenvolvido por Linus Torvalds em 2005, o Git é um sistema de controle de versão distribuído, o que significa que cada desenvolvedor tem uma cópia completa do histórico de um projeto em seu próprio computador. Isso proporciona flexibilidade e robustez excepcionais ao gerenciar o código-fonte de um projeto.
O Git permite que várias pessoas trabalhem simultaneamente no mesmo projeto, facilitando a colaboração e o gerenciamento de diferentes versões de um software.

"Para começar a utilizar o Git, é necessário instalá-lo e configurá-lo no seu computador. As etapas para instalação variam dependendo do sistema operacional que você está utilizando.

No Windows você pode acessar o site oficial do Git(https://git-scm.com/), fazer o download do instalador, executar e seguir as instruções na tela.
No macOS e Linux, você pode instalar o Git usando os gerenciadores de pacotes de sua distribuição. Por exemplo, no macOS, você pode usar o Homebrew, e no Linux, você pode usar o apt-get no Ubuntu ou o yum no CentOS.

Configurando o Git

Antes de sair usando o git, você precisa definir um nome de usuário e e-mail que serão utilizados nos commits.
Para isso, siga o exemplo abaixo:

```
$ git config --global user.name "Quase Nada"
$ git config --global user.email quasenada@example.com
```

Nesse exemplo estamos usando a opção **-- global**, essa opção indica ao git que você quer aplicar essa configuração apenas ao usuário atual.

Além disso, há duas outras opções:

Usando o **–system**, a configuração se aplica a todos os usuários e todos os repositórios no sistema.
Já com o **--local** você aplica a configuração apenas no repositório que estiver ativo, para usar essa opção você precisa estar com o repositório git iniciado. Vamos falar sobre isso no próximo capítulo.

02

DIA-A-DIA

“Posso fazer isso
o dia todo”

git init

O primeiro passo para iniciar o uso do Git em um novo projeto é executar o seguinte comando no terminal:

```
$ git init
Initialized empty Git repository in D:/repos/ebook/.git/
```

O comando **git init** é utilizado para inicializar um novo repositório Git em um diretório existente. Quando você executa **git init**, o Git cria uma nova subpasta chamada **.git** no diretório atual. Essa pasta é onde o Git armazena todo o histórico de versões, metadados e configurações para o seu projeto.

Basicamente, o **git init** transforma um diretório comum em um repositório Git. Sendo um repositório Git, todas as mudanças nos arquivos desse diretório serão rastreadas e você poderá utilizar todos os recursos do controle de versão do Git, como criar commits, criar branches, mesclar alterações e muito mais.

git status

O comando **git status** obtém informações sobre o estado atual do seu repositório. As informações fornecidas pelo comando são as seguintes:

Branch Atual: Refere-se à branch na qual você está atualmente trabalhando.
Arquivos Modificados: Inclui todos os arquivos no seu diretório de trabalho que foram modificados desde o último commit, incluindo aqueles que foram editados, excluídos ou adicionados recentemente.
Arquivos na Staging Area: São os arquivos que foram adicionados à área de preparação (staging area) e estão prontos para serem incluídos no próximo commit.**Arquivos Não Rastreados**: Referem-se aos arquivos que estão presentes no diretório de trabalho, mas que o Git ainda não está rastreando.

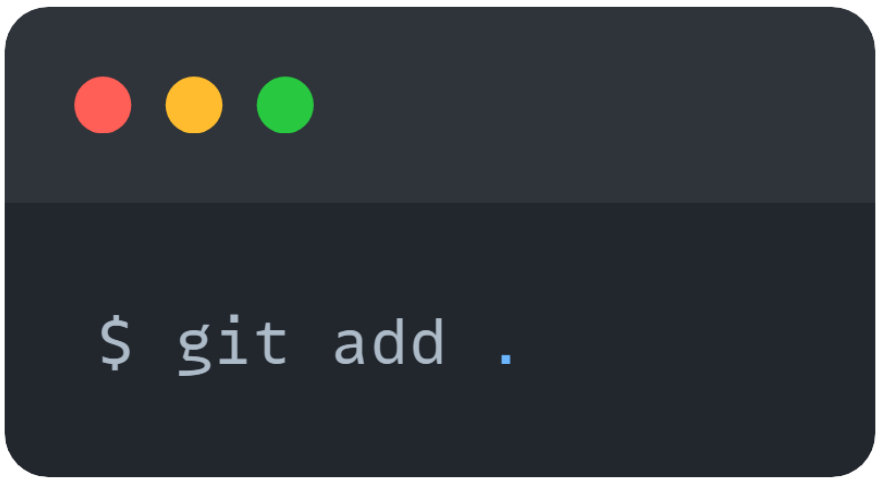
```
$ git status
On branch master

No commits yet

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

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

git add

Com o commando **git add**, você poderá adicionar suas alterações à staging area, que é uma área onde você indica ao Git quais arquivos serão incluídos no próximo commit.

Existem várias opções para adicionar os arquivos:

git add .: Este comando adiciona todos os arquivos editados, excluídos ou adicionados recentemente, utilizando o ponto para representar todos os arquivos no diretório.
git add fileA.txt: Este comando adiciona um arquivo específico chamado fileA.txt.
git add *.txt: Este comando adiciona todos os arquivos com a extensão .txt.



git commit

O comando **git commit** é usado para criar um novo commit no repositório Git. Um commit é essencialmente um "ponto de verificação" que representa uma mudança específica no seu projeto.

Ao executar este comando, o Git cria um novo commit que inclui todas as alterações atualmente na staging area.

Você pode adicionar uma mensagem de commit usando a opção **-m** seguida da sua mensagem entre aspas. Por exemplo:

```
$ git commit -m "feat: add a nice commit"
```

A mensagem de commit é uma descrição curta e significativa das alterações que você está adicionando com o commit.

Se você não fornecer uma mensagem de commit usando **-m**, o Git abrirá o editor de texto padrão para que você possa escrever uma mensagem mais longa e detalhada.

git clone

O comando **git clone** é utilizado para criar uma cópia local de um repositório Git existente. Ele realiza uma cópia completa de todo o histórico de commits, branches e arquivos do repositório remoto para o seu diretório local.

Para clonar um repositório, você precisa fornecer a URL do repositório remoto como argumento para o comando **git clone**. Por exemplo:

```
$ git clone <repo_URL>
```

Isso criará uma cópia local do repositório remoto na pasta atual, com o mesmo nome do repositório.

O comando **git clone** obtém todo o histórico de commits do repositório remoto, incluindo todas as branches e tags.
Isso significa que você terá acesso a todas as versões do código, assim como os colaboradores do projeto original.

git remote

O comando **git remote** é utilizado para lidar com repositórios remotos em um projeto Git. Ele permite visualizar, adicionar e remover conexões com repositórios remotos.

Para visualizar os repositórios remotos atualmente configurados no seu projeto, você pode usar o seguinte comando:

```
$ git remote -v
```

Isso mostrará as URLs dos repositórios remotos, bem como seus apelidos associados.

Além disso, você pode usar outros comandos com o git remote para adicionar, remover ou renomear um repositório remoto, utilizando respectivamente os seguintes comandos:

```
$ git remote add <nick> <repo_URL>
$ git remote remove <nick>
$ git remote rename <current_name> <new_name>
```

git fetch

O comando **git fetch** é utilizado para buscar todas as atualizações das branches de um repositório remoto para o seu repositório local. No entanto, ele não integra essas atualizações automaticamente à sua branch local, em vez disso, ele as traz para o seu repositório local para que você possa revisá-las e decidir o que fazer com elas.

Essas atualizações podem incluir commits, branches, tags e outras referências que podem ter sido adicionadas ao repositório remoto desde a última vez que você sincronizou.

git merge

O comando **git merge** é utilizado para combinar as alterações de duas branches diferentes. Ele une o histórico de commits de duas branches e cria um novo commit que incorpora as mudanças de ambas.

Para realizar um merge simples, você primeiro precisa estar na branch para onde deseja mesclar as alterações. Em seguida, você executa o comando **git merge** seguido pelo nome da branch que deseja mesclar. Por exemplo:

```
$ git merge another_branch
```

Isso irá combinar as alterações da branch **another_branch** com a branch atual e criar um novo commit de merge.

git merge

Às vezes, podem ocorrer conflitos de mesclagem durante o processo de merge, especialmente se duas branches modificarem o mesmo trecho de código de maneiras diferentes.

Se ocorrerem conflitos, o Git indicará quais arquivos têm conflitos de mesclagem. Você precisará resolver esses conflitos manualmente, editando os arquivos conflitantes para escolher quais alterações manter e como combiná-las.

Após resolver os conflitos, você precisa adicionar os arquivos modificados usando **git add** e, em seguida, confirmar o merge usando **git commit**.

O Git suporta diferentes tipos de merges:

O merge **fast-forward** ocorre quando a branch a ser mesclada pode ser movida facilmente à frente da branch atual, sem criar um novo commit de merge. Isso acontece quando não há alterações na branch atual desde que a branch foi bifurcada.
O merge **recursivo** é o tipo padrão de merge usado quando é necessário criar um novo commit de merge para combinar as alterações de duas branches.

git pull

O comando **git pull** é utilizado para buscar as atualizações do repositório remoto e integrá-las automaticamente à sua branch local. Ele combina dois comandos: **git fetch** para buscar as atualizações e **git merge** para mesclá-las à sua branch local.

Como o **git pull** executa implicitamente um **git merge**, é possível encontrar conflitos de mesclagem ao executar o comando, especialmente se você e seus colegas de equipe estiverem trabalhando nos mesmos arquivos.
Se ocorrerem conflitos de mesclagem, o Git irá sinalizá-los, exigindo que você os resolva manualmente antes de poder continuar com o **git pull**.

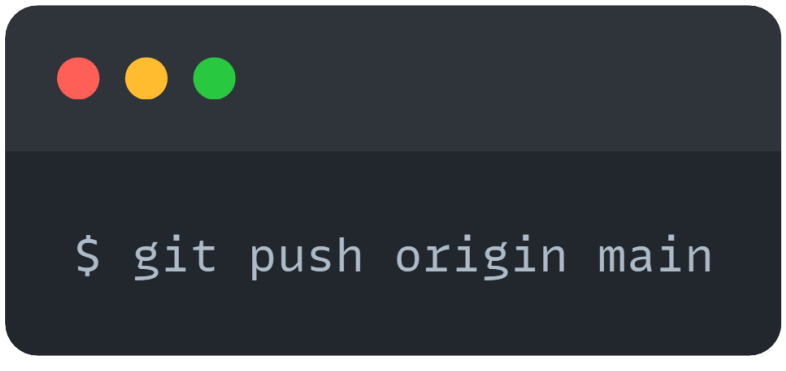
git push

O comando **git push** é usado para enviar as alterações do seu repositório local para um repositório remoto. Ele atualiza o repositório remoto com as mudanças que você fez na sua branch local. Isso significa que os outros colaboradores do projeto podem ver e acessar suas mudanças.
Após o **git push**, as atualizações da sua branch local se tornam parte do histórico do repositório remoto, permitindo que outros membros da equipe as vejam e trabalhem com elas.

Se você criou uma nova branch local e deseja enviá-la para o repositório remoto, pode fazer isso usando **git push** com a opção **-u** ou **--set-upstream**.

Por exemplo: **git push -u origin new_branch** define **new_branch** como a branch remota padrão para futuros **git pull** e **git push**.

A sintaxe básica do **git push** é **git push <remote_repo> <local_branch>**. Por exemplo:



git branch

O comando **git branch** é utilizado para listar, criar, renomear e excluir branches em um repositório Git.

Quando você executa **git branch** sem argumentos, o Git lista todas as branches presentes no repositório local.

Para criar uma nova branch, você pode usar **git branch** seguido pelo nome da nova branch. Por exemplo:

```
$ git branch new_branch
```

Você pode mudar para uma branch diferente utilizando **git checkout** seguido pelo nome da branch desejada. Por exemplo:

```
$ git checkout another_branch
```

git branch

O comando **git branch -m** seguido pelo nome atual da branch e pelo novo nome permite renomear uma branch local. Por exemplo:

```
$ git branch -m old_branch new_branch
```

Para excluir uma branch local, você pode usar **git branch -d** seguido pelo nome da branch a ser excluída. Por exemplo:

```
$ git branch -d branch_to_delete
```

Para listar branches remotas, você pode usar o comando **git branch -r**. Isso mostrará todas as branches remotas rastreadas pelo seu repositório local.

03

HISTÓRICO

“O que que tá acontecendo?”

git log

O comando **git log** é utilizado para visualizar o histórico de commits de um repositório Git. Ele exibe uma lista detalhada de todos os commits, mostrando informações como autor, data, mensagem de commit e o hash SHA-1 de cada commit.

O **git log** possui várias opções de formatação que permitem personalizar a saída do log de acordo com suas preferências. Por exemplo, você pode usar **--oneline** para exibir cada commit em uma única linha.

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

Outras opções de formatação incluem **--graph** para exibir uma representação gráfica do histórico de commits e **--decorate** para exibir referências de branch e tags ao lado dos commits.

git log

Você pode filtrar o histórico de commits usando várias opções do **git log**. Por exemplo, é possível visualizar apenas os commits feitos por um autor específico usando **--author** seguido do nome do autor.

Além disso, é possível limitar o número de commits exibidos usando a opção **-n**, seguida do número desejado de commits.

```
$ git log --author "Quase Nada" -n 3
```

Você pode utilizar o **git log** em conjunto com opções como **-p** ou **--patch** para exibir as diferenças introduzidas em cada commit. Isso mostra as alterações específicas feitas em cada commit.

git diff

O comando **git diff** é utilizado para visualizar as diferenças entre o estado atual do seu diretório de trabalho e o estado do repositório Git. Ele mostra as alterações feitas em arquivos que ainda não foram adicionados à staged area, bem como as diferenças entre a staged area e o último commit.

Quando você executa **git diff** sem argumentos, o Git mostra as diferenças entre os arquivos no seu diretório de trabalho e os arquivos na staged area.
Isso mostra as alterações que você fez desde o último commit, mas que ainda não foram adicionadas à staged area.

Você pode especificar arquivos ou diretórios específicos como argumentos para o **git diff** para ver apenas as diferenças nesses arquivos ou diretórios. Por exemplo:

Isso mostrará as diferenças no arquivo **FileA.txt** entre o diretório de trabalho e a staging area.

git diff

Se você quiser ver as diferenças entre a staging area e o último commit, você pode usar **git diff --staged** ou **git diff --cached**. Esses comandos são equivalentes e mostram as diferenças entre o que está na staging area e o último commit.
Isso é útil para revisar as alterações que serão incluídas no próximo commit antes de fazer o commit.

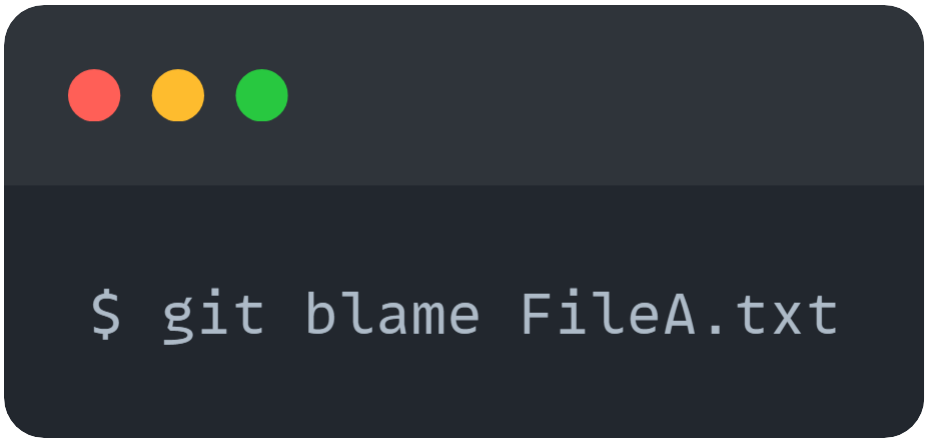
Além disso, você pode usar **git diff** com hashes de commits para visualizar as diferenças entre commits específicos. Por exemplo:

```
$ git diff last_commit_hash..actual_commit_hash
```

Isso mostrará as diferenças entre o commit anterior e o commit atual.

git blame

O comando **git blame** é uma ferramenta útil para rastrear a autoria de cada linha em um arquivo específico. Ele exibe o autor de cada linha, juntamente com o hash do commit onde a linha foi modificada pela última vez.



O **git blame** é frequentemente usado para identificar responsabilidades sobre partes específicas de um código-fonte. Por exemplo, ao revisar um trecho de código problemático, você pode usar **git blame** para identificar quem é responsável por esse trecho e discutir possíveis melhorias ou correções com o autor original.

git blame

O **git blame** possui várias opções que podem ser usadas para personalizar sua saída. Por exemplo, você pode usar **-L** para especificar um intervalo de linhas específico a ser analisado, ou **-C** para habilitar a detecção de cópias (identificar origens de linhas copiadas de outros lugares no código).

A detecção de cópias é útil para entender as origens das linhas copiadas e quem foi o autor original dessas linhas.

```
$ git blame -C FileA.txt
```

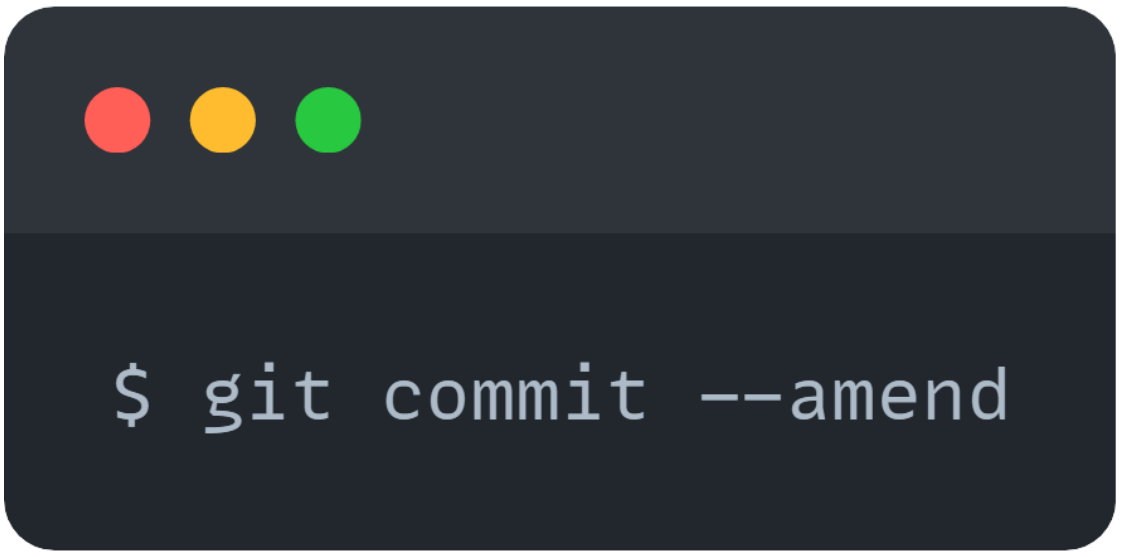
04

REFATORAÇAO

“Errei, fui mlk”

git commit --amend

O comando **git commit --amend** é utilizado para fazer alterações em um commit existente. Ele combina a etapa de staging (adicionar arquivos) e a criação de um novo commit em uma única etapa, permitindo que você faça correções ou adicione mais alterações ao commit mais recente.



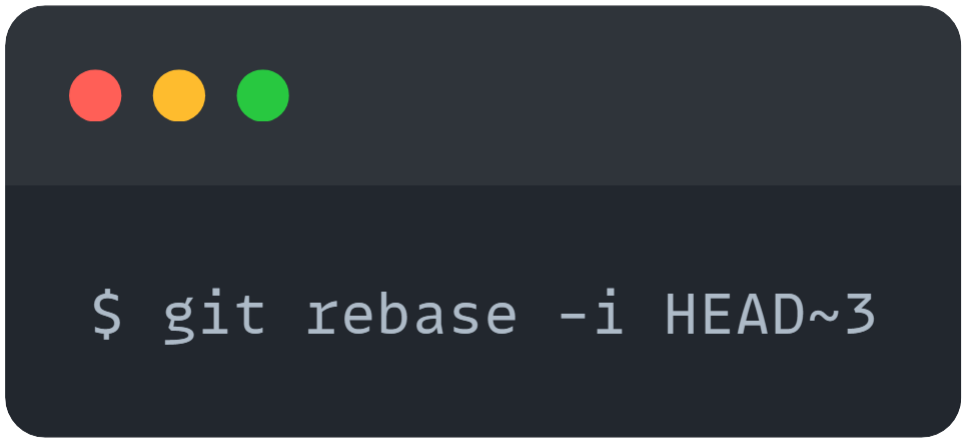
Quando você executa **git commit --amend**, o Git abre um editor de texto (geralmente o editor padrão configurado no seu sistema) com a mensagem de commit do último commit.
Com o **git commit --amend** você pode editar a mensagem do commit. Além disso, você pode adicionar mais alterações ao commit adicionando-as com o **git add**.
Após adicionar as alterações, você executa **git commit --amend** para criar um novo commit que inclui tanto as alterações adicionadas quanto a edição da mensagem de commit.

git rebase -i

O comando **git rebase -i** é uma ferramenta avançada do Git que permite reorganizar, editar e combinar commits durante o processo de rebase de uma branch. A opção **-i** significa "interativo", indicando que você estará interagindo com uma interface interativa para especificar as operações desejadas durante o rebase.

Isso permite que você reescreva o histórico de commits de forma interativa, o que é útil para reorganizar, editar mensagens de commit, combinar commits e até mesmo descartar commits desnecessários.

Para iniciar o rebase interativo, você precisa especificar o commit a partir do qual deseja iniciar o rebase. Por exemplo:



git rebase -i

Isso abrirá um editor de texto com uma lista de commits desde o commit atual até três commits atrás. Você pode ajustar essa lista conforme necessário.
Na interface interativa, cada linha representa um commit na ordem em que eles serão aplicados durante o rebase. Ao lado de cada commit, há uma palavra-chave que especifica uma operação a ser executada durante o rebase.

As operações mais comuns incluem:

pick: Aplica o commit sem fazer alterações.
reword: Permite editar a mensagem de commit.
edit: Pausa o rebase para permitir que você faça alterações no commit.
squash ou **fixup**: Combina o commit com o commit anterior.
drop: Descarta o commit, removendo-o do histórico.

Você pode ajustar a ordem dos commits, combinar commits, editar mensagens de commit e até mesmo remover commits indesejados.

05

AGRADECIMENTOS

“Obrigado amigo,
você é um amigo.”

agradecimentos

Obrigado por acompanhar até o final!

Este eBook foi elaborado com o auxílio da inteligência artificial e aprimorado através da revisão e diagramação cuidadosa realizada por um profissional humano. Acredito que a combinação entre tecnologia e expertise humana resultou em um conteúdo de qualidade que espero que tenha sido útil e esclarecedor para você.

Gostaria de expressar minha sincera gratidão ao professor Felipe Aguiar, à DIO e à Formação ChatGPT for Devs.

Se você tiver alguma dúvida, comentário ou sugestão, não hesite em entrar em contato comigo. Estou sempre aberto a feedbacks e pronto para ajudar no que for necessário.

Mais uma vez, obrigado por fazer parte deste projeto e por dedicar seu tempo à leitura deste eBook. Desejo a você sucesso na sua jornada!

Marcelo Hadad
GitHub | LinkedIn