Versionamento de Código

Módulo 2

TIC em Trilhas

MÓDULO 2

Este módulo tem como objetivo compreender o funcionamento minucioso e detalhado de conceitos importantes do Git, operações complexas e estratégias de controle de qualidade de software a fim de se obter um conhecimento mais profundo da ferramenta.

CAPÍTULO 1

DESENVOLVIMENTO COLABORATIVO

Este capítulo mostra estratégias e ferramentas avançadas para melhorar a experiência do desenvolvimento colaborativo usando Git e GitHub.

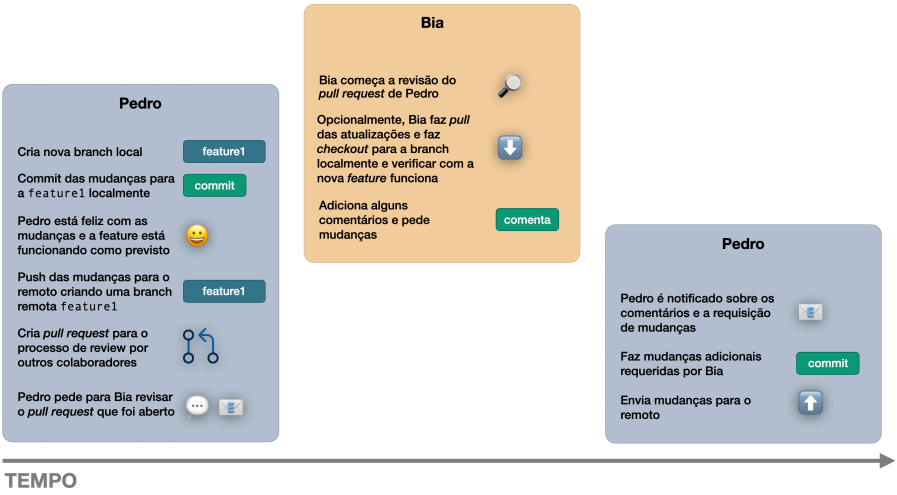
Desenvolver colaborativamente significa ter ferramentas que facilitem e agilizem a experiência em um projeto. Dentre estas ferramentas, é possível citar uma das mais úteis e utilizadas para incrementar o software, ou seja, para a criação de novas funcionalidades e correções de falhas chamada de *pull request*. Pull request nada mais é do que a validação de que algum desenvolvedor fez está certo ou segue as diretrizes de desenvolvimento estabelecidas para um projeto específico. Em outras palavras, quando um colaborador escrever código e quiser unificar suas mudanças na branch principal (a branch na qual o projeto funcional está localizada) ele deve subir seu código para o repositório remoto e solicitar que outro colaborador avalie seu código, executando ele, verificando possíveis erros, boas práticas e ao fim comentando se o que foi feito pode

ser realmente aprovado como envio para a branch principal ou se será necessário fazer mudanças no código por qualquer motivo comentado pelo(a) avaliador(a) (seja correção de falhas, melhoria de boas práticas, etc...).

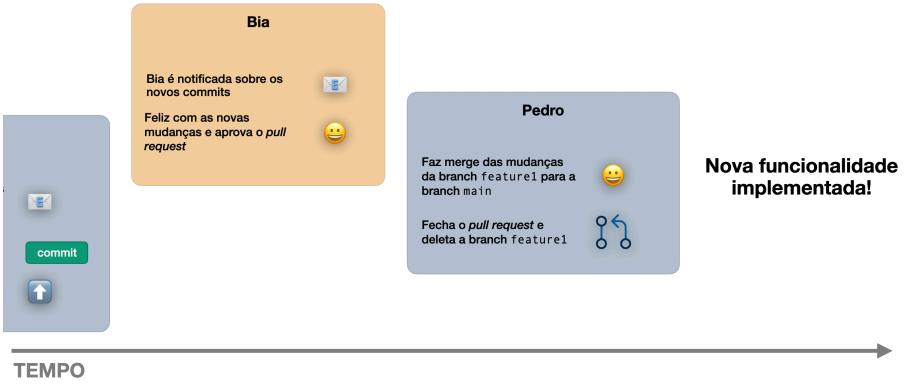
Para criar um *pull request* é bem simples. Supondo que um desenvolvedor escreveu código na branch `feature1` local e enviou suas alterações para a branch `feature1` remota. Agora ele quer unir as mudanças feitas nessa branch com a branch `main`. Para isso, ele deve ir na plataforma do GitHub, selecionar o repositório em que está trabalhando e clicar em '*Pull requests*'. Na página, basta clicar em 'Novo *pull request*' e selecionar qual branch será a fonte e qual será a destino. Ao criar um *pull request* o GitHub irá automaticamente configurar o ambiente para que todos os colaboradores do projeto possam realizar a avaliação do código enviado.

A fins de demonstração do processo de criação e aplicação de um *pull request*, suponhamos que existam dois colaboradores, Pedro e Bia, em um projeto em que estão no mesmo time desenvolvendo melhorias e novas funcionalidades para uma aplicação. Pedro acabou de desenvolver uma nova funcionalidade em uma branch chamada `feature1` e quer unir essas mudanças com a branch `main`. Pedro então cria um *pull request* no GitHub e pede para que Bia revise o seu *pull request* que foi aberto. Para que ela consiga revisar essa nova versão, ela pode analisar o código feito pela própria interface do GitHub e/ou entrar na branch `feature1` que foi criada pelo Pedro para puxar o código feito e testar em seu dispositivo. Feito

isso, Bia pode deixar seus comentários e críticas na plataforma do GitHub solicitando alterações de código para que Pedro conserte ou modifique. Se não houverem problemas para se resolver, Bia pode apenas comentar que o código já pode ser unificado na branch `main`.



Se houverem modificações, Pedro verá os comentários feitos por Bia, realizará as alterações solicitadas e enviará novamente as mudanças para a branch `feature1` do repositório remoto para nova avaliação. Se desta vez a Bia estiver satisfeita com as mudanças feitas, ela pode aprovar que as alterações sejam unidas na branch `main` e deletar a branch `feature1`, já que não será mais necessária. Desta forma o processo de *pull request* foi concluído e a nova funcionalidade foi implementada e está disponível na branch estável do projeto para que outros colaboradores possam puxar ela para seus repositórios locais e continuar o desenvolvimento a partir dela.



Em qualquer momento da criação do *pull request*, é possível que ocorram erros de merge, ou seja, não será possível fechar ou concluir o *pull request* devido a conflitos que possam aparecer entre a branch fonte e destino. Anteriormente, exemplos de resolução de conflitos foram vistos, entretanto, existem outros tipos de resolução de conflitos. Um conflito em um arquivo requer que o desenvolvedor escolha entre as mudanças atuais ou as que estão vindo de outra branch (*current changes* ou *incoming changes*), entretanto, é possível que ambas as mudanças podem ser apagadas e o desenvolvedor pode escrever o que desejar manualmente, ou seja, se ele achar necessário que um código customizado seja escrito no ato da resolução de um conflito em específico, basta apagar o padrão de escolha das mudanças e escrever manualmente o que quer que seja válido quando for realizar o merge. Pode ocorrer também que ambas as alterações precisem ficar, ou seja, o desenvolvedor pode escolher manter ambas as alterações (*both changes*). A escolha de quais alterações manter depende do contexto em que as novas funcionalidades estão inseridas, cabendo ao desenvolvedor

discernir e tomar a decisão sobre o que ficar e o que for removido enquanto estiver resolvendo os conflitos.

Ao concluir um pull request, usualmente a branch fonte é deletada. Para deletar uma branch remota via terminal basta digitar `git push origin -d <branch>`. Com o comando `git show-ref <branch>` é possível comparar branches locais e remotas, a fim de verificar quais conexões estão mantidas ou não com as branches locais. É possível deletar a branch localmente, entretanto, ela irá continuar conectada com a branch remota (*remote tracking branch*). Para resolver isso, é possível cortar essa ligação usando o comando `git remote update origin --prune` que irá podar as branches que não possuem mais conexão entre o remoto e o local.

CAPÍTULO 2

TÓPICOS AVANÇADOS DE GIT

Este capítulo mostra de forma teórica e prática os conceitos de *rebasing* e *squashing*.

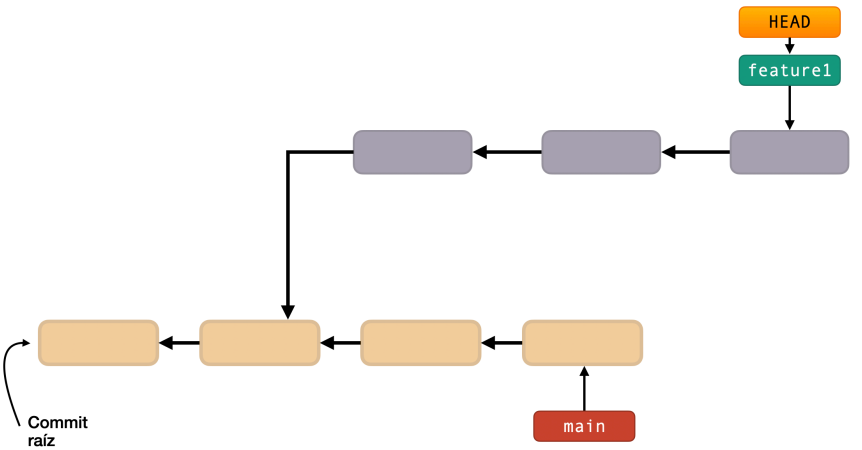
REBASING

A técnica de *rebasing* é uma maneira alternativa de se fazer merge de duas branches ou mais juntas. Existem vantagens e desvantagens de se usar esta abordagem. A vantagem é a de que *rebase* faz com que o histórico de commits fique linear, ou seja, com merge os commits podem ter mais de um pai (commits que originaram o novo commit) e com *rebase* cada commit possui apenas um pai. Já a desvantagem é a de que ele re-escreve o histórico de commits, ou seja, *rebasing* não mantém o histórico completo de todos os commits e, na verdade, alguns commits são perdidos durante o processo.

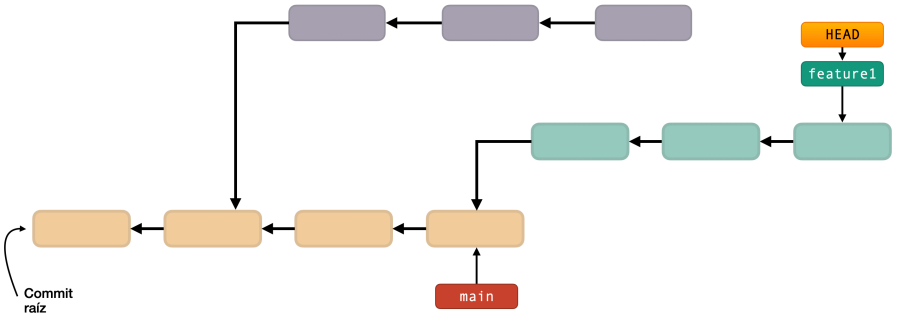
O processo de *rebasing* é feito em duas etapas. A primeira etapa consiste em fazer *checkout* para a branch fonte

(geralmente a branch que contém dados que vão ser enviados para uma branch principal) usando o comando `git checkout <branch fonte>`. Depois é preciso executar o comando `git rebase <branch destino>`. Este comando irá criar uma cópia de todo o histórico de commits na branch atual e criar uma conexão linear com a branch destino, ou seja, a branch que foi passada como parâmetro no comando de *rebase*. A segunda etapa consiste em fazer checkout para a branch destino com o comando `git checkout <branch destino>` e realizar o merge desta nova cópia dos commits da branch fonte sem múltiplos pais em cada commit com o comando `git merge <branch fonte>`.

Por exemplo, uma branch nova chamada de `feature1` foi criada a partir da branch `main` em algum ponto do histórico de commits. Foram realizados diversos commits na branch `feature1`. O estado atual do histórico de commits é representado pela figura a seguir.

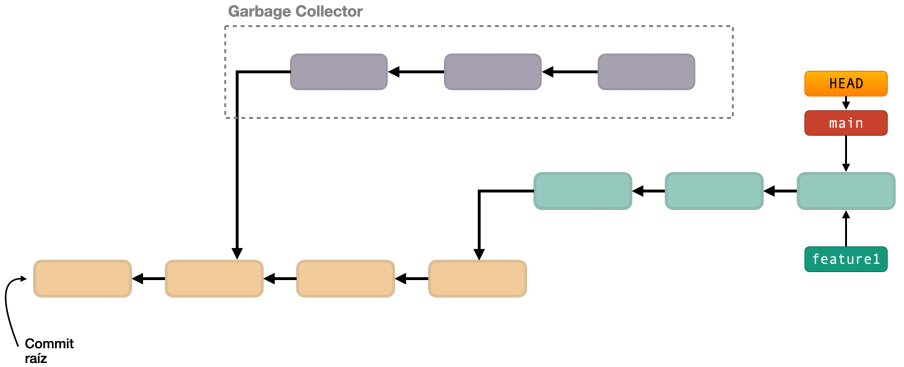


O ponteiro HEAD aponta para a branch `feature1` e nela existem alguns commits criados. A primeira etapa do processo de *rebasing*, neste caso, é fazer *checkout* para a branch destino (`feature1`) em que, dada a imagem, já está como branch atual, ou seja, a HEAD está apontando para ela. A seguir deve-se usar o comando `git rebase main` para que uma cópia dos commits da branch fonte sejam feitos automaticamente pelo Git e sejam alinhados de forma linear e conectados com o último commit da branch destino, como mostra a imagem abaixo.



Nota-se que a branch feature1 passa a apontar para a cópia do último commit feito nela. Vale ressaltar que todos os commits novos que foram copiados agora estão com um código SHA-hash diferente devido ao fato de que o conteúdo deles podem ser diferentes (um commit na linha do tempo podia ter dois pais, mas como a sequência do histórico de commits agora é linear, os pais foram removidos e o commit cópia possui apenas um pai). Por fim, ao digitar o comando `git checkout main` e fazer o merge da branch `feature1` cópia com o comando `git merge feature1` é possível perceber que o ponteiro HEAD

se deslocou para o último commit da branch `feature1`. Isso porque como foi feita uma cópia de todos os commits e eles foram colocados de forma sequencial a partir do último commit da branch destino, a estratégia de merge aplicada foi o fast-forward, ou seja, o último commit da sequência não terá dois pais e sim um.



A imagem acima demonstra o resultado final do processo hipotético de rebase entre duas branches. Nota-se que os commits originais da branch feature1 perderam a referência e não estão mais atrelados a nenhuma branch. Por isso, o Git possui um mecanismo de limpeza de memória, chamado de *Garbage Collector*, que ao longo do tempo irá deletar todos os commits inalcançáveis por qualquer método que não seja acessando via seu código SHA-hash.

Rebasing é muito utilizado no mercado como ferramenta para procurar falhas. Por exemplo, aonde na cadeia de commits feitos em uma branch começou a ocorrer uma falha. O *rebase* pode ser associado à ação de um comando que é executado quando o *rebase* opera, começando a

copiar commit a commit para a branch destino e aplicando esse comando em cada cópia (este comando pode ser qualquer comando que ajude a encontrar a falha na cadeia de commits). Existem diversas outras utilidades que, combinadas com o *rebasing*, auxiliam no desenvolvimento e automação do Git como, por exemplo:

- **Rodar testes:** Um dos usos mais frequentes da *flag* `--exec` na operação de *rebase* é o de executar testes em cada commit. Isso garante que cada um deles na sequencia do *rebase* passará no teste, auxiliando na identificação de possíveis problemas. Exemplo: `git rebase -i --exec 'npm test' <branch>`

- **Padronização de Código:** É possível rodar um padronizador de código (usualmente chamado no mercado de *linter*) para garantir que cada commit adere aos padrões de organização de código. Exemplo: `git rebase -i --exec 'eslint .' <branch>`

- **Checar sucessos de build:** É possível checar se cada commit no histórico compila. Exemplo: `git rebase -i --exec 'make build' <branch>`

- **Rodar scripts customizados:** Qualquer script ou comando que for preciso rodar com propósitos de validação pode ser usado, seja ele um script de migração de banco de dados ou até mesmo um script que verifica por um certo padrão no código. Exemplo: `git rebase -i --exec './validate-commit.sh' <branch>`

- **Gerar documentação:** Em alguns fluxos de trabalho pode acontecer de que seja necessário garantir que a documentação do código seja gerada sem erros para cada commit. Exemplo: `git rebase -i --exec 'generate-docs' <branch>`

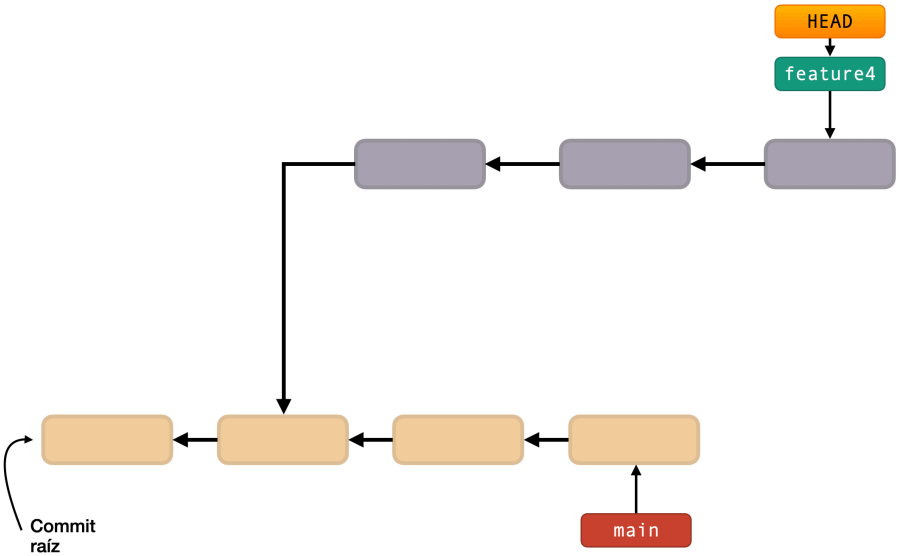
- **Rodar verificações de segurança:** É possível rodar comandos que verifiquem a segurança de cada commit criando um relatório para auxiliar na identificação de quaisquer vulnerabilidades introduzidas durante o desenvolvimento. Exemplo: `git rebase -i --exec 'npm audit' <branch>`

SQUASHING

A função da técnica de *squashing* é reduzir diversos commits em apenas um. Ela é útil em projetos grandes e com vários colaboradores e geralmente nos momentos em que se vai realizar um *pull request*. Um colaborador geralmente cria uma branch para desenvolver uma nova funcionalidade e lá dentro faz vários commits. Ao finalizar a funcionalidade, em alguns casos, todos os commits criados não interessam para as branches principais deste grande projeto como, por exemplo, `development` ou `release`, ou seja, não faz sentido que qualquer commit que tenha sido feito antes do último (que contém a funcionalidade completa) seja adicionado em uma branch principal. Por isso, ao invés de o desenvolvedor adicionar uma sequencia de vários commits de sua branch para a branch `development`, ele junta todos esses commits em apenas um e o unifica com a branch principal através de *rebasing*

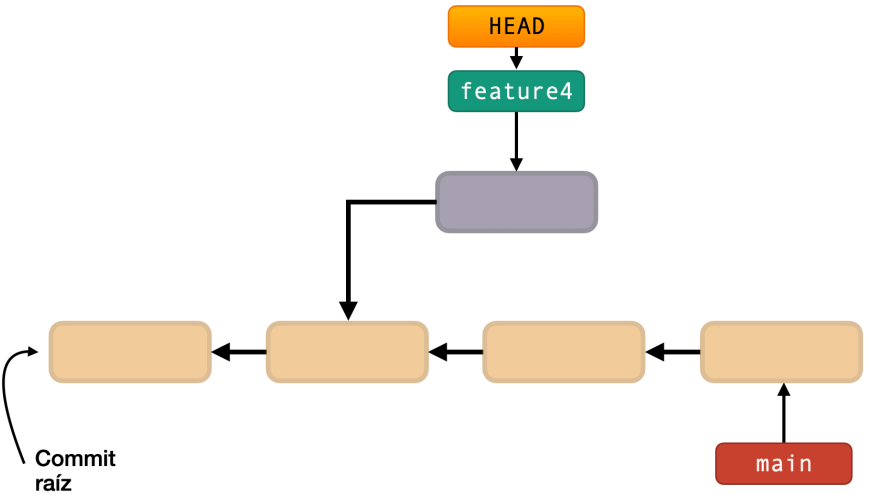
interativo com *squashing*. O que é *rebasing* interativo? O *rebasing* interativo é demarcado com a opção `-i` junto ao comando de *rebase*.

Considere o cenário hipotético a seguir:

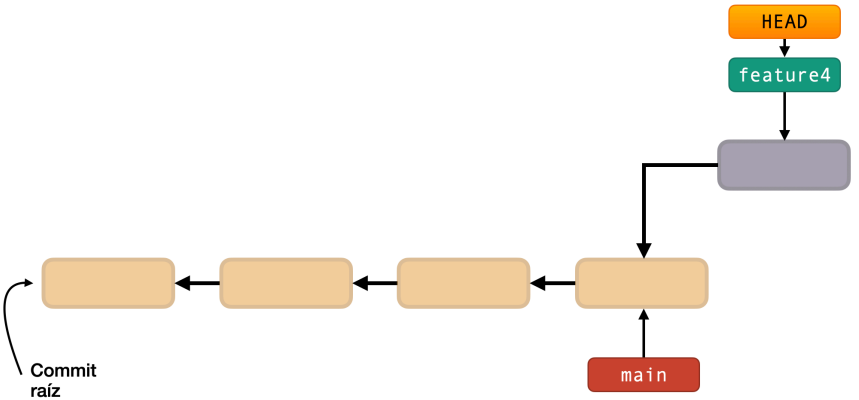


A branch `main` possui 4 commits e uma nova branch chamada de `feature4` foi criada a partir do segundo commit dela. O objetivo do cenário hipotético será comprimir todos os commits da branch `feature4` em apenas um commit e fazer com que este único commit seja o último commit da branch `main`. Com este intuito, primeiro é preciso converter os 3 últimos commits (poderiam ser os *n* últimos commits) em um. Para isso, basta digitar `git rebase -i HEAD~3` no terminal. O que este comando faz é iterar sobre todos os commits e aplicar

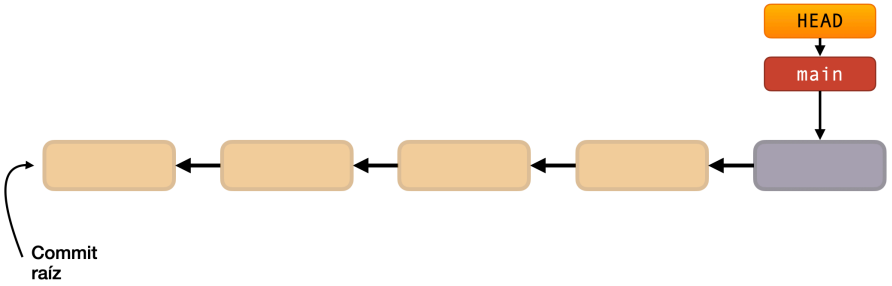
uma ação que será especificada pelo usuário. Ao digitar esse comando, um arquivo irá abrir contendo os commits em sequencia (e seus respectivos hashes e mensagens do commit). Na frente dos commits haverá a palavra "pick" escrita. O *squashing* acontece quando essa palavra for substituída por *squash*. Cabe ao desenvolvedor escrever ela no lugar da palavra "pick". Entretanto, todos os commits serão reduzidos em um, ou seja, é necessário que o primeiro commit da sequencia continue com a palavra "pick". Em resumo, todos os commits exceto o primeiro serão substituídos pela palavra "squash". Depois disso, é preciso salvar o arquivo. Na sequencia, um outro arquivo irá se abrir a fim de editar a mensagem do novo commit que será gerado a partir dos 3. Se não for necessário alterara a mensagem, basta salvar o arquivo. Com isso, o primeiro passo foi concluído e a situação atual do Git se encontrará da seguinte maneira:



O segundo passo é, de alguma maneira, enviar a branch `feature4` para o último commit da branch `main`, como se a branch tivesse sido criada a partir dele. Neste caso, será necessário aplicar o *rebasing* nessa branch fazendo *checkout* para a branch `feature4` e digitando o comando `git rebase main`. O estado futuro do histórico após o comando será o seguinte:



O terceiro e último passo para se atingir o objetivo especificado na criação do cenário hipotético é fazer o merge da branch `feature4` (branch fonte) na branch `main` (branch destino). Para isso, basta fazer *checkout* para a branch `main` com `git checkout main` e realizar a operação de merge digitando `git merge feature4` no terminal. O Git irá realizar o merge com estratégia *fast-forward* porque não existem commits a frente para que um novo commit de merge precise ser criado. O resultado final é o seguinte:



Note: a branch `feature4` pode ser removida, pois não será mais usada porque já foi unida à branch `main`.

CAPÍTULO 3

RESOLUÇÃO DE PROBLEMAS E RECUPERAÇÃO

Este capítulo explica o funcionamento da recuperação de erros feitos no Git usando *reset*, *revert* e *reflog*.

Este capítulo irá demonstrar e discutir sobre comandos destrutivos no Git. Comandos destrutivos são geralmente comandos que excluem/deletam informações. Neste caso, comandos destrutivos podem remover arquivos, commits, branches, etc. É preciso ter muito cuidado ao usar estes comandos e entender com clareza o que eles fazem para evitar possíveis perdas irreversíveis em projetos grandes ou pequenos. Antes de aprofundar no conteúdo, vale ressaltar que não é recomendado usar estes tipos de comandos em repositórios públicos e em branches principais como, por exemplo, branch `main`, `release` (nome geralmente dado a branches que possuem commits que significam versões estáveis do código que foram

lançadas como um produto de fato), `homolog` (nome geralmente dado a branches que estão a uma etapa de validação e teste para que seja movida a branch de release). Este conselho se deve ao fato de que ao deletar um commit, por exemplo, quando qualquer outro colaborador adicionado ao projeto no GitHub puxar as alterações da branch para seu repositório, este repositório pode entrar em um estado de inconsistência e/ou comportamentos não esperados em outros dispositivos. Portanto, é recomendado o uso desse tipo de comando apenas em branches em que um colaborador apenas está modificando ou qualquer repositório em que não existam outras pessoas trabalhando junto.

O primeiro comando deles é o `git reset`. Com ele é possível descartar mudanças que já foram commitadas. Por exemplo, foi feito um commit, entretanto, o desenvolvedor não está feliz com essas mudanças e quer descartar esse commit e voltar para o estado anterior no repositório onde esse commit não existia. Isso é possível com este comando. De fato, existem 3 opções de flags possíveis de serem usadas junto ao comando. São elas: `--hard`, `--mixed` (padrão) e `--soft`. Dependendo da opção usada, diferentes mudanças serão aplicadas no Working Directory, Staging Area e Git Repository. O comando que descarta commits no Git é `git reset [flags] <SHA-hash do commit>`. Este comando diz que todos os commits após o commit passado no comando *reset* sejam descartados. A função de cada opção está descrita a seguir:

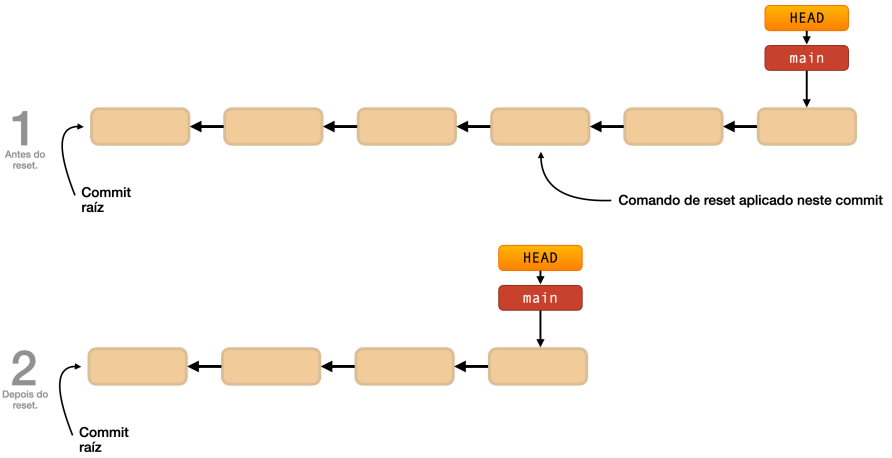
- `--mixed`: Esta opção é a opção padrão. Isso significa que se o desenvolvedor escolher não adicionar nenhuma flag junto ao comando de *reset*, essa flag está implícita no comando. Ao usar esta opção o Git descarta o commit, as mudanças na Staging Area e coloca no Working Directory todas as mudanças que foram feitas no commit descartado em relação ao anterior a ele. Por exemplo, se o commit A foi feito e logo após o commit B foi feito, todas as alterações do commit B serão levadas ao Working Directory e o commit B será removido. Esta é uma maneira segura de se usar o comando, pois as mudanças feitas ainda estão visíveis ao desenvolvedor, ou seja, supondo que ele se equivocou ao executar o comando, é possível acessar as mudanças ainda e salvá-las novamente no repositório Git.

- `--soft`: Esta opção é semelhante a opção *mixed*. Ela descarta os commits e mantém as mudanças no Working Directory. A única diferença é que ela mantém as mudanças também na Staging Area.

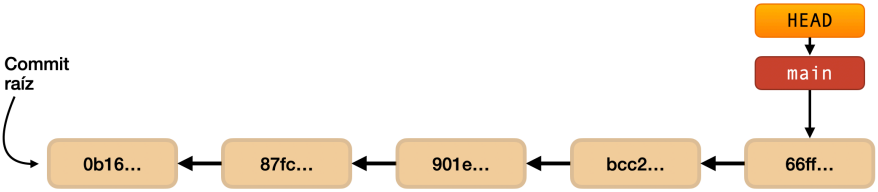
- `--hard`: Esta é a opção mais destrutível das 3. Com ela, os commits são descartados, as mudanças na Staging Area são descartadas e as mudanças no Working Directory também são descartadas.

É possível descartar um commit no meio da cadeia de commits. Isso fará com que todos os commits subsequentes a eles, ou seja, que vieram depois dele, sejam descartados também, fazendo com que o ponteiro da branch aponte para o commit a qual o comando *reset*

especificou. Também é possível descartar uma quantidade específica de commits para trás com o comando `git reset HEAD~<quantidade>`. Por exemplo: `git reset HEAD~5` descartará os 5 últimos commits da branch.

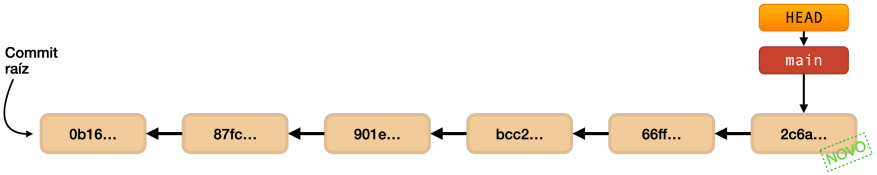


A operação de *revert*, ao contrário do *reset*, não é uma operação destrutiva e não modifica o histórico de commits do Git. Por isso, é seguro usar o comando em qualquer branch pública como, por exemplo, `main` ou `development`. A função da operação de *revert* é reverter apenas um commit, ou seja, seu papel é inverter o commit que foi passado como parâmetro no comando e criar um novo commit a partir dele só que revertido. Suponha a seguinte sequência de commits:



Suponha também que no commit `bcc2` haviam 3 arquivos chamados de `arquivo1.txt`, `arquivo2.txt` e `arquivo3.txt` respectivamente e que no commit `66ff` haviam 4 arquivos: os 3 arquivos que já existiam no commit `bcc2` e um arquivo novo criado e salvo nesse commit chamado de `arquivo4.txt`. O comando de *revert* no Git é `git revert <hash do commit>`. Ao digitar o comando `git revert HEAD` (HEAD está apontando para o commit `66ff`, ou seja, seria equivalente a digitar o comando `git revert 66ff`) o que o Git faz na prática é criar um novo commit com o inverso das mudanças feitas nele. Neste caso, não existia nenhuma mudança nem deleção de arquivo, somente a criação do arquivo `arquivo4.txt`. Isso significa que o que o comando fará é criar um novo commit sem `arquivo4.txt` porque isso é o que foi adicionado. Como o *revert* cria um commit com o inverso do que o commit tinha, o inverso do commit `66ff` é ele sem o `arquivo4.txt` que foi criado do commit `bcc2` para o commit `66ff`. Para fins de simplificação do entendimento, neste caso o que irá acontecer é que o novo commit criado será exatamente igual ao commit `bcc2`, entretanto, com um novo código SHA1-hash porque ele é um novo commit. Como visto anteriormente, código hash é gerado com um conjunto de informações que incluem hora de criação do commit, ou seja, mesmo que os arquivos sejam o mesmo

do commit `bcc2`, o código do novo commit não iria ser o mesmo por este motivo. O resultado da operação de *revert* nesse cenário hipotético é o seguinte:

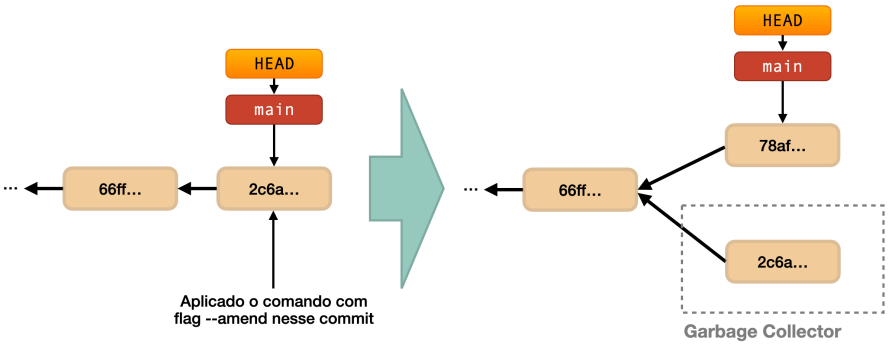


Um novo commit foi criado depois do commit `66ff` com o código `2c6a` e o ponteiro da branch foi deslocado para o último commit. Este comando é muito útil quando o colaborador já enviou suas mudanças para o repositório público em uma branch pública e outro colaborador já puxou essas mudanças para seu dispositivo. Neste caso, foi citado acima que usar o comando de *reset* nessas ocasiões não é apropriado devido a inconsistências e comportamentos inesperados que podem ocorrer. Por isso o comando revert existe. Ele basicamente é uma forma segura de reverter mudanças sem precisar deletar commits.

Note que é possível reverter commits anteriores ao último commit. Pegando o commit `901e` da imagem ilustrativa acima e aplicando o comando de revert nele, o resultado disso é a criação de um novo commit que será adicionado depois do commit `2c6a`. Entretanto, é provável que um conflito de merge aconteça entre o commit `901e` e `2c6a`. Para continuar a operação, basta resolver o conflito de merge, salvar as alterações, digitar o comando `git add .` para adicionar todas as mudanças decididas na resolução

dos conflitos e digitar o comando `git revert --continue`. Com isso, as mudanças de um commit anterior são revertidas a partir do último commit na branch para que qualquer colaborador possa ter acesso a esta nova versão no repositório.

Existe outra operação destrutiva que pode ser aplicada somente no último commit de uma branch. Esta operação é chamada de *amend*. Esta operação é basicamente uma flag posta no comando de commit para alterar informações do último commit como, por exemplo, mensagem ou autor. O que a operação faz é criar um novo commit a partir do commit editado e o commit editado é removido pelo Garbage Collector por não ter mais nenhuma referência apontado para ele. O comando usado no Git é `git commit --amend [parâmetro] [novas alterações]`.



No caso da imagem acima o comando `git commit --amend -m "Nova mensagem"` foi aplicado no último commit da branch e o resultado foi a criação de um novo commit com as alterações passadas por parâmetro no

comando. Percebe-se que agora existe uma nova ligação de parentesco no commit criado e o commit antigo será removido pelo Garbage Collector do Git.

Existe outro comando que opera sobre um único commit que se chama *cherry-pick*. O que este comando faz é simplesmente pegar um commit de qualquer lugar do histórico de commits e coloca em uma branch específica. A única condição é que este commit não pode ser um commit da própria branch na qual se quer colocar. Este comando é útil em diversos cenários. Por exemplo, suponha que um desenvolvedor está na branch `main` e existe a branch `feature8` com 5 commits feitos. Caso ele queira pegar o conteúdo do terceiro commit da branch `feature8` porque julga que ela tem um conteúdo importante, basta ele pegar o hash desse commit e aplicar o comando de *cherry-pick* nele. O comando de *cherry-pick* do Git é `git cherry-pick <commit>`. No caso de exemplo acima, o procedimento para colocar o commit na frente da branch é feito dado os seguintes passos: primeiro é preciso fazer *checkout* para a branch em que se quer colocar o único commit com `git checkout main`. Depois, basta digitar `git cherry-pick <hash do terceiro commit da branch feature8>` e o Git irá criar um novo commit na branch `main`, movendo o ponteiro da branch para ele.

É possível também usar o comando de cherry-pick sem que se tenha um novo commit criado. Para isso, basta usar a flag `--no-commit` junto do comando e todas as alterações serão puxadas para a Staging Area da branch destino. O comando de execução da operação completo é `git`

`cherry-pick --no-commit <commit>`. Esse comando pode ser útil caso o desenvolvedor queira criar o próprio commit com uma mensagem customizada. É possível ainda ver quais alterações foram trazidas do commit com o comando `git status -v` (-v significa *verbose*, ou seja, informações detalhadas). Este comando também é útil caso o estado da HEAD esteja em *detached* HEAD. Neste caso, ao invés do desenvolvedor precisar criar uma branch para manter o commit que não está sendo monitorado pelo Git, é possível apenas levá-lo até uma branch com o uso da operação de *cherry-pick*.

Por último, este capítulo irá explicar e demonstrar o uso de um comando importante do Git que lista o histórico de todas as operações feitas em um repositório local. O comando descrito é o `git reflog`. Este é um comando poderoso que auxilia o desenvolvedor a reverter estados do Git. Por exemplo, suponha que um colaborador em seu repositório local efetuou o comando `git reset --hard` em um commit. Ao digitar o comando `git reflog` é possível ver qual é o código SHA1-hash do último commit em que ele estava e que tinha sido perdido, pois não existe nenhuma branch mais que aponte para ele, ou seja, este commit seria perdido. Agora que é possível "lembrar" do commit que tinha sido removido, o colaborador pode usar o comando `git reset --hard` passando o hash do commit que tinha sido removido para que o histórico de commits volte a ser o que era antes. Importante: o histórico de operações listado pelo comando de *reflog* tem prazo de validade padrão de 90 dias pelo Git.

CAPÍTULO 4

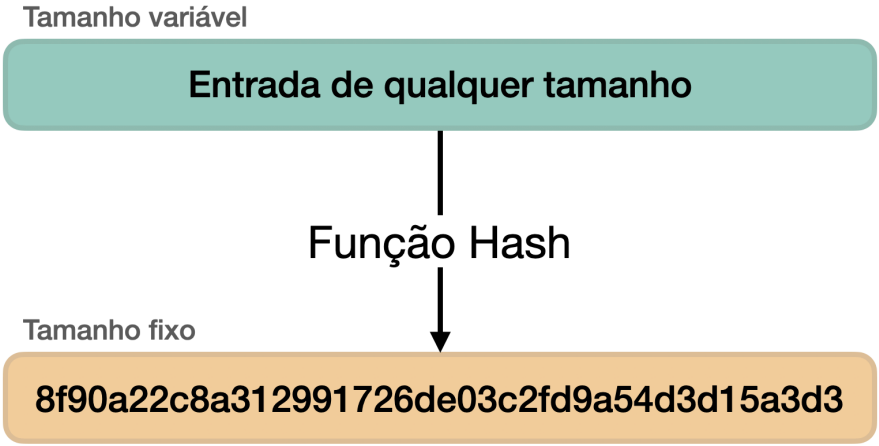
SHA1 HASH

Este capítulo descreve como é gerado o código SHA-1 hash que é atrelado aos tipos de arquivos do Git e responde perguntas pertinentes quanto ao seu limite.

O código hash é um cálculo matemático utilizado para criptografar um documento original. Existem vários métodos de criptografia. Um deles é o *Secure Hash Algorithms* (Algoritmos de Segurança Hash) ou mais comumente conhecidos como SHA. SHA são uma família de funções hash criptográficas. Dentre elas, é possível citar: SHA-1, SHA-256, SHA-512, etc. O Git usa o SHA1-hash para criptografar os arquivos e tipos de arquivos.

A criptografia acontece quando uma entrada de qualquer tamanho é dada para a função e ela retorna um código hash de tamanho fixo. O código SHA-1 hash possui 160 bits de tamanho, ou seja, 40 caracteres hexadecimal. Isso significa que o código gerado irá sempre ter 40 caracteres de tamanho (caracteres hexadecimais são: 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, A, B, C, D, E e F). O Git combina vários dados para

gerar o hash como, por exemplo, conteúdo do arquivo, data de criação, nome do autor, etc.



Como o código hash é fixo em 40 caracteres, perguntas relevantes surgem como, por exemplo:

- Quantos arquivos o Git pode armazenar em um mesmo repositório, visto que ele armazena códigos hash como arquivos, ou seja, se o tamanho do código é fixo ele tem um limite?

- Qual a chance de produzir exatamente o mesmo código hash para diferentes arquivos?

A combinação de dois bits é 2^2, ou seja, 4 combinações possíveis. Já a combinação de três bits é 2^3 que resulta em 8 combinações. SHA-1 possui 160 bits, ou seja, a combinação é de 2^{160} que resulta em 1461501637330902918203684832716283019655932542976

possíveis combinações (um quinquilhão...). Isso significa que a probabilidade dois códigos hash iguais é $\frac{1}{2^{160}} \times \frac{1}{2^{160}} = \frac{1}{2^{320}}$, ou seja, é extremamente improvável (quase impossível) que existam dois arquivos iguais e o limite de arquivos que podem ser armazenados é próximo deste número. O número de combinações possíveis do SHA-1 é tão grande que, se formos comparar ele com dados compreensíveis pelo ser humano, ele é maior do que a quantidade de grão de areia existentes no planeta terra (7.5×10^{18}), é maior do que a quantidade de estrelas no universo observável (10^{24}) ou do que a quantidade de segundos desde a explosão do Big Bang (4.35×10^{17}).

CAPÍTULO 5

.GITIGNORE

Um dos arquivos mais importantes do Git é o .gitignore. Com ele é possível ignorar arquivos desnecessários que seriam adicionados em toda mudança feita.

O *gitignore* é um arquivo especial do Git que ignora arquivos que irão ser salvos em uma versão de commit no repositório. Porque usar o `.gitignore`? O uso do *gitignore* é importante por várias razões:

- **Redução de *Clutter*:** Previne que arquivos desnecessários ou temporários sejam rastreados pelo Git. Isso inclui arquivos como *logs*, arquivos temporários criados por editores de texto ou sistemas operacionais, e arquivos de compilação.

- **Segurança:** Ajuda a evitar que arquivos sensíveis, como chaves privadas, senhas ou informações confidenciais, sejam acidentalmente adicionados ao repositório. Isso é crucial para manter a segurança e a privacidade dos dados.

- **Eficiência:** Reduz o tamanho do repositório e melhora a eficiência do Git. Arquivos e diretórios ignorados não são rastreados, o que significa menos dados para o Git processar, resultando em operações mais rápidas, como commit, *push*, *pull*, etc.

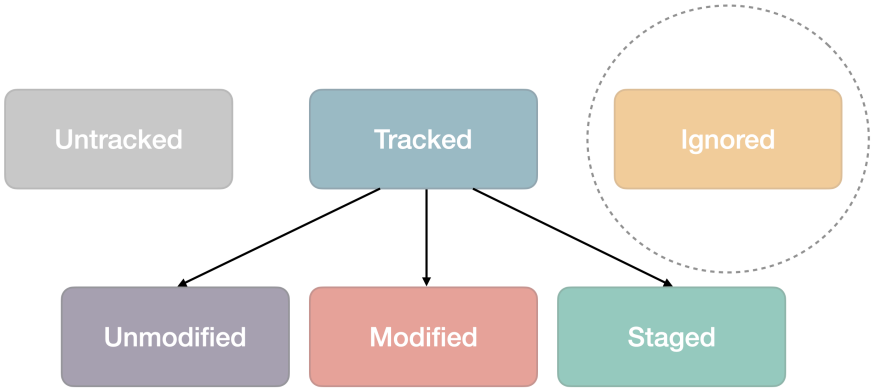
- **Controle e Limpeza:** Permite que os desenvolvedores tenham um maior controle sobre o que é incluído no repositório. Isso ajuda a manter o repositório limpo e organizado, contendo apenas os arquivos necessários para o projeto.

- **Personalização por Ambiente:** Diferentes ambientes de desenvolvimento ou diferentes desenvolvedores podem necessitar de regras de ignore distintas. O `.gitignore` pode ser configurado de forma a atender a essas necessidades específicas.

Existem arquivos específicos em alguns projetos que não precisam ser adicionados. Por exemplo, no desenvolvimento no sistema operacional macOS existe um arquivo chamado de `.DS_Store` que armazena informações personalizadas sobre as preferências de visualização de uma pasta, como a posição dos ícones, a escolha de uma imagem de fundo para a pasta, e outras configurações de visualização. Esse tipo de configuração não é importante para outros colaboradores, ou seja, cada colaborador tem suas preferências de visualização pessoais em seu dispositivo o que faz com que salvar esse tipo de arquivo não faça sentido, a não ser que outros colaboradores

queiram ter exatamente a mesma configuração de outro dispositivo.

O `.gitignore` é um arquivo que contém o nome dos arquivos ou as extensões de arquivos que devem ser ignorados. Geralmente em projetos grandes, existem diversos arquivos que podem ser ignorados dependendo das tecnologias utilizadas. Existe um site que gera automaticamente o texto de um arquivo `.gitignore` com as tecnologias que são descritas nesse site. O site é: https://www.toptal.com/developers/gitignore. Nele é possível ignorar arquivos de sistemas operacionais, IDEs ou linguagens de programação desnecessários. Por padrão, o Git não ignora nenhum arquivo.



Com o *gitignore*, um novo tipo de estado de arquivo foi adicionado a arquitetura de estados vista no início da trilha e é chamado de *Ignored* (arquivos ignorados).

Na prática, o desenvolvedor precisa adicionar o arquivo no projeto com as restrições de arquivos e diretórios que melhor lhe convém. Portanto, o primeiro passo após a criação de um repositório novo é o de adicionar o arquivo `.gitignore` e colocar dentro dele todas as regras de supressão necessárias. Depois, realizar o commit dessa mudança para que, a partir daí, o desenvolvimento possa começar a partir deste commit (o commit aonde foi salvo o arquivo `.gitignore`). Se o colaborador que criou o repositório estiver usando alguma plataforma de hospedagem como, por exemplo, o GitHub, ele deve fazer a operação de envio (*push*) deste commit para o repositório remoto a fim de que os outros colaboradores baixem para seus dispositivos (*clone* ou *pull*) essa alteração e possam iniciar seu desenvolvimento.

Há o caso em que um projeto já começou e já foram feitas novas funcionalidades em um projeto, ou seja, o arquivo `.gitignore` não foi adicionado no começo e será adicionado no meio do projeto em andamento. Neste caso, o que se quer fazer é ignorar arquivos que já foram commitados. Existem algumas opções para lidar com esse caso que é comumente esperado e cabe ao desenvolvedor escolher a que lhe melhor se adapta ao cenário em que ele está enfrentando no momento. São elas:

- **Opção 1**:

    1. Adicionar a nova regra ao `.gitignore`.

    2. Deletar o arquivo no Working Directory.

3. Fazer o commit das mudanças (neste caso o commit terá a nova regra adicionada no `.gitignore`).

4. Adicionar novamente o arquivo que foi anteriormente deletado.

- **Opção 2**:

1. Adicionar a nova regra ao `.gitignore`.

2. Deletar arquivo apenas do repositório, mantendo ele no Working Directory usando o comando `git rm --cached <arquivo>`.

3. Fazer o commit das mudanças.

Existem diversos tipos de arquivos que podem ser ignorados. Um exemplo foi citado acima. Entretanto, é uma boa prática ignorar vários outros tipos de arquivos como, por exemplo:

- Arquivos com tamanho muito grande (podem consumir muita memória do repositório remoto, pois salva *snapshots* de cada mudança no histórico, ou seja, se um arquivo de 1 GB foi commitado, alterado e commitado novamente, o Git irá ter salvo 2 GB em seu histórico, pois quando um arquivo é modificado um novo hash é gerado para ele).

- Diretórios de *build* como, por exemplo, `bin/`.

- Diretórios de dependências como, por exemplo, `node_modules/`.

- Arquivos compiladores e arquivos de histórico como, por exemplo, `*.pyc`, `*.log`.

- Arquivos do sistema operacional ocultos como, por exemplo, `Thumbs.db` ou `.DS_Store`.

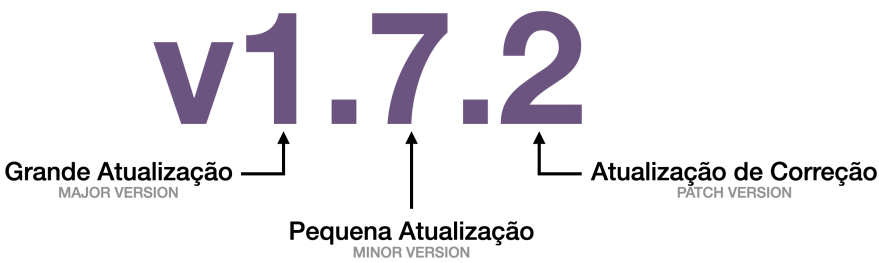
CAPÍTULO 6

TAGS NO GIT

Tags são um jeito elegante de se armazenar versões importantes de commits no repositório para fácil acesso ou usando-as para automatizar processos.

Uma tag é basicamente um ponteiro textual estático que aponta para um commit específico. Branches são ponteiros textuais dinâmicos, ou seja, se ao longo do tempo um novo commit for feito em uma branch, o seu ponteiro irá se mover e apontará automaticamente para esse novo commit. Já uma tag é estática, ou seja, não importa o que aconteça ela continuará apontando para um commit específico. Com isso, é possível voltar para versões usando tags e, por consequência, causar o estado de *detached head*.

Tags são usadas primariamente para marcar versões de lançamento (comumente chamada de *release version*). A semântica (ou significado) do versionamento pode ser encontrada com mais detalhes no site https://semver.org/. Abaixo, encontra-se uma imagem com o significado geral e simplificado dos números de uma versão:



Tags também podem ser usadas para criar referências memoráveis para certos commits ao invés de apenas lembrar do código SHA1 hash do commit, para capturar o estado atual de um projeto em relação a um evento significante, demarcar diferentes estágios (*alpha*, *beta*, candidato a *release*), entre outros. Na linha de integração contínua e implantação contínua (*Continuous Integration* e *Continuous Deployment* respectivamente, comumente mencionado como CI/CD), tags podem ser usadas como gatilhos para processos de implantação. Isso permite fluxos de implantação automatizados para novas versões (*releases*) baseado na presença de uma nova tag.

Existem dois tipos de tags no Git:

- *Lightweight* (leve): tags leves são armazenadas apenas no diretório `.git/refs/tags`. Elas servem apenas para demarcar um commit no histórico de commits sem se importar com o tempo em que ela foi criada nem com o autor que criou ela. O comando usado para criar uma tag leve é `git tag <nome da tag>`. Exemplo de uso: `git tag v1.0.0`.

- *Annotated* (anotada): este tipo de tag é armazenada no diretório `.git/refs/tags` e também no diretório `.git/objects` porque, além de guardar a referência de um commit, ela também guarda a data de criação da tag, o nome do autor que criou a tag e uma mensagem atrelada a tag, ou seja, uma mensagem extra, além do nome que foi dado a ela. Uma tag anotada é também um objeto do Git. Como visto anteriormente, o Git armazena quatro tipos de objetos: *blob*, *tree*, commit e *annotated tag*. O comando usado para criar uma tag anotada é `git tag -a <nome> -m <mensagem>`, onde `-a` denota que a tag é anotada e `-m` denota a mensagem que vai ser atrelada a tag. Exemplo de uso: `git tag -a v1.0.0 -m "Nova tag”`.

Ao criar uma tag com o nome de, por exemplo, v1.0.0 o Git cria um arquivo com o mesmo nome no diretório de tags. Digitando o comando `cat .git/refs/tags/v1.0.0` é possível verificar o conteúdo dentro do arquivo criado e nota-se que o que aparece é o código SHA-hash do commit para qual a tag está apontando. Desta forma, é possível compreender o funcionamento do armazenamento de cada ponteiro de tag para o seu respectivo commit. Nota-se que ao digitar o comando `git tag -v v1.0.0` (comando usado para verificar informações sobre uma tag) o Git não reconhece o nome dessa tag como um tipo de objeto que é armazenado pelo git, ou seja, se ela não pode ser listada é porque ela é uma tag leve ou *lightweight*. Assim como não é permitido criar branches com o mesmo nome, não é possível criar tags de mesmo nome. Vale também lembrar que tags não são enviadas para o repositório remoto com o

comando `git push` , para isso é preciso adicionar o parâmetro `--tags` junto do comando (`git push --tags`). Para enviar apenas uma tag das possíveis criadas localmente é preciso especificar qual a tag que irá para o repositório remoto usando o comando `git push <repositório> <tag>` no terminal. Por exemplo: `git push origin v1.1.0`. Quando tags são enviadas para o remoto, commits não são, ou seja, é preciso digitar o comando `git push` para enviar os commits feitos para que eles também sejam salvos no GitHub.

Já ao criar uma tag anotada com o nome v1.1.0, percebe-se ao digitar o comando `cat .git/refs/tags/v1.1.0` que um outro código SHA-hash é criado, entretanto, não se refere ao hash do commit para qual a tag está apontando e sim para o objeto *annotated tag* que foi criado pelo Git e armazenado no repositório.

Para listar todas as tags disponíveis, basta digitar o comando `git tag`. É possível, ao escolher uma tag da lista (se houver sido criada alguma), verificar suas informações digitando o comando `git show <tag>`. É possível também remover uma tag do Git usando o comando `git tag -d <nome da tag>`. Já para removê-la do seu respectivo repositório remoto (se estiver sido enviada para lá), basta digitar o comando `git push --delete <repositório> <nome da tag>`.

CAPÍTULO 7

STASHING

Um comando "quebra-galho" conhecido no Git é o *stashing*. Com ele é possível guardar mudanças feitas no Working Directory para usá-las e/ou combiná-las em outro momento.

Enquanto o desenvolvedor tiver mudanças feitas em uma branch no Working Directory, não é possível ir para outra branch, ou seja, não é possível usar o comando `git checkout <branch ou commit>`. É aí que o comando de *stashing* do Git entra em ação. Esta operação permite que o desenvolvedor consiga guardar as alterações feitas em uma branch em um lugar especial para que o Working Directory fique limpo e a operação de *checkout* seja permitida. Em outras palavras, *stashing* permite salvar alterações não commitadas.

Para aplicar o comando de stashing na prática é preciso criar uma branch que pode ser chamada de, por exemplo, `temp` e realizar algumas mudanças em arquivos nesta branch. O Git agora sabe que existe trabalho modificado que precisa ser salvo para que as alterações não sejam perdidas. Entretanto, o desenvolvedor quer ir para outra

branch chamada de `feature3` para pegar um pedaço de código que só existe lá e adicioná-lo na branch `temp` para usar esse pedaço de código junto da branch. O problema é que o desenvolvedor só pode ir para esta branch com *checkout* se o Working Directory estiver limpo ou vazio, mas neste momento, ele possui alterações. Portanto, é preciso usar o comando `git stash` para que essas alterações sejam temporariamente salvas em outro local que não é o Working Directory. Com o comando aplicado, percebe-se que a branch voltou ao estado original, ou seja, o estado em que não existiam mudanças feitas nos arquivos do último commit da branch. Aonde foram parar essas mudanças? As mudanças foram salvas na pasta `.git/refs/stash`. Para verificar que elas foram parar nesta pasta basta digitar `cat .git/refs/stash` e a lista de objetos salvos em stash aparecerão listados por este comando. Na verdade, uma lista de códigos SHA1-hash será exibido. Cada código, basicamente, representa um commit temporário que o Git cria para as alterações feitas e salvas pelo *stash*. Isso acontece porque a partir daí o Git consegue, através do commit temporário criado, voltar as mudanças que haviam sido feitas. Depois de fazer *stash* das mudanças o desenvolvedor pode fazer *checkout* sem problemas. Para voltar com as mudanças feitas novamente na branch `temp` que foram salvas no *stash*, basta digitar `git stash pop`.

É possível que o desenvolvedor queira fazer *stash* na branch `temp`, ir para a branch `feature3` e voltar para a branch `temp` trazendo pedaços de código. Neste caso, a branch `temp` estará com arquivos em estado de modificado. Isso

significa que pode acontecer que, quando o desenvolvedor pegar as mudanças que tinham sido salvas no *stash* com `git stash pop`, ocorram conflitos nos arquivos. A partir disso, é só resolver os conflitos e continuar o desenvolvimento normalmente.

Note que, ao utilizar o comando `git stash pop`, o Git trás as alterações que haviam sido salvas, entretanto, deleta o commit temporário que tinha sido criado para essas mudanças. Existe uma maneira de trazer as alterações sem que o commit temporário seja deletado. Isso é possível com o comando `git stash apply`. O benefício deste comando está relacionada a possibilidade de experimentar algo, descartar o que foi experimentado e puxar novamente as mudanças do *stash*, caso o desenvolvedor não tenha gostado de suas experimentações. É possível também deletar o commit mais recente feito no *stash* com o comando `git stash drop`. Outro comando útil é a deleção de apenas um commit específico do *stash*. Para isso, primeiro é preciso listar os commits com `git stash list` e escolher o commit que se quer remover. Depois é só pegar o hash do commit e digitar `git stash drop <hash>` para removê-lo. Quando se terminar de trabalhar um projeto e o desenvolvedor quiser deletar todos os commits do *stash*, basta digitar o seguinte comando: `git stash clear`.

Por fim, em um caso muito específico, se o desenvolvedor quiser restaurar um commit temporário do *stash* que foi deletado, é preciso encontrar o hash dele e digitar o comando `git stash apply <hash>`.

CAPÍTULO 8

CHAVES SSH

O GitHub permite que o usuário se autentique de diversas maneiras. Além da maneira padrão com usuário e senha, é possível cadastrar uma chave SSH de autenticação.

O GitHub permite que o usuário se autentique de diversas maneiras. A mais comum, rápido e fácil delas é a autenticação com usuário e senha ou usuário e *token* de autenticação padrão. Entretanto, existe outra maneira de autenticação chamada de autenticação via SSH. Esse tipo de autenticação carrega consigo mais benefícios (se comparado com autenticação usuário e senha) por diversos motivos, entretanto, fazer a configuração do SSH no sistema pode levar mais tempo e esforço se comparado ao método de utilizar usuário e senha. Dentre os benefícios de se usar o SSH, é possível citar:

- **Segurança:** chaves SSH são mais seguras do que senhas. As chaves geradas são chaves criptografadas que são quase impossíveis de se decifrar usando força bruta. Senhas podem ser facilmente descobertas ou roubadas, especialmente se ela é fácil ou reusada em diversos dispositivos.

- **Conveniência:** uma vez a chave SSH configurada, não é necessário digitar o usuário e senha toda vez que uma interação com o repositório for feita.

- **Controle de acesso:** chaves SSH podem ser facilmente adicionadas ou removidas sem afetar outras credenciais.

- **Resistência a Phishing:** Phishing é uma estratégia de capturar pacotes de dados que são trafegados de um dispositivo com a internet. O usuário não precisa digitar a senha com método de autenticação com a chave SSH, ou seja, a senha do usuário não trafega pela internet e não pode ser vista no dispositivo.

Para configurar a chave SSH, uma série de passos deve ser seguida:

1. `ssh-keygen -t ed25519 -C "<email da conta GitHub>`
    1. Enter file in which to save the key [APERTAR ENTER]
    2. Enter passphrase (empty for no passphrase): [APERTAR ENTER]
    3. Confirm passphrase [APERTAR ENTER]
2. `eval "$(ssh-agent -s)"`
3. Tentar passo 4 antes, se não existir arquivo então digitar: `touch ~/.ssh/config`
4. `open ~/.ssh/config`
5. Digitar isso dentro do arquivo:

```
Host *.github.com
 AddKeysToAgent yes
 UseKeychain yes
 IdentityFile ~/.ssh/id_ed25519
```

6. Salvar o arquivo.
7. `ssh-add --apple-use-keychain ~/.ssh/id_ed25519`
8. Copiar o conteúdo de id_ed25519.pub pro clipboard: `pbcopy < ~/.ssh/id_ed25519.pub`
9. Acessar GitHub > Clicar em ícone do perfil no canto superior direito > Settings > SSH and GPG keys > New SSH key > Colar chave no campo de chave

Chaves SSH são boas práticas e é recomendado interagir com o GitHub se autenticando por ela devido aos seus benefícios em segurança e conveniência.

Muitos dos conteúdos visto neste módulo são essenciais para uma compreensão mais profunda do funcionamento das ferramentas de versão de código. Algumas das operações que são feitas requerem um conhecimento mais abrangente de seu funcionamento para evitar comportamentos inesperados. Git é uma ferramenta poderosa que pode ser usada de forma complexa para colaborar de forma performática em times de desenvolvimento.