

Versionamento de Código

Módulo 1



TIC em Trilhas

MÓDULO 1

Este módulo tem como objetivo te ajudar a entender o ambiente e fluxo de ferramentas de controle de versão de código e aprender na prática o uso de comandos básicos que operam sobre diferentes versões de código e colaborar com outras pessoas para incrementar o código de forma distribuída.

CAPÍTULO 1

INTRODUÇÃO

Este capítulo mostra o que é o controle de versão e quais ferramentas e frameworks estão disponíveis para uso no mercado.

O controle de versão ou versionamento de código é a prática de gerenciar, rastrear e monitorar alterações de arquivos, pastas, imagens, vídeos e documentos em projetos individuais ou colaborativos. Sistemas de controle de versão são ferramentas de software que auxiliam times de desenvolvimento a gerenciar mudanças no código-fonte de projetos ao longo do tempo. A prática do controle de versão permite os times a aumentar a produtividade e qualidade dos projetos porque é possível acessar diferentes versões de um mesmo projeto, ou seja, é possível colaborar usando a versão do projeto de outros colaboradores. Por exemplo, um colaborador que foi adicionado no projeto implementa uma função de busca de lançamento de filmes na internet. Outro colaborador está implementando a funcionalidade de listar filmes na tela do usuário, entretanto, ele precisa ter a lista de filmes atualizadas para poder mostrar na tela. É aí que entra o controle de versão. O colaborador implementando a lista de filmes consegue

unificar as mudanças das funções de buscar lançamentos de filmes da internet com a sua versão de implementação da tela e utilizar a função feita por outro colaborador em sua versão de desenvolvimento.

O controle de versão mantém o registro de cada modificação dos arquivos em um banco de dados que possui uma arquitetura de armazenamento especial e que será explicada nos capítulos seguintes.

Dentre as ferramentas disponíveis no mercado para a prática, é possível citar:

- Git;
- CVS (Concurrent Version System);
- SVN (Apache Subversion);
- Mercurial;
- Monotone;
- Bazaar;
- TFS (Team Foundation Server);
- VSTS (Visual Studio Team Services);
- Perforce Helix Core;

- IBM Rational ClearCase;
- Revision Control System;
- VSS (Visual SourceSafe).

A ferramenta mais utilizada dentre as citadas acima é o Git. A fim de estudo teórico e demonstrações práticas dos conceitos do controle de versionamento de código, a ferramenta que será utilizada como base das explicações dos próximos capítulos será o Git.

PLATAFORMAS DE HOSPEDAGEM

O Git pode ser utilizado de forma offline, ou seja, é possível monitorar os arquivos de um projeto em um único dispositivo. Entretanto, o Git é comumente utilizado em projetos colaborativos, ou seja, projetos em que outras pessoas possam trabalhar e compartilhar os mesmos arquivos.

- GitHub;
- BitBucket;
- SourceForge;
- TaraVault;
- Dogs;

- GitBucket;
- AWS CodeCommit;
- Beanstalk;
- Phabricator;
- Gitea;
- Allura.

A plataforma que será utilizada para demonstrar e explicar o funcionamento de plataformas de hospedagem de código-fonte usando o Git é o GitHub.

DIFERENÇA ENTRE GIT E GITHUB

Git é uma ferramenta de versionamento de arquivos, ou seja, ela permite que o usuário mantenha o monitoramento destes arquivos de forma que seja possível manter versões diferentes dos arquivos. O GitHub é uma plataforma de hospedagem de código-fonte e arquivos utilizando o Git como base do controle de versão. Ele permite que vários usuários mantenham o progresso e as diferentes versões de projetos em diferentes dispositivos e que podem ser unidos em um único lugar hospedado remotamente para que todos mantenham sincronizadas as mesmas versões dos projetos e/ou arquivos.

CAPÍTULO 2

INSTALAÇÃO DAS DEPENDÊNCIAS

Para demonstrar praticamente os conceitos que serão ensinados é necessário instalar as ferramentas que serão utilizadas no módulo.

INSTALAÇÃO DO GIT

O Git será uma ferramenta essencial para a compreensão teórica e prática dos conteúdos que forem expostos neste documento. Para instalar o Git basta acessar o site <https://git-scm.com> e, ao carregar, clicar em 'Downloads'. Em seguida, já na página de downloads, escolha seu sistema operacional (macOS, Windows ou Linux/Unix) e clique no seu respectivo link de download.

No sistema operacional macOS, é preciso ter instalado o programa Homebrew no dispositivo para instalar o Git via terminal. Para isso, acesse o site <https://brew.sh/> e, ao carregar, copie o comando de instalação que o site sugere, abra um terminal de comando e execute o comando

copiado no terminal. Ao concluir a instalação do Homebrew os próximos passos da instalação no site do Git podem continuar. Ao finalizar a instalação do Git, é possível verificar se a instalação foi feita com sucesso executando o comando `git --version` no terminal. Em caso positivo é possível ver uma mensagem exibindo a versão atual do Git instalada no dispositivo.

INSTALAÇÃO DO VISUAL STUDIO CODE

Visual Studio Code é uma Integrated Development Environment (IDE) ou Ambiente Integrado de Desenvolvimento e editor de código focada no desenvolvimento de código-fonte. Ele permite a vasta modificação da interface e instalação de plugins que auxiliam o programador em vários aspectos do desenvolvimento.

Para instalar o Visual Studio Code basta acessar o site <https://code.visualstudio.com/> e realizar o download da última versão (versão mais estável) do programa clicando no botão 'Download' na página principal. Ao concluir o download, basta abrir o arquivo baixado para executar o programa.

CAPÍTULO 3

FUNCIONAMENTO DO GIT

O Git por debaixo dos panos possui conceitos muito valiosos que são importantes para a compreensão das operações básicas.

O Git utiliza o conceito de repositório para armazenar tudo relacionado a um projeto (arquivos, pastas, etc.). Um repositório nada mais é do que um local para armazenamento de todas as informações de um projeto. É possível comparar um repositório a um banco de dados, onde todos os dados armazenados possuem uma lógica de armazenamento, inserção, leitura, atualização e deleção neste banco de dados. Para explicar o funcionamento do Git é preciso primeiro criar um repositório novo. Para isso, basta seguir os seguintes passos:

1. Criar uma nova pasta (o repositório e o projeto irão residir nesta pasta).
2. Abrir o terminal e caminhar até esta pasta.

3. Digitar o comando `git init` no terminal para inicializar toda a estrutura de um novo repositório Git.
4. Neste momento o repositório deve ter sido criado. Para verificar isto basta digitar o comando `ls -la` no terminal e verificar se um arquivo ".git" foi criado.

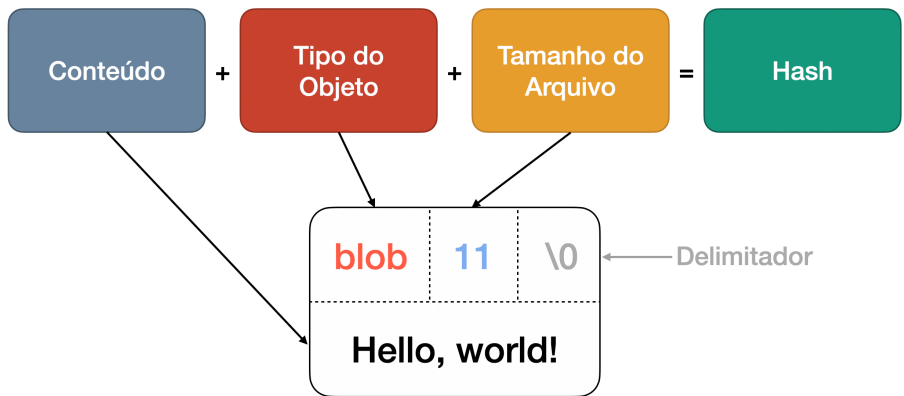
Ao explorar o arquivo ".git" através do comando `cd .git`, percebe-se que o programa gerou uma estrutura de arquivos para gerenciar o repositório criado. Dentre eles, existe uma pasta que armazena todos os arquivos de um projeto e as várias versões que ele possa ter. O Git armazena quatro tipos de objetos nesta pasta:

- **Blob:** armazena arquivos de qualquer extensão (vídeo, imagens, texto).
- **Tree:** armazena informação relacionada a diretórios. Um diretório pode ser vazio, conter outros diretórios ou vários arquivos dentro dele (ex: conjunto de blobs).
- **Commit:** armazena diferentes versões do projeto.
- **Annotated Tag:** é um ponteiro de texto que aponta (persiste) para um commit.



O comando `git init` fez com que o Git passasse a gerenciar a pasta em que a pasta ".git" se encontra.

Todos esses objetos são armazenados em um repositório do Git e todo arquivo armazenado é um código único gerado através deste arquivo chamado de *SHA hash*. Estes códigos são gerados sempre que o usuário fizer o commit de mudanças que forem feitas no repositório. Cada tipo de objeto possui uma lógica própria para resultar seu conteúdo em um código *SHA hash* único. Por exemplo, um arquivo chamado de `file1.txt` é criado no projeto e o conteúdo dentro do arquivo de texto é uma frase que diz "Hello, world!". Quando ele for salvo no repositório do Git, a lógica de criação do código *hash* é a junção das seguintes informações: conteúdo + tipo do objeto + tamanho do arquivo.



Para facilitar a compreensão e abstrair o objeto criado para cada tipo de objeto, abaixo estão dispostos exemplos de cada tipo de objeto criado representado por uma imagem:

8a31fb Blob

Hello, world!

3b95df Tree

file1.txt file2.txt

b7aec5 Commit

Nome e email do autor
Descrição do commit
Hash do pai
3b95df

Os 4 tipos de objetos são armazenados no repositório e são usados para que seja possível fazer a manipulação dos arquivos e versões ao longo do tempo no repositório. Para isso, existem 3 áreas de trabalho do Git:

- **Working Directory:** os arquivos são exibidos na pasta principal do projeto para serem usados, editados ou executados. Este ambiente pode conter apenas uma versão do projeto em qualquer momento do tempo.
- **Staging Area:** este é o local de transição obrigatório para mover arquivos do Working Directory para o Git Repository ou vice-versa. Este local permite que as mudanças possam ser agrupadas de forma significativa antes de serem commitadas (é chamado de commit a operação de registrar as alterações feitas pelo usuário no repositório para que elas fiquem salvas).
- **Git Repository:** local usado para armazenar os arquivos criados. O responsável pela gerência destes arquivos é o Git. Existe o repositório local e o remoto. O repositório local é o repositório que armazena as informações do projeto no dispositivo de um usuário. Já o remoto é um repositório que também armazena informações do

projeto, entretanto, pode ser acessado por outros colaboradores no mesmo projeto.



O fluxo de transição de arquivos entre as áreas do ambiente acontecem da seguinte maneira: o colaborador do projeto pode pegar alguma versão do projeto (por exemplo, a última) e continuar o desenvolvimento dele a partir dessa versão. Para pegar esta versão, o Git trás todos os arquivos relacionados a ela e os move para o *Working Directory*. Por que? Porque é no diretório de trabalho que os arquivos ficam visíveis ao desenvolvedor/colaborador e prontos para serem editados. Quando esses arquivos forem editados ou novos forem criados é preciso salvá-los no repositório local para que essa versão com melhorias e/ou coisas novas esteja segura e pronta para que outro colaborador acesse ou seja possível voltar nessa versão quando preciso. Então é preciso preparar os arquivos para enviá-los ao repositório. É aí que entra a *Staging Area*. Os arquivos que já existiam

no repositório, mas foram modificados precisam ser preparados para serem adicionados e os arquivos que foram criados não existiam no repositório, ou seja, precisam também serem adicionados a área de preparação a fim de que possam ser monitorados pelo sistema do Git. Quando todos os arquivos modificados estiverem preparados e prontos para envio ao repositório, basta realizar a operação de commit. Outro ponto importante, e que não havia sido mencionado, é o de que existe a transição de arquivos da *Staging Area* para o *Working Directory*. Para que essa transição ocorra é só modificar ou deletar algum arquivo, porque quando for alterado, ele não está mais pronto ou preparado para o envio, ou seja, ele foi movido novamente para o *Working Directory* para que as mudanças continuem ocorrendo e precisa novamente ser adicionado à *Staging Area*. É desta forma que se torna possível transicionar os arquivos entre todas as áreas do ambiente. A única transição não mencionada é a do repositório para a área de preparação. Já que todos os arquivos que estão salvos não precisam ser preparados, não faz sentido dizer que eles podem ser movidos do repositório para a área de preparação.

COMANDOS BÁSICOS DO TERMINAL (BASH SHELL)

A compreensão da manipulação de arquivos e pastas pelo terminal é de suma importância para realizar operações com o Git. Existem diversos comandos de terminal para operar sobre os diretórios, entretanto, serão elencados os

principais comandos e suas funcionalidades na tabela abaixo.

Comando	Descrição da Funcionalidade
mkdir	Cria uma pasta/diretório.
touch	Cria um arquivo.
nano	Edita um arquivo.
clear	Limpa o terminal (conteúdo e textos).
echo	Printa/exibe no terminal.
man	Ajuda em um comando específico.
cat	Lista os conteúdos do arquivo.
rm	Remove arquivos e diretórios.
ls	Lista todos os arquivos e diretórios dentro da pasta corrente.
ls -la	Lista arquivos ocultos também (-a) e de forma tabular (-l).
cd	Navega entre diretórios (entra ou sai de diretórios).
.	Abreviação para o diretório corrente.
..	Abreviação para o diretório pai ou anterior.
>	Escreve para um arquivo.
>>	Anexa para um arquivo.
Tab	Comando usado para auto-completar.

CAPÍTULO 4

OPERAÇÕES BÁSICAS

O Git possui várias operações de manipulação do ambiente de arquivos. Entretanto, algumas são essenciais e usadas frequentemente.

Portanto, é importante ter o domínio destas operações.

Como visto no capítulo anterior, para iniciar o processo de envio de arquivos modificados do *Working Directory* para o *Git Repository* é preciso movê-los para a *Staging Area* e fazer commit das mudanças para que a operação armazene os arquivos no *Git Repository*. A criação de commits está sempre associada ao conteúdo modificado e informações do autor do commit como nome e email. Por isso, é preciso configurar o nome e email do autor no ambiente do Git. Existem dois comandos principais para a configuração de qual usuário será associado a um commit. O primeiro comando é usado para configurar o nome do autor no ambiente através do comando `git config --global user.name <nome>`. O segundo comando é usado para configurar o email do autor no ambiente e pode

ser configurado através do comando `git config --global user.email <email>`. É possível verificar se as informações foram aplicadas corretamente digitando o comando `git config --list`. Este comando exibe várias informações de configurações do ambiente e, dentre elas, o autor e email atrelados aos registros de commits que forem criados. Note que ao configurar o nome e email do autor, uma flag `--global` foi usada junto ao comando, entretanto, existem 3 escopos diferentes de organização dos autores atrelados aos commits:

- **Local (flag `--local`):** configuração para um repositório específico.
- **Global (flag `--global`):** configuração para o usuário atualmente logado e todos os seus repositórios.
- **Sistema (flag `--system`):** configuração para todos os usuários e todos os repositórios no dispositivo.

Por exemplo, se o dispositivo possui dois repositórios chamados de "RepoA" e "RepoB" e a configuração de autor foi dada localmente, ou seja, com a flag `--local`, o autor registrado ao realizar a operação de commit no "RepoA" será diferente ao do "RepoB" caso os dois repositórios tenham sido configurados com autores diferentes.

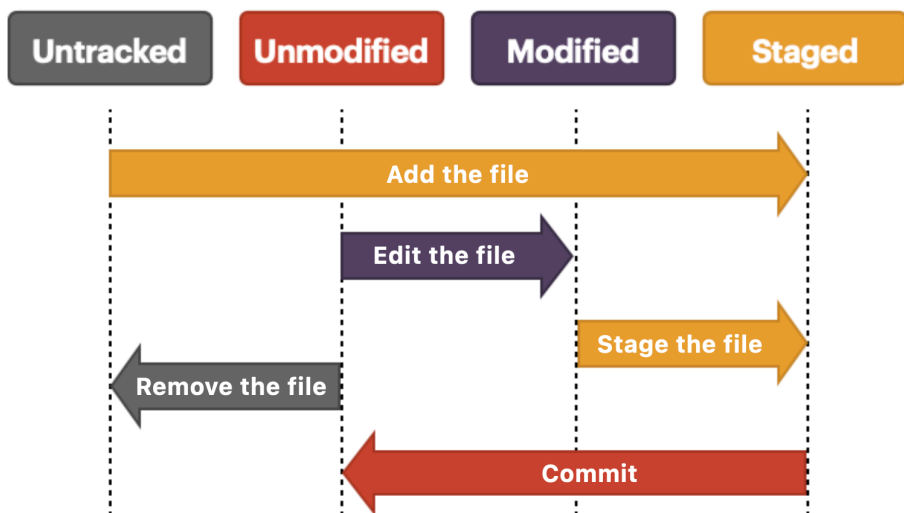
Com isso configurado, agora é possível mover arquivos entre áreas de trabalho do ambiente Git.

CICLO DE VIDA E COMANDOS BÁSICOS

Cada arquivo do Git pode ter um dos 4 status de monitoração: *untracked*, *unmodified*, *modified* e *staged*. Vimos anteriormente também que existem 3 estágios na arquitetura do ambiente Git: *Working Directory*, *Staging Area* e *Git Repository*. O que são esses 4 status e como eles se conectam com as áreas de trabalhos do Git? Para responder a esta pergunta precisamos relacionar as operações que levam os arquivos de um estágio para outro e que cada arquivo pode ter estados diferentes nas áreas de trabalho do ambiente. A funcionalidade de cada estado é a seguinte:

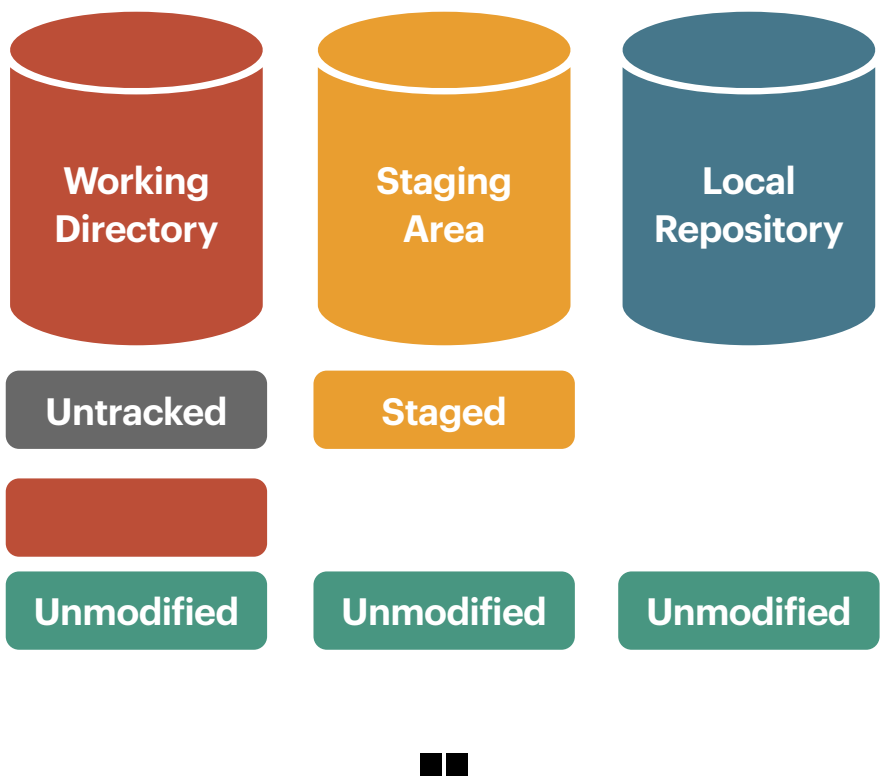
- *Untracked*: o Git não está monitorando este arquivo. Basicamente significa que o Git vê um arquivo que não existia no último commit, ou seja, ele foi adicionado ao *Working Directory*, mas ainda não foi incluído no sistema de rastreamento do Git ou foi removido do controle de versão (com o comando `git rm` que será visto mais a frente).
- *Unmodified*: depois de rastrear um arquivo (usando `git add`), ele se torna "unmodified", o que significa que está sendo rastreado pelo Git, mas não sofreu alterações desde o último commit.
- *Modified*: foram feitas alterações no arquivo após o último commit, mas ainda não foi colocado na área de "staging" para o próximo commit. Essencialmente, são mudanças ainda não adicionadas à área de preparação.

- **Staged:** ocorre quando um arquivo modificado é adicionado à área de preparação usando o comando `git add`. Isso significa que ele está pronto para ser incluído no próximo commit. A área de "staging" é uma maneira de preparar e organizar os arquivos que você deseja incluir em um commit, permitindo que você crie commits de forma granular e organizada.



Por exemplo, no *Working Directory* (ambiente em que é possível editar e trabalhar com arquivos) encontram-se arquivos não modificados (*Unmodified*), modificados (*Modified*) e não monitorados (*Untracked*). Existe um quinto estado chamado de *Deleted*, entretanto, ele é similar ao estado *Modified*. Na *Staging Area* encontra-se arquivos não modificados (*Unmodified*) e preparados (*Staged*). Já no *Git Repository* encontra-se arquivos já salvos, ou seja, commitados. Esses tipos de arquivos estão no estado

Unmodified. Caso exista a mudança de algum arquivo, o seu estado passa a ser *Modified*, ou seja, precisa ser enviado ao ambiente de preparo para ser enviado ao repositório para ser salvo, como mostra a imagem acima.

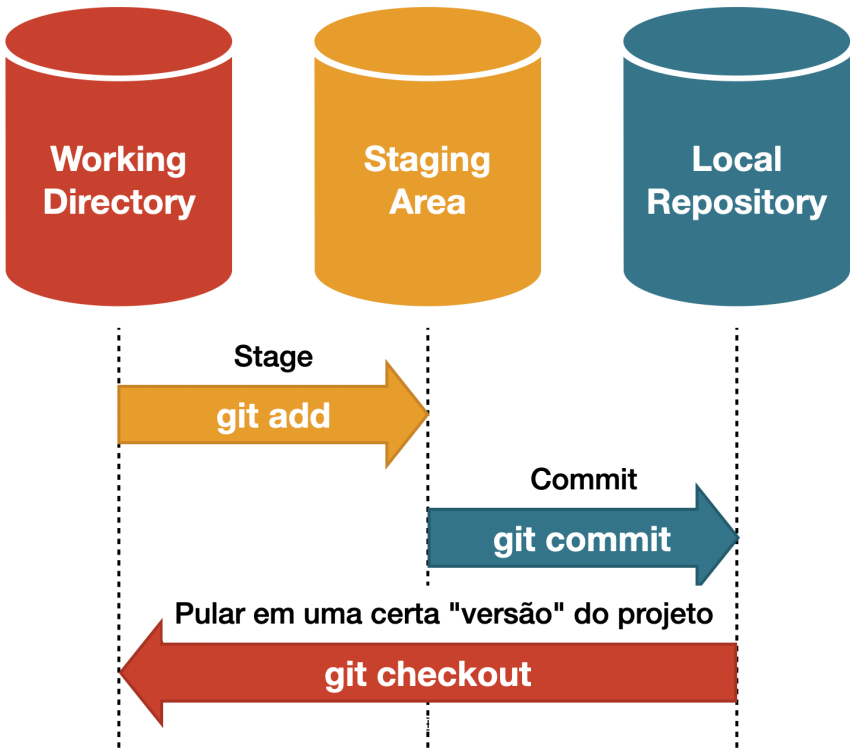


Os conceitos de adicionar um arquivo para a área de preparação, editar um arquivo, remover um arquivo e fazer o commit de um arquivo são conceitos teóricos e que precisam espelhar-se em comandos práticos para serem

aplicados. A lista dos comandos principais e essenciais para a execução dos conceitos está listada abaixo:

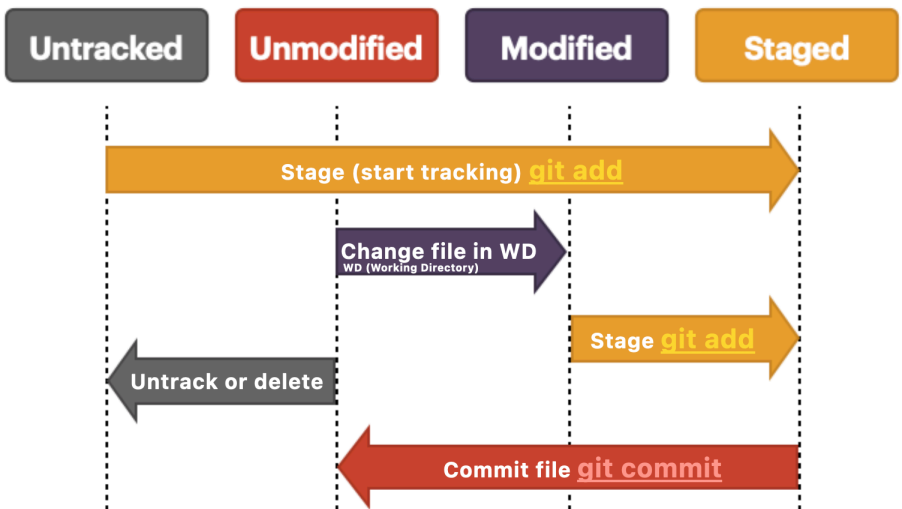
- `git add <arquivo>` - Este comando é utilizado para mover arquivos específicos que foram modificados para a área de preparação, a fim de serem preparados para salvá-los no repositório local.
- `git add .` - Este comando tem a mesma funcionalidade do comando acima, entretanto, ao invés de escolher um arquivo em específico para enviar para a *Staging Area*, envia todos os arquivos que estão na *Working Directory* e foram modificados.
- `git commit -m <mensagem>` - Envia todos os arquivos que foram preparados na *Staging Area* para o repositório local do Git, ou seja, como os arquivos foram preparados para serem adicionados, o Git criou a estrutura do commit para que o *snapshot* da versão que acabou de ser salva possa ser acessada e/ou modificada futuramente. O commit salvará informações como nome do autor, email do autor, mensagem do commit e o *SHA hash* que identifica o commit criado, bem como o *SHA hash* da árvore de arquivos associados a este commit.
- `git checkout <branch ou SHA hash do commit>` - Este comando é utilizado para escolher um commit ou branch (o conceito de branch será explicado futuramente) para qual o usuário gostaria de ir, ou seja, este comando é utilizado para escolher uma versão do projeto a qual ele quer visitar ou trabalhar. Este comando

move os arquivos do repositório local da versão escolhida para o *Working Directory*.



- `git status` - Comando utilizado para verificar o status dos arquivos na área de trabalho do ambiente. Lista os arquivos que estão em no *Working Directory* e foram modificados, na *Staging Area* e prontos para serem commitados, arquivos *untracked*, *modified* e *staged*. O comando é comumente utilizado para verificar o status atual do projeto após editá-lo e/ou usar alguma operação no ambiente.

- `git log` - Comando utilizado para verificar o histórico de mudanças feitas ao longo do tempo no repositório. Este comando lista o histórico de commits feitos.
- `git log --graph` - Este comando faz a mesma listagem do histórico de commits do comando acima, entretanto, mostra graficamente a sequência de commits ao longo do tempo.
- `git rm --cached <arquivo>` - Este comando serve para remover arquivos que já estão na *Staging Area*.



CAPÍTULO 5

BRANCHES, HEAD E MERGING

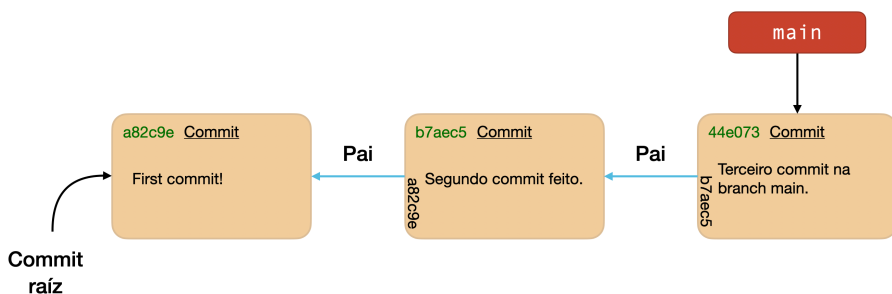
Este capítulo demonstra o uso de branches, explica a função do ponteiro HEAD no Git e como unir o conteúdo de branches distintas.

BRANCHES

Uma branch é apenas uma referencia textual para um commit. Dentro de uma branch é possível realizar commits, ou seja, salvar informações do estado do projeto por alguém do time. Podem existir várias branches com várias versões do projeto no repositório e com isso é possível manter o monitoramento de todas estas versões ao longo do desenvolvimento, realizar o incremento do código por vários colaboradores e juntar o código feito por vários membros do time em diferentes branches para apenas uma. Branches são úteis para dividir o trabalho, ou seja, diferentes coisas podem ser feitas de forma paralela. Por exemplo, na branch chamada de `appConfig` é possível desenvolver a página de configurações de uma aplicação e na branch `appOnboarding` é possível desenvolver a página

de *onboarding* da mesma aplicação. No final do desenvolvimento é possível unir as duas branches em uma só para que todo o trabalho feito em locais separados estejam agora juntos (esse é o conceito chamado de "merge" que será visto mais pra frente).

A branch principal do Git é criada automaticamente e por padrão o nome dado a ela é `main`. Abaixo é possível verificar um exemplo de esquema de commits feitos na branch `main`.



O primeiro commit de um projeto sempre é chamado de commit raiz. Isso se dá pelo motivo de que uma sequência de commits é ligada através do conceito de pai (*parent* em inglês). Todo commit possui um código *SHA hash* que o identifica e uma ligação ao commit anterior. Para armazenar essa ligação, um commit guarda também o *SHA hash* que identifica o commit anterior ou pai. Todos os commits possuem essa estrutura, exceto o primeiro commit que é chamado de commit raiz. No exemplo acima, foram feitos 3 commits ao longo do tempo e a branch `main` está referenciando o último commit.

Em um único repositório podem haver várias branches e o ponteiro de cada branch está localizado na pasta “./git/refs/heads”. É importante mencionar que a branch atual (corrente) monitora novos commits, ou seja, o ponteiro de cada branch move automaticamente depois de cada commit para o commit mais atual (o último commit). Para trocar de branch é muito simples, basta usar o seguinte comando: `git checkout <nome da branch>`. Com este comando é possível alcançar diferentes versões do projeto. O nome de uma branch deve ser caracteres alfa-numéricos. Em projetos de média/grande escala, boas práticas são seguidas para a nomenclatura de branches, como segue na tabela abaixo:

Palavra da Categoria	Significado
hotfix	Usado para rapidamente consertar problemas críticos, geralmente com soluções temporárias.
bugfix	Usado para consertar bugs/problemas/falhas.
feature	Usado para adicionar, remover ou modificar uma feature/modificação.
test	Usado para experimentar algo que não é um bug/problema/falha.
wip	Usado para trabalho em progresso.

É importante sempre escolher um nome descritivo para uma nova branch para que seja possível entender o que

será feito nela. Outras dicas valiosas de boas práticas são evitar usar apenas números e nomes muito longos.

Exemplos de nome de branch:

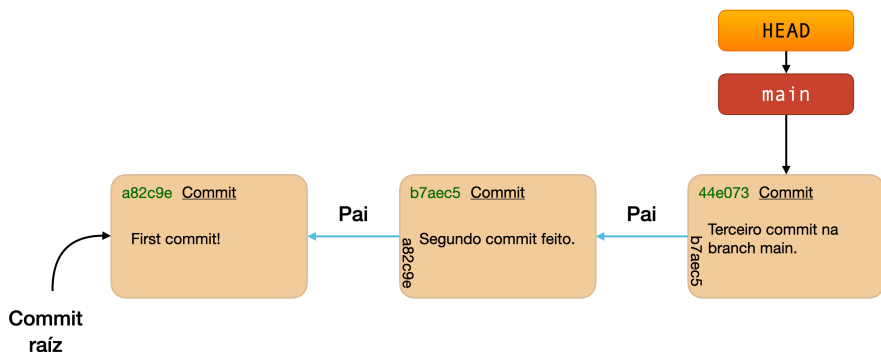
- `wip/450-otimizar-analise-dados` - Este nome indica que a tarefa de otimizar a análise de dados está relacionada ao chamado 450 e é um trabalho em progresso.
- `feature/onboarding` - Este nome indica que a página de Onboarding da aplicação será desenvolvida nessa branch.

O mercado geralmente usa um padrão para a criação de branches em um fluxo de trabalho. Por exemplo, suponha que uma empresa desenvolve um aplicativo para transporte de pessoas e que ela adotou uma estratégia para a criação de branches. A branch `main` não é utilizada, mas ela existe apenas para fins de armazenar o commit raiz. A branch chamada de `prod` é responsável por armazenar versões estáveis da aplicação e que estão disponíveis para o usuário. Geralmente o último commit nessa branch é a versão mais recente da aplicação. A branch chamada de `homolog` é responsável por homologar versões da aplicação antes de ela entrar em produção e ser movida para a `prod`. A branch chamada de `develop` contém versões de pré-produção da aplicação, ou seja, são versões que contém novas *features* que já foram testadas e estão funcionando. Ao longo do ciclo de desenvolvimento, uma variedade de de branches suporte são usadas como, por exemplo, branches começando com `feature-<nome da`

feature>, aonde todas as novas funcionalidade da aplicação são construídas separadamente em uma branch com o nome começando por feature.

O QUE É "HEAD"?

O repositório pode conter várias branches e vários commit, ou seja, é possível acessar várias versões do projeto através do *SHA hash* do commit ou do nome da branch. Entretanto, como o Git sabe em qual versão ele está atualmente (a versão que está no *Working Directory* para ser modificada)? O Git sabe qual é a versão em foco através de um ponteiro chamado "HEAD" que aponta para esta branch ou commit. "HEAD" nada mais é do que uma referencia para a branch ou commit que foi atualmente "*checked-out*", ou seja, para mudar a referência da HEAD, geralmente é associado ao uso do comando `git checkout <nome da branch ou SHA hash>`. Na imagem abaixo, a versão que está disponível para ser usada no Working Directory é o último commit da branch *main*, porque o HEAD aponta para ela.



Outras informações importantes a se notar são a de que "HEAD" é significativo apenas localmente, ou seja, se for o caso de trabalhar com um local de armazenamento remoto (exemplo: GitHub), quando mudar a referência da "HEAD" outras pessoas não sofrerão a mesma mudança em seus repositórios locais e nem o repositório remoto sofrerá com essa alteração (os outros repositórios não vão mover o seu ponteiro "HEAD" para a referência que o usuário moveu localmente). O ponteiro é armazenado em ".git/HEAD" e é possível alterar a referência da "HEAD" para uma branch específica digitando `git checkout <nome da branch>`. O usuário também pode escolher mudar a versão de trabalho para um commit digitando `git checkout <SHA hash do commit>`.

Pode acontecer de que alguém queira ir para um commit e não para uma branch usando o *SHA hash* do commit. Quando isso acontecer, este estado é chamado de "*detached HEAD*". Isso significa que o usuário foi para a versão de um commit anterior que não é o último porque não está associado ao ponteiro de uma branch (uma branch aponta para o último commit na linha do tempo). É possível navegar pelos arquivos da versão do projeto neste commit específico, realizar mudanças experimentais e commitar elas. Você também pode descartar qualquer commit feito nesse estado sem impactar qualquer outra branch fazendo outro *checkout*. Se você quiser criar uma nova branch para reter os commits criados durante esse tempo, você precisa apenas digitar o comando `checkout -b <nome da nova branch>`. Exemplo: `git checkout -b <nome da nova branch>`.

Os comandos que operam sob branches mais utilizados são:

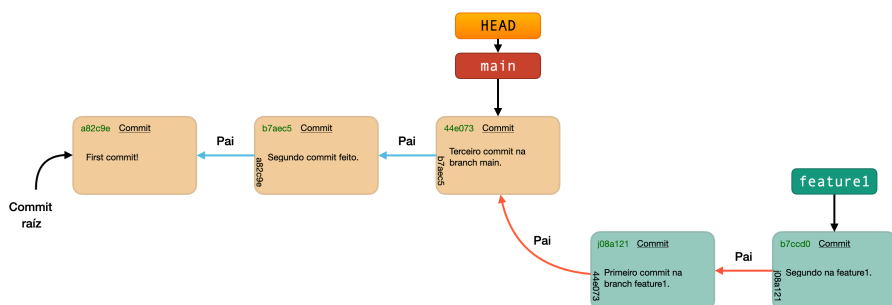
Comando	Descrição
git branch	Lista todas as branches locais (branches remotas seriam, por exemplo, branches criadas por outras pessoas e publicadas/commitadas em um repositório remoto como o GitHub).
git branch <nome>	Cria uma nova branch com o nome dado junto do comando.
git checkout <nome>	Vai para uma branch específica.
git branch -d <nome>	Deleta uma branch específica (deleta todos os commits feitos nelas também) que foi mergeada (para deletar uma não mergeada usar -D).
git branch -m <antigo nome> <novo nome>	Renomeia uma branch específica.
git checkout -b <nome>	É uma junção de dois comandos: criar uma nova branch e ir para ela (git branch e git checkout).

MERGING (UNINDO BRANCHES)

O processo de unir, mesclar ou fundir branches (mais conhecido como *branch merging*) tem como objetivo juntar o trabalho feito em duas branches diferentes em uma só, ou seja, se duas pessoas trabalharam em um projeto, a

primeira desenvolveu a tarefa A e a segunda desenvolveu a tarefa B em branches diferentes, quando o merge das branches acontecer, o esperado é que se veja em apenas uma das branches o trabalho A e B que é equivalente a junção do que foi feito por cada um individualmente. Por exemplo, uma pessoa está desenvolvendo um recurso de listar filmes em uma branch e outra pessoa está desenvolvendo o recurso de pagar para assistir a um filme específico. Como cada um tem apenas um recurso sendo desenvolvido em cada branch, não é possível tentar pagar por um filme na branch aonde só existe a lista de filmes sendo desenvolvida. Para isso, precisamos realizar o merge entre as branches a fim de termos os dois recursos disponíveis em apenas uma branch.

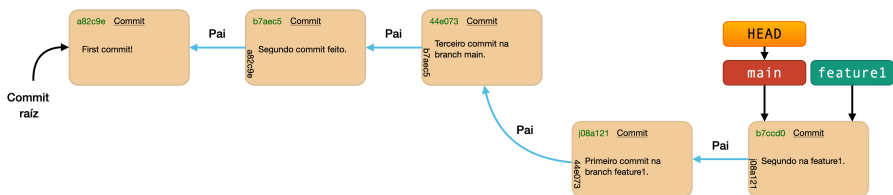
Existem vários processos de resolução de merge. O mais simples deles é o *fast-forward* merge. Este método de junção do conteúdo de branches só pode ocorrer quando a branch que receberá o conteúdo de outra branch se essa outra branch foi criada a partir da branch que vai receber as modificações e se a branch que vai receber as modificações não tiver outros commits feitos depois da criação da nova branch.



Para realizar o processo de *fast-forward* merge precisamos entender alguns conceitos. Primeiramente existem duas branches distintas que vão se unir. Uma das branches receberá o conteúdo da outra, ou seja, a branch que receberá o conteúdo da outra terá todas as modificações da outra branch unidas nela. O processo de *fast-forward* merge acontece em 4 passos:

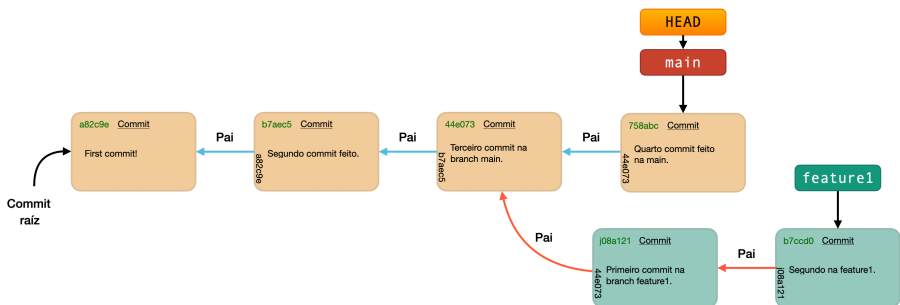
- Realizar o commit das mudanças feitas em ambas as branches;
- Fazer checkout para a branch que receberá todas as mudanças (lembre-se de que uma das branches receberá o conteúdo da outra, ou seja, faça checkout para a branch em que você quer unir todas as mudanças);
- Digitar o comando `git merge <nome da branch a unir>` para aplicar o processo de merge.

O resultado do merge do exemplo da imagem acima usando a estratégia de *fast-forward* resulta no seguinte esquema:

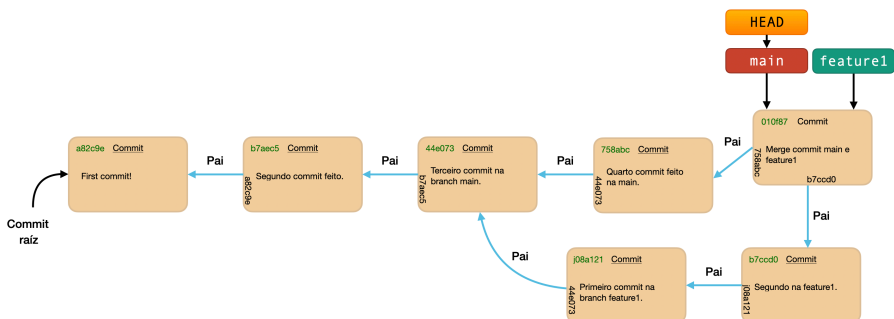


Como a branch `feature1` não precisa mais existir, pois todo conteúdo do último commit dela já está na branch `main`, é possível deletá-la com segurança.

Existe outra estratégia de merge utilizada para resolver merges mais complexos chamado de 3-way. O processo de merge 3-way é uma estratégia diferente do *fast-forward* porque acontece quando a branch que receberá as mudanças já tem novos commits depois de outra branch já ter sido criada a partir dela. O resultado do merge neste cenário pode ser um pouco confuso, por isso é possível utilizar o comando `git log --graph` para analisar o grafo de commits criados graficamente pelo terminal para facilitar a compreensão. Ao analisar o mesmo exemplo do cenário descrito acima, entretanto com apenas um commit a mais na branch `main`, percebemos que a estratégia *fast-forward* não consegue mais solucionar este tipo de merge. Para isso, será aplicado o 3-way merge.



No 3-way merge, o que acontece é que já existe um commit que pode ter informações totalmente divergentes das informações que existem em outro commit. No exemplo, o último commit da branch `main` pode ter mudanças que não estão na branch `feature1` e ela pode ter mudanças que não estão na `main`. Neste caso, o Git junta as mudanças das duas branches e cria um terceiro commit que é o commit merge destes dois últimos commits na branch destino resultando no seguinte esquema:



No processo de merge temos 2 branches: a que vai receber as mudanças e a branch que tem mudanças que vão ser enviadas para a branch receptora, ou seja, branch destino e

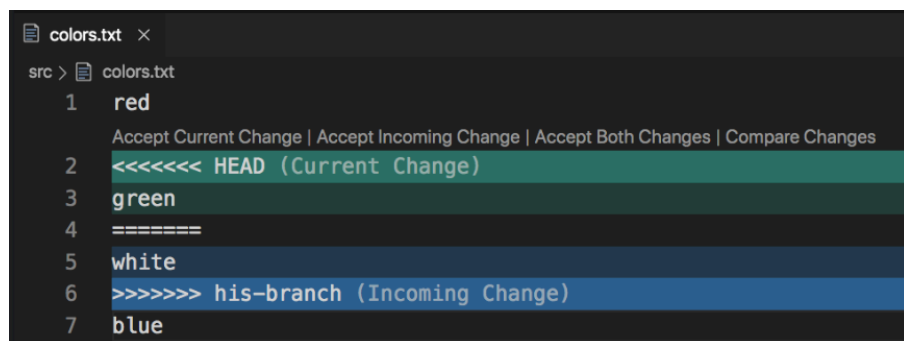
branch fonte, respectivamente. Para que o processo ocorra, é preciso estar na branch receptora, ou seja, é necessário digitar o comando `git checkout <branch destino>` a fim de que o ponteiro "HEAD" se mova para lá. Com a branch receptora em foco então é possível utilizar o comando `git merge <branch fonte>` para que o processo de merge finalmente ocorra. Feito isso, o merge está concluído. Existem casos em que possam ocorrer conflitos de merge. Nesses casos é preciso resolver os conflitos antes de realizar o merge.

CONFLITOS DE MERGE

Pode acontecer a qualquer momento (não é possível que isso aconteça em merges do tipo *fast-forward*) um conflito de merge. Quando isso acontecer, é necessário resolver os conflitos e commitar as mudanças para que o merge ocorra. Caso não seja mais necessário resolver o conflito ou algo aconteça que seja necessário desfazer a operação de merge basta digitar o comando `git merge --abort` que as mudanças que foram trazidas e resultaram em um conflito voltarão ao seu estado original (estado anterior à tentativa de junção das mudanças). O Git consegue encontrar os commits que estão envolvidos no merge devido a mais um ponteiro que se chama "MERGE_HEAD". "MERGE_HEAD" nada mais é do que um ponteiro que armazena o *SHA hash* dos commits envolvidos no merge e o *SHA hash* do commit que originou a branch fonte. Conflitos de merge acontecem quando um mesmo arquivo em branches distintas possuírem diferentes conteúdos neles. Neste caso será necessário decidir quais mudanças

ficarão e quais mudanças serão descartadas para que o arquivo esteja em harmonia ou estável.

A ferramenta comumente utilizada para resolver conflitos no mercado é o Visual Studio Code. Inclusive, o aplicativo GitHub Desktop, que é uma interface para visualizar e manipular as operações do Git, recomenda resolver os conflitos quando acontecem através dele. É possível resolver conflitos diretamente via terminal, entretanto, quanto maior o arquivo ou projeto, mais complicado fica de se resolver, pois podem haver vários conflitos em um único arquivo. Com isso, é importante ter em mãos uma ferramenta que facilita a visualização e decisão de quais mudanças queremos descartar e quais mudanças queremos manter. Após todas as mudanças serem resolvidas, deve-se adicionar elas na área de preparação e fazer o commit das mudanças para concluir o merge. O Git finaliza o processo de merge criando um novo commit com a junção de ambas as branches e o ponteiro "HEAD" passa a apontar esse novo commit que coincidentemente é a branch destino. A imagem abaixo mostra o exemplo de um arquivo com conflito aberto no Visual Studio Code.



```
src > colors.txt
1  red
2  <<<<<< HEAD (Current Change)
3  green
4  =====
5  white
6  >>>>>> his-branch (Incoming Change)
7  blue
```

CAPÍTULO 6

REPOSITÓRIOS REMOTOS

Desenvolver um projeto distributivamente, ou seja, com outros colaboradores requer o uso de um repositório remoto ou público. Este capítulo demonstra o uso de operações para este repositório colaborativo.

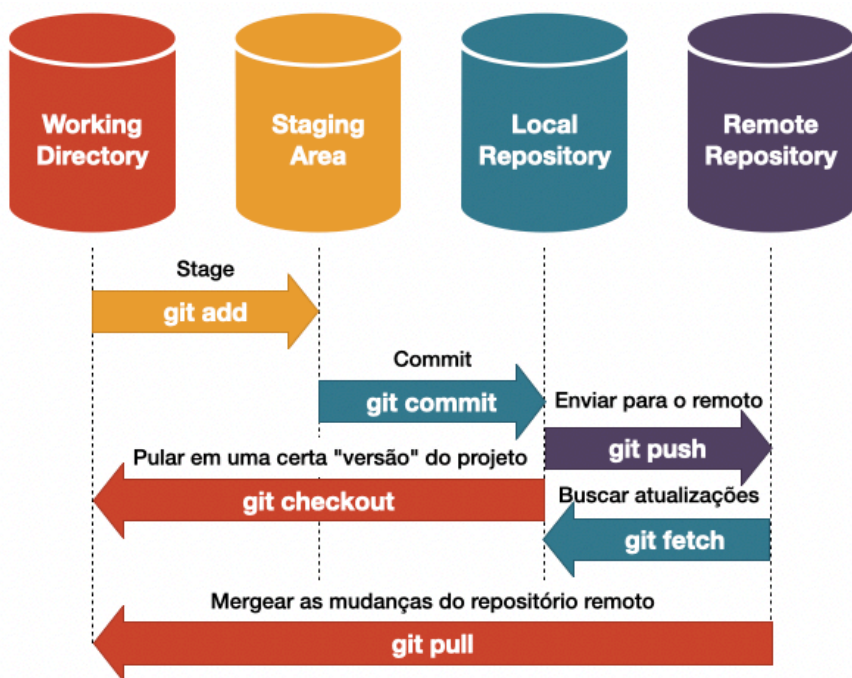
Repositórios públicos ou remotos são repositórios que armazenam as informações em um serviço de hospedagem em nuvem, ou seja, todos os arquivos de um projeto estão visíveis para todos os colaboradores dele. Através de um repositório remoto é possível compartilhar arquivos e trabalhar distribuída. Para trabalhar com repositórios públicos é preciso primeiro criar ou clonar um de algum serviço de armazenamento. Para fins de demonstração, este documento utilizará da plataforma GitHub. Temos diversas maneiras de clonar um repositório do GitHub, ou seja, puxar um projeto em que alguém está trabalhando:

- Clonar via HTTPS (trará a pasta .git junto e é a maneira mais comum de se clonar um repositório);
- Download do ZIP do projeto (não vai trazer junto do download a pasta .git, ou seja, não vai ter o histórico de commits nem branches);
- Abrir o projeto direto na ferramenta GitHub Desktop.

Para encontrar as maneiras disponíveis de se clonar um repositório, basta acessar o site do GitHub e procurar o repositório que quer clonar. Por exemplo: <https://github.com/leachim6/hello-world.git>.

Também é possível criar o próprio repositório remoto no GitHub. Para isso, basta acessar o site do GitHub e clicar em 'Novo Repositório'. Este repositório estará disponível para qualquer colaborador que for adicionado no projeto. Ele será permitido de clonar (puxar para seu computador) o repositório e trabalhar nele. Para clonar o repositório remoto basta copiar o link HTTPS dele (é possível encontrar o link seguindo as instruções citadas acima) e digitar o seguinte comando no terminal: `git clone <link do repositório>`. Geralmente o link de um repositório termina com ".git".

Com o repositório clonado, uma nova camada ou estágio é adicionado a arquitetura do ambiente e ela é chamada de Repositório Git Remoto ou repositório remoto como mostra o esquema:



Com essa nova camada, novos comandos são necessários para mover arquivos entre as áreas do ambiente.

OPERAÇÕES BÁSICAS

Como visto anteriormente, ao clonar um repositório, dois tipos de repositórios passam a existir: o repositório local e o repositório remoto. Com a existência de um novo repositório, novos mecanismos para identificar e operar sobre eles são necessários. Dentre esses mecanismos, as operações mais importantes são as de envio, deleção, atualização e busca de mudanças e arquivos de ambos os ambientes.

É possível enviar mudanças para diferentes repositórios remotos, ou seja, colaborar em diferentes projetos. Entretanto, é preciso existir operações para identificar os prováveis vários repositórios em que o usuário queira trabalhar ou utilizar paralelamente. O comando que faz a listagem dos repositórios para que seja possível os identificar é `git remote`. Ao digitar o comando no terminal, um repositório com o nome `origin` pode aparecer na lista. O que isso significa? Isso significa que o repositório remoto padrão é o `origin` (analogamente, a `branch main` é a branch padrão quando um novo repositório é criado, remoto ou local). As operações de envio e busca de arquivos está atrelada a links que são utilizados como se fossem canais ou tubos por onde a informação será enviada. Ao digitar o comando `git remote -v` é possível ver quais os links estão atrelados ao envio (*push*) e busca (*fetch*) de informações.

Os 3 principais comandos para operar a conexão entre o ambiente local e os ambientes remotos e a descrição de suas respectivas funções são:

- `git fetch [repositório remoto]` - Copia todas as informações contidas no repositório remoto e cola no repositório local, entretanto não atualiza as branches monitoradas. Por exemplo, todas as branches, commits e blobs são enviados para o repositório local. Vale lembrar que, como o parâmetro do comando de nome do repositório está entre colchetes, não é obrigatório passar o nome do repositório.

- `git pull` - Este comando é um processo que une dois comandos do git sequencialmente: o `fetch` seguido do `merge`. Basicamente este comando copia todas as informações do repositório remoto para o local e logo em seguida trás essas mudanças do repositório local atualizado para o *Working Directory*. É possível que existam diferenças entre os arquivos que foram trazidos para o *Working Directory* podendo, assim, causar conflitos de merge. Esses devem ser resolvidos para que as mudanças naquela branch possam continuar ocorrendo localmente. Para realizar o processo de *pull*, deve-se seguir os seguintes passos: *checkout* para a branch que o usuário quer atualizar o *Working Directory* com as mudanças mais recentes do repositório remoto e puxa as mudanças do repositório remoto digitando o comando `git pull`.
- `git push` - Comando usado para enviar dados para o repositório remoto. O *push* faz a atualização das mudanças do repositório local no repositório remoto, ou seja, todos os novos commits, branches e arquivos criados no computador de um colaborador são enviados para o repositório remoto a fim de que todos possam ter acesso a esta versão que, no caso, é a versão mais atualizada. Para enviar os dados para o repositório remoto é preciso ter mudanças feitas no projeto, caso contrário, o Git entende que não há a necessidade de operar este comando porque o repositório remoto já está atualizado com o local. Quando houverem mudanças, é necessário fazer o commit delas para que elas estejam salvas no repositório local. Por que? Porque o Git envia

arquivos modificados que estão no repositório local, ou seja, é preciso salvar as mudanças feitas para que a operação possa ser feita. Mudanças que estão no *Working Directory* ou na *Staging Area* não estão ainda gravadas no repositório local. Por isso, é necessário fazer commit dessas mudanças a fim de subir as novas alterações para o repositório remoto. Feito o commit é só digitar o comando `git push`. Se uma nova branch tiver sido criada, o Git trará um erro de não reconhecimento desta branch no repositório remoto. Portanto, basta digitar `git push --set-upstream origin` para que essas mudanças ocorram.

Quando ocorrer o envio de dados para o repositório remoto com o uso da flag `--set-upstream`, o Git coloca a branch atual, ou seja, a branch em que o usuário estava quando o *push* foi feito como sendo monitorada. Adicionar uma branch remota para ser monitorada significa que o Git sabe o que o usuário vai querer fazer quando usar `git fetch`, `git pull` ou `git push` no futuro.

Uma branch monitorada remotamente (*remote tracking branch* ou *tracking branch*) é uma branch que existe no repositório remoto e o Git sabe qual é a branch equivalente no repositório local de algum colaborador. Com isso, é possível realizar as operações citadas acima com o Git entendendo para qual branch ele envia os dados atualizados. Por exemplo, quando uma nova branch é criada com `git branch <nome>` no repositório local, o repositório remoto ainda não sabe da existência desta branch e, portanto, ele não consegue saber de onde pegar

os dados ou para qual branch ele envia os dados. Por que ele não consegue enviar os dados? Porque ele não sabe em qual branch do repositório remoto ele deve colocar essas novas atualizações. Para que ele consiga relacionar qual branch remota ele deve enviar os dados da branch local é possível conectar as branches com algumas alternativas:

- Enviando os dados direto com a flag `--set-upstream` como visto no comando acima (ele irá criar uma branch com o mesmo nome no repositório remoto e já fazer *tracking* da branch);
- Criando a branch no repositório remoto e conectando ela com o repositório local. Para isso é preciso fazer um *fetch* para pegar as atualizações e trazer a nova branch criada no remoto e conectá-la utilizando o comando `git branch -u origin/<nome da branch>`;
- Criando a branch no repositório remoto e conectando ela com o repositório local fazendo o *fetch* dos dados atualizados e dando *checkout* para a branch remota digitando o comando `git checkout <nome da branch remota>`. Com isso, ele fará automaticamente a conexão com a branch remota, tornando a branch monitorada.

O funcionamento básico e as operações essenciais são a fundação para que tópicos mais complexos e aprofundados do Git sejam compreendido com mais facilidade. Com esse conhecimento já é possível minimamente gerenciar um repositório por completo.