SHELL/BASH:

Windows: -cd, -dir, -mkdir, -del <arquivos>/ -rmdir <diretório> /S /Q

Unix: -cd, -ls, -mkdir, -rm -rf (deleção recorrente)

--------GIT: Trabalhando com hash:

openssl sha1 <nomeDoArquivo>

echo 'conteudo' | git hash-object --stdin :

>>>função de git espera receber arquivo (devido --stdin) e retorna o SHA1

--------USANDO O OPENSSL DO BASH NORMAL:

echo -e 'conteudo' | openssl sha1 :

>>>o -e é específico do echo para interpretar parâmetros em strings:

```
      \a    Alert (bell)
      \b    Backspace
      \c    Suppress trailing newline
      \e    Escape
      \E    Escape
     \f    Form feed
      \n    New line
      \r    Carriage return
      \t    Horizontal tab
      \v    Vertical tab
      \\    Backslash
      \0nnn   The eight-bit character whose value is the octal value nnn (zero to
three octal digits)
              if nnn is not a valid octal number, it is printed literally.
      \xHH    The eight-bit character whose value is the hex value HH (one or two
hex digits)
      \uHHHH    The Unicode (ISO/IEC 10646) character whose value is the hex value
HHHH (one to four hex digits)
      \UHHHHHHHH  The Unicode (ISO/IEC 10646) character whose value is the hex
value HHHHHHHH (one to eight hex digits)
```

Openssh está no bash do linux, gerando SHA1 diferente do SHA1 do GIT.

Isto porque o GIT trabalha com objetos para manipular arquivos.

OBJETOS GIT:

--------BLOBS: são blocos básicos de composição do GIT

Objeto (BLOB) terá metadados próprios, o que faz o SHA1 ser diferente.

~~~~Conteúdos do BLOB: tipo-objeto (blob), tamanho-arquivo, /0 e o SHA1

do arquivo (BLOB NÃO GUARDA NOME DO ARQUIVO, SÓ O SHA1) que está armazenando.

>>>FAZENDO SHA1 COM BLOB:

echo -e 'blob 9/0conteudo' | openssl sha1 :

VAI PASSAR METADADOS DO BLOB NO ARQUIVO DE tamanho 9, /0, E O conteudo

UNINDO TUDO SOBRE SHA1:

$ echo 'conteudo' | git hash-object --stdin

fc31e91b26cf85a55e072476de7f263c89260eb1

$ echo codes.txt | git hash-object --stdin

10388c9e2985fb14075370f1a788feefce0290d4

$ echo -e 'blob 9/0codes.txt' | openssl sha1

(stdin)= 14affa297d47e7d6fda588a98e4bd4e26801fe92

--------TREES: São objetos que armazenam e apontam para BLOBS e COMMITS.

É responsável por montar estrutura de localização do arquivo, de modo que

aponta tanto para BLOBS (a partir de seu SHA1 único)

quanto para outras árvores (UMA VEZ QUE S.O USA DIRETÓRIOS RECURSIVOS DE FORMA HIERÁRQUICA)

Objeto (TREE) terá metadados próprios da hierarquia que armazena (SHA1 ÚNICO).

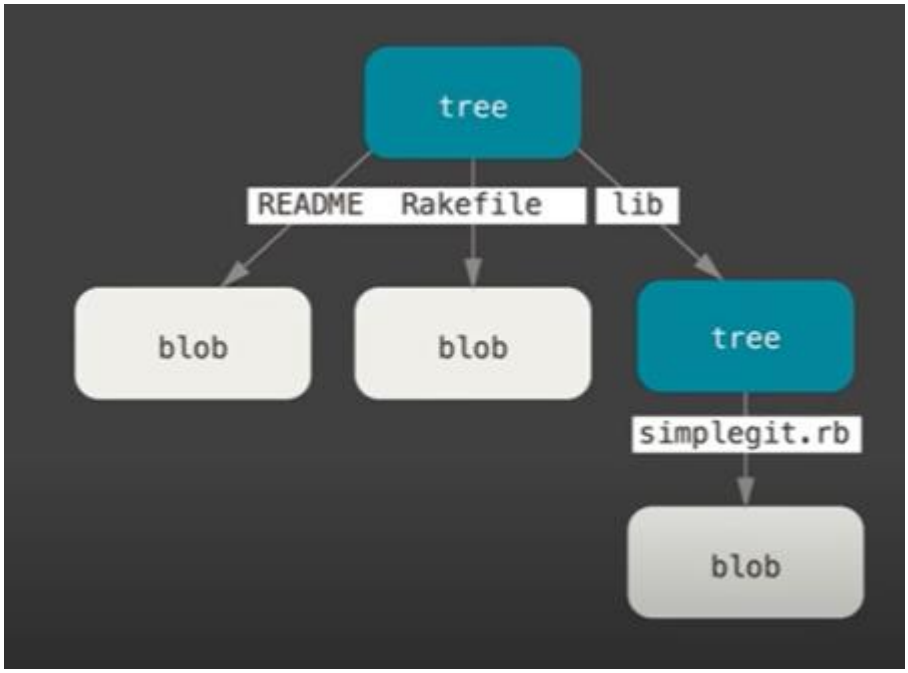
~~~~Conteúdo na Tree: tipo-objeto, tamanho-arquivo, /0, blob ou commit para o qual

está apontando, SHA1 deste objeto apontado (ex: blob) e NOME DO ARQUIVO que o

blob está armazenando.

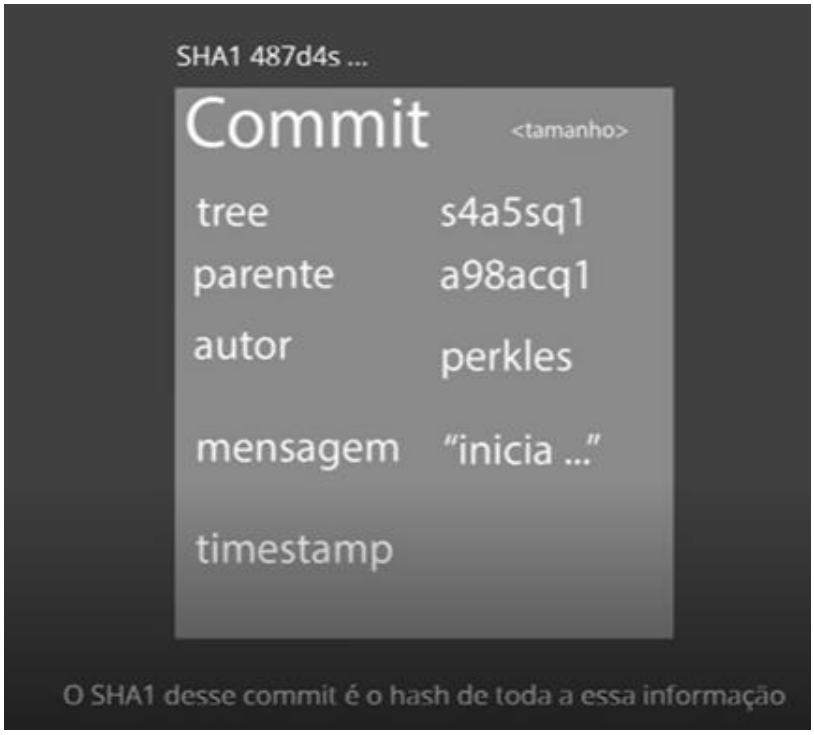
ENQUANTO BLOB GUARDA SHA1 DO ARQUIVO, A ÁRVORE GUARDA SHA1-blob (identificação do blob) e o nome-arquivo dentro do blob (intra-blob)

ÁRVORE RETÉM INFORMAÇÃO DE ARQUIVOS, APONTANDO PARA BLOBS, OU ÁRVORES (que contém outros blobs), DE MODO QUE SERVE PARA ENCAPSULAR O COMPORTAMENTO DE DIRETÓRIOS E APONTAR PARA DIRETÓRIOS COM ARQUIVOS.



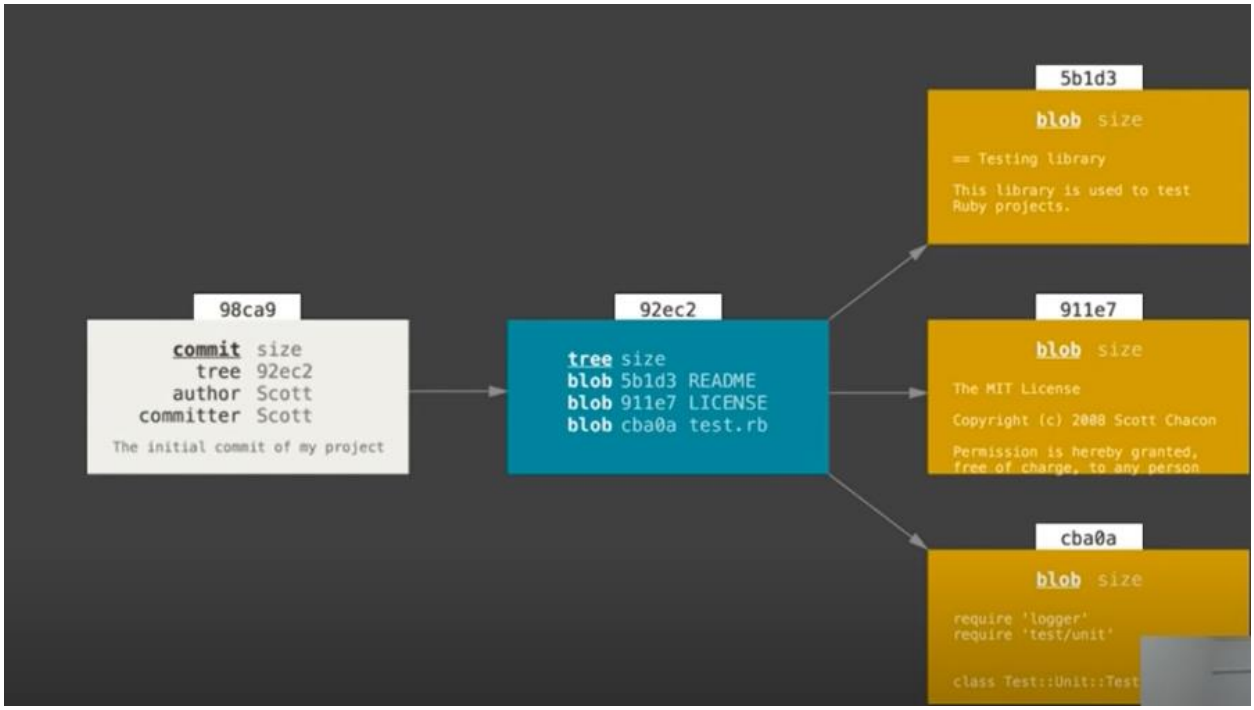
--------**COMMITS:** Objeto que reúne a hierarquia completa E DÁ SENTIDO ÀS ALTERAÇÕES IDENTIFICADAS PELOS SHA1 do GIT:

Conteúdo do commit: SHA1 com metadados próprios, além de apontar para árvore com SHA1 próprio, aponta para **parente (= commit anterior a ele)**, aponta para autor e para mensagem contida no commit (**autor e mensagem trazem o sentido das alterações dos arquivos nos blobs, diretórios nas árvores etc**), por fim, commit contém TIMESTAMP.



Como COMMIT aponta para uma árvore (e esta pode apontar para outras árvores), ao mudar qualquer coisa no arquivo, isto refletirá na árvore e, consequentemente, no COMMIT.

- Uma vez que COMMIT APONTA PARA SEU **PARENTE**, e o parente aponta para outro parente etc..., MONTA-SE UMA TIMELINE DE COMMITS DE MODO SEGURO E ÍNTEGRO (pois fica claro no histórico de COMMITS).



GIT É SISTEMA DISTRIBUÍDO CONFIÁVEL: tanto versão mais recente no servidor da nuvem GitHub quanto versões distribuídas em cada máquina dos usuários mantainers dos commits possuem SHA1 único, igual e confiável.

GERENCIAMENTO DE CREDENCIAIS PELO GITHUB:

> No GIT ainda é possível só a autenticação login e senha, assim como do GIT para outras plataformas.

CHAVES SSH PARA AUTENTICAÇÃO NO GITHUB:

- Chave pública da máquina deve ser enviada para GitHub para que este pré-reconheça a máquina física do commit.
- Settings > SSH e GPG keys > NEW SSH KEY > ir no git bash

No GIT BASH: ssh-keygen –t <tipo de criptografia da chave dada pelo github = > ed25519 –C <email>

- Localização: /c/Users/flavi/.ssh/id_ed25519.pub
- Colocar no github
- CLI: **eval $(ssh-agent –s)** e iniciará ssh no plano de fundo com o PID mostrado
- ssh-add <caminho de onde a chave privada está (é a chave sem o .pub)> e colocar senha
- Clonar repositório = pegar repositório do github e clonar na máquina
- Não se usa git clone <url> quando se usa chave SSH da máquina, e sim ir em Code do repositório e git clone <ssh>

Fazer isto no LINUX agora:

- ssh-keygen –t <tipo de criptografia da chave dada pelo github = > ed25519 –C <email>

TOKEN PARA AUTENTICAÇÃO NO GITHUB:

PODE CRIAR TOKEN EM VEZ DE USAR O SSH, MAS PRECISA ARMAZENAR NA MÁQUINA O ARQUIVO CONTENDO-O.

- Settings > Developer settings > Personal access tokens > New token e colocar descrição, expiração do token e outras permissões para uso do token no github e Generate token > COPIAR O TOKEN E SALVAR EM ALGUM LUGAR
- Agora para clonar é usar git clone <URL> (pelo token, usa-se o HTTPS)
- Será gerado um processo similar à federação para colocar o token para autenticação

- ***GIT CREDENTIAL MANAGER CORE*** (usado neste processo) ***SÓ ESTÁ DISPONÍVEL A PARTIR DA VERSÃO GIT 2.30***

COMANDOS PARA USAR O GIT

- **INICIAR REPOSITÓRIO GIT:**

git init inicializa repositório no GIT, sendo ele o repositório onde comando rodou, a ser upado para o github. Este comando gerará a pasta /.git , que é uma pasta oculta (usar ls –a) por ser pasta GERENCIAL (de manipulação de objetos git). É possível, contudo, verificar que repositório está com (master) por já ser um repositório git.

FORNECER O EMAIL DO AUTOR DO COMMIT COM:

git config –global (está setando o autor master de forma global, em vez de para um repositório específico)/**--local** (para o repositório apenas) **user.email** **flaviarosadolima@gmail.com**

git config –global user.name <Nome>

CASO ERRE A CONFIGURAÇÃO, BASTA: **git config --global --unset <user.email/user.name etc> <valor do parâmetro>**

LEMBRANDO QUE ISTO TUDO ESTÁ OCORRENDO NA PASTA ONDE O GIT INIT RODOU

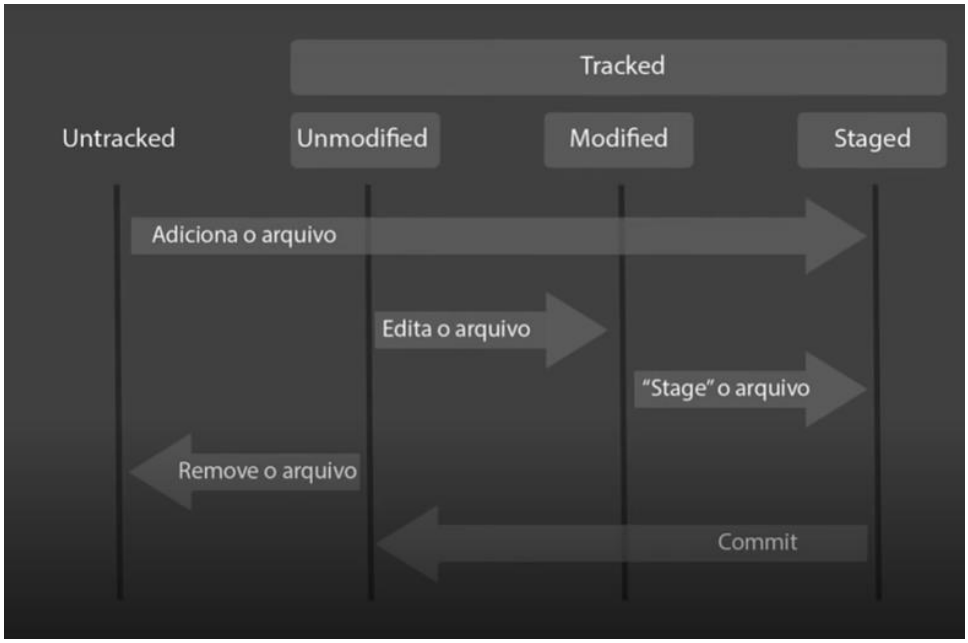
- **MOVER ARQUIVOS E INICIAR VERSIONAMENTO NO GIT:**

git add * vai adicionar tudo ao versionamento ‘ativado’

- **CRIAR COMMIT:**

git commit –m (flag de commit) **“Comentário sobre commit”**

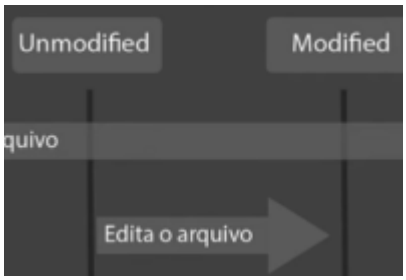
Arquivos TRACKED e UNTRACKED pelo GIT:



Arquivos rastreados (arquivos que GIT tem ciência da sua existência no repositório) e podem ser NÃO MODIFICADOS, MODIFICADOS ou ENCENADOS.

- Unmodified
- Modified: arquivos que sofrem alterações, mudando o SHA1
- Staged: onde ficam arquivos que estão se preparando para participarem de um agrupamento no Git

Arquivos UNTRACKED: git não tem ciência dele apesar de estar dentro do repositório/arquivo não está sendo versionado, pois o arquivo ainda não sofreu GIT ADD ou GIT COMMIT prévio. **Após git add, o arquivo untracked é movido para tracked staged.**



Arquivo que não sofreu modificação **após algum git init** é um arquivo TRACKED UNMODIFIED, ou seja, GIT tem ciência dele mas nunca foi modificado. **Após editar o arquivo, ele é marcado como arquivo TRACKED MODIFIED pela COMPARAÇÃO DO SHA1 pelo GIT.**

O arquivo MODIFIED antes de UM NOVO git add não faz parte da fila atual do GIT **(ou seja, APÓS MUDAR DE UN PARA MODIFIED, o GIT já conhece o arquivo, já sabe que ele mudou, mas o arquivo MODIFIED ainda não foi colocado na esteira de versionamento)**

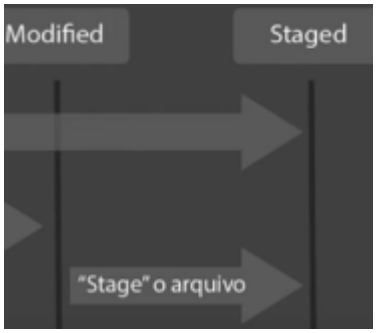
APÓS UM NOVO GIT ADD com o arquivo MODIFIED, o arquivo **muda para STAGED**, ou seja, é colocado na esteira de versionamento:

TRACKED STAGED:

o arquivo fica ensaiando/na esteira de versionamento

do backstage/AGRUPAMENTO ESPECIAL criado pelo GIT

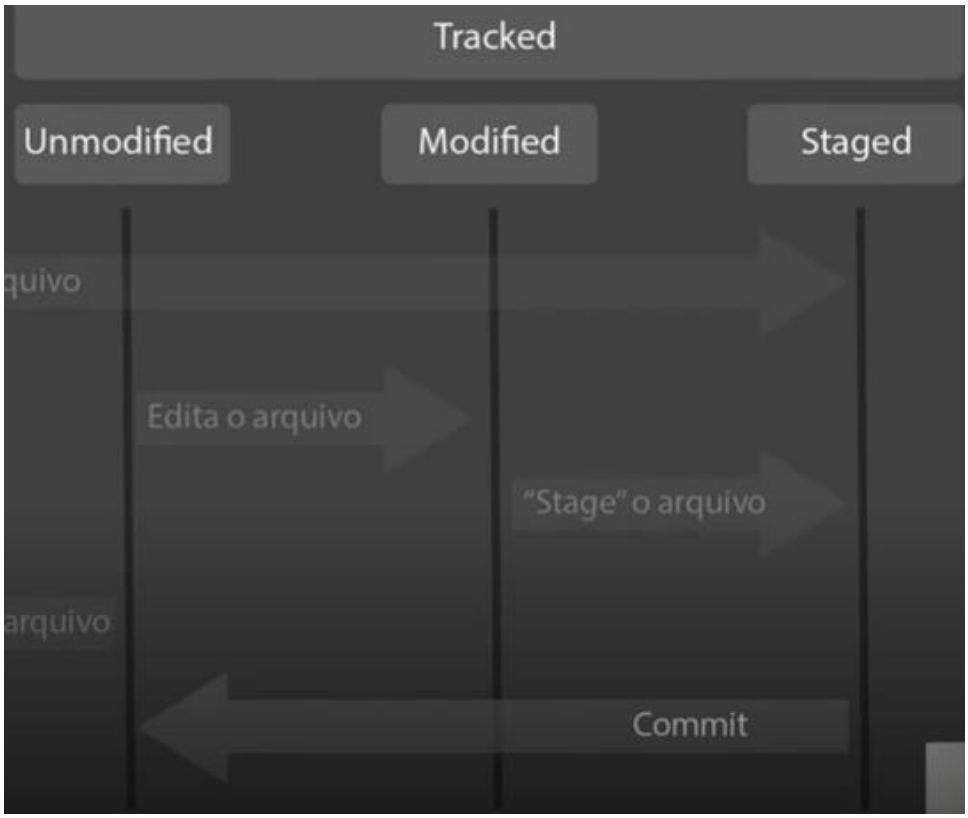
para entrar em palco/executar uma ação



Um arquivo **TRACKED UNMODIFIED**: após algum git commit (ou seja, o GIT sabe que ele existe em seu repositório) ainda não sofreu nenhuma ação até então. Se o arquivo simplesmente **FOR REMOVIDO**, **voltará para a pilha de UNTRACKED.**

Quando o arquivo está pronto para sofrer alguma ação, quer dizer que ele foi agrupado/encapsulado pelo GIT para sofrer um git commit. **Ao sofrer git commit, GIT envelopa todo o conteúdo, faz a assinatura de autor, mensagem, timestamp, gera SHA1 e "lança" o conteúdo para a plataforma.**

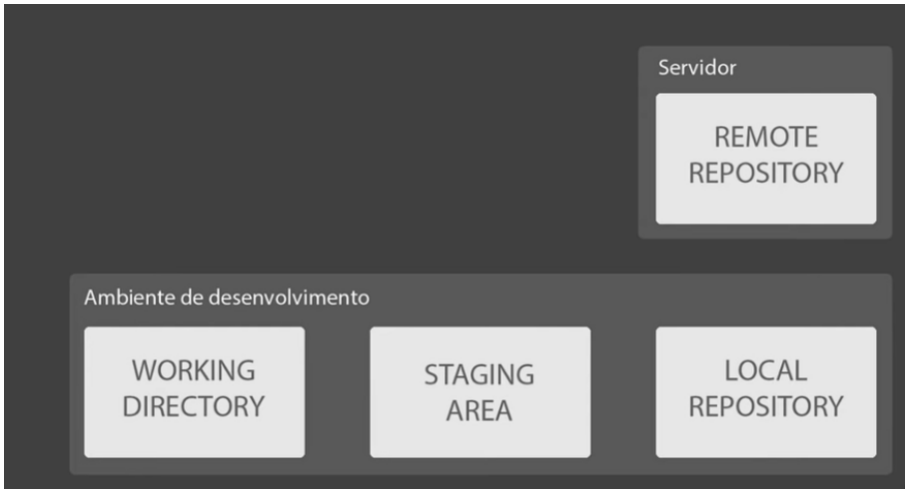
Após git commit, os arquivos STAGED voltam para UNMODIFIED para que haja gerenciamento e controle de versionamento pelo GIT.



GIT commita: isto significa que ele terminou de fazer as alterações necessárias no agrupamento de código e faz um **SNAPSHOT da situação atual do agrupamento/código**. Após isto, **Git SALVA O SNAPSHOT DENTRO DO OBJETO COMMIT e retorna os arquivos (após a 'foto') para UNMODIFIED.**

OU SEJA: EXISTE UM CICLO ENTRE UNMODIFIED, MODIFIED E STAGED.

Em relação ao git init, inicia-se um REPOSITÓRIO:



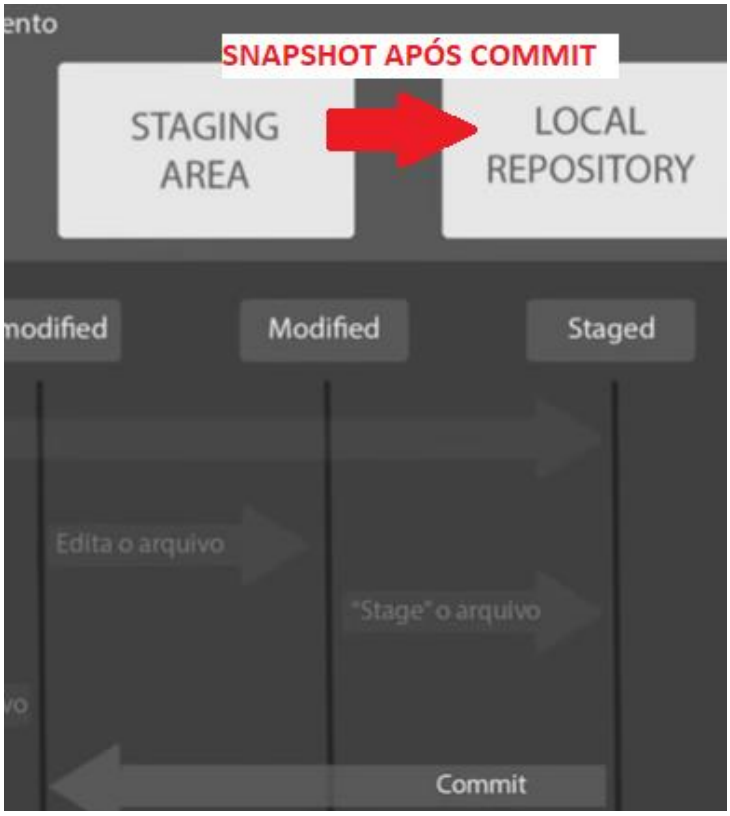
Há dois ambientes: **Ambiente de DEV** (tudo que está na máquina) e o **Ambiente SERVIDOR** (sistema distribuído remoto do GIT, a exemplo do GitHub).

Ambiente de Desenvolvimento é composto por **DIRETÓRIO DE TRABALHO, ÁREA DE ENSAIO/STAGING AREA** e **REPOSITÓRIO LOCAL**

- **WORKING DIRECTORY** é onde o desenvolvedor está manipulando os arquivos de código (git init)
- **STAGING AREA**: onde arquivos agrupados para ação de commit ficam.

Arquivos no ambiente de desenvolvimento ficam alterando entre WORKING DIRECTORY e STAGING AREA.

- **LOCAL REPOSITORY: quando se faz um COMMIT, o arquivo,** que fica entre o diretório e o staging área, **passa a se integrar ao REPOSITÓRIO LOCAL do Git na máquina.**



OS ARQUIVOS NO REPOSITÓRIO LOCAL PODEM SOFRER **push** PARA O REPOSITÓRIO REMOTO DO GIT, o que causa, por fim, a alteração de código no GitHub/plataforma

- **VERIFICAR ESTADO DE CADA UM DOS ARQUIVOS E DOS OBJETOS**

git status: mostra o que tem em cada branch das working trees no WORKING DIRECTORY , classificação dos arquivos

- Quando muda arquivo de pasta, por exemplo, deve-se fazer novo git add para 1. reconhecer que há nova pasta (untracked para tracked staged) e 2. reconhecer que o arquivo mudado está na nova pasta (modified para staged):

git add <arquivo_mudado> <nova_pasta> para levar para staged e depois commitar

git restore –staged <arquivo> PARA TIRAR DE STAGED

INTERAÇÃO ENTRE LOCAL REPOSITORY e REMOTE REPOSITORY (GIT e GITHUB)

git config –list: traz lista das configurações do GIT

git remote add origin <link do github> : PARA CRIAR O REPOSITÓRIO ORIGEM

git remote –v : lista os repositórios remotos no git. O ORIGIN é alias para o link https do github

git push -u <alias, por padrão: **origin> <**branch de onde está enviando codigo: **main/master etc>**

RESOLVENDO CONFLITOS DO GIT local repo <> GITHUB remote repo

git pull <alias> <branch>: FAZ MERGE DE TODAS AS BRANCHES DO PROJETO PARA MANTER TUDO CONSISTENTE. **SEMPRE FAZER PULL ANTES DE INICIAR UM PROJETO.**