**ESTUDO GIT – Rodrigo Branas**

O Git é um VCS, Version Control System ou sistema de Controle de Versão, criado por Linus Torvalds em 2005 para auxiliar no desenvolvimento do Kernel do Linux.

**Git init** – Cria repositório vazio pra começar a utilizar.

Armazena na pasta .git

Tree.git/ - Estrutura de pastas , todos os arquivos do projeto.

**Git**  **add** – Armazena no contêiner (staging área) para fazer o git commit

**Git status** – Eles são armazenados para pode fazer o commit

**Git commit** - -m “a.txt e b.txt”

**Git log** – Mostra lista de commit realizados.

**Git Branch** – Criar e verificar branch ( git branch feature/nome )

**\*Criando um arquivo e depois dando commit em uma branch** :

Echo d > d.txt

Git add – A

Git commit -m “d.txt”

**\*Comando para visualizar o que você fez**:

Git log - - online - - decorate

Comando para mostrar outros branch:

Git log - - online - - decorate - - all

**\*Repositorios no GIT HUB**

Repositório criado no Git Hub, ou clona ou da um remote – Comando: git remote origin link do repositório. Comando para vê mas detalhado: git remote -v ou -verbose.

Empurrar arquivos para o repositório – Comando: git push origin máster

Puxar arquivos - Comando: git pull origin máster

**\*Clonando repositório do GIT HUB**

Clonando 1 repositorio 2 vezes – Comando: git clone url gitrepresentation2

**\*Entrando no diretório GIT**

Comando cd nome do diretório

Git fetch

Faz o pull pela metade e só traz atualização e não faz merge

Observação: O git pull faz merge automaticamente.

**Estudo Git – Mango**

Anotações de Git

**\*Inicio de Repositório**

**. git init**: iniciar um repositório local

**. git clone**: clonar um repositório remoto

**. git remote**: acessar o repositório remoto   
  **git remote -v**: listar os repositórios remotos   
**git remote add origin** url: integrar repositório remoto ao repositório local

**\*Configurações de Usuário**

. git config: acessar as configurações (sistema, usuário ou projeto)   
  git config --list: exibir lista de comandos   
  git config --system --edit: abrir arquivo para editar   
  git config --global --edit: editar configurações do usuário   
  git config --global core.editor code: abrir configurações de usuário no VS Code   
  git config --local --edit: abrir configurações do projeto

! Nota: configurações no arquivo do projeto tem mais prioridade do que configurações no arquivo do usuário, que por sua vez tem mais prioridade do que configurações no arquivo do sistema.

. git status: listar quais arquivos sofreram alteração   
  git status -s: versão reduzida do git status

. git add . ou (nome do arquivo): adicionar todos os arquivos ao commit (somente se você estiver no diretório certo   
  git add --all: adicionar todos os arquivos ao commit

. git commit -m “nome do commit”: criar commit

! Nota: para criar atalhos (alias) você precisa abrir o arquivo de configurações e adicionar um alias, exemplo: (s = !git status -s), nesse atalho, você fez com que toda vez que você digitar “git s” o comando “git status -s” seja ativado.

. git commit --amend --no-edit: junta o commit atual com o commit anterior

. git stash: guardar modificações no projeto para commitar depois   
  git stash list: mostra listas de stash feitos   
  git stash apply: volta com arquivos guardados em, mas mantem ele na lista de stashs

  git stash pop: volta com arquivos guardados em stash, mas remove ele da lista de stashs   
  git stash clear: limpa lista de stashs

! Nota: Conventional Comits é um padrão usado para escrever commits, melhorando o entendimento do histórico de commits de um projeto -> conventionalcommits.org/en/v1.0.0/

. git log: mostrar histórico de commits   
  git log --oneline: mostrar histórico de commits de maneira simplificada   
  git log --pretty=format:’%h’: git logsomente com parâmetro passado (substituir o %h)

! Nota: é possível alterar a cor dos parâmetros listados no comando git log --pretty=format, para isso, usamos “%C(blue)” para fazer com que todos os dados que vierem depois dessa configuração sejam mostrados na cor azul. Também podemos usar mais de um parâmetro por vez, como “%h %d”. O parâmetro “%H” exibe a hash(acredito que seja assim que escreva) completa do commit, a hash grande. Se colocado em minúsculo(“%h”), exibe a hash de modo reduzido. O %d mostra a branch que está sendo mexida. O %s mostra a mensagem do commit. O %cn mostra o nome da pessoa que fez o commit e o %cr mostra a data que o commit foi feito.

. git tag nomeDaTag: criar tag para marcar um local no histórico do commits   
  git tag -d nomeDaTag: excluir tag   
  git tag nomeDaTag -m “anotação sobre a tag”: criar tag com anotação   
  git tag -a “nomeDaTag” -m “anotação da tag” idDoCommit: criar tag em commit existente

Se usarmos o git push origin master, as tags não sobem nenhuma tag, envia apenas as commits. Para que a tag seja enviada, é necessário que coloque: git push origin master –follow-tags

Para que a tag seja removida do servidor, é necessário usar o git push –delete “nome da tag”

Nota: Para que não seja necessário que sempre coloque o “--folow-tags", para que ele vá automaticamente, você pode acessar o .gitconfig, e adicionar:

[push]

FollowTags = true

Nomes e palavras que estão na cor azul clarinho são mutáveis, como exemplo de nomes criados e termos que variam

Git tad -d “nome da tag” – Remover tag local

**Remover coisa no git**

Git reset nome do repositório – Voltar um arquivo

Git reset ID soft -- Remove comitê acima do selecionado e volta pro Index

Git commit “ novo nome” --Renomear commit

Git reset HEAD~1 (cabeça da arvore ) --mixed – Remove commit e volta arquivos para Workspace direto.

Git rest (nome da arvore) --hard – Reverte todos os commit e volta os arquivos ao estado inicial.

Git revert nome da arvore – Gera commit que revert ao interior

Git revert HEAD~1-- noed-edit – Pula uma etapa no commit

Git revert HEAD~1- - no-commit -

Git checkout nome do arquivo – Esse comando desfaz a mudança feita no arquivo

Observação; O comando checkout só é aceito em arquivos já trackeados, caso o contrario não é aceito esse comando.

Ele tambem não funciona dentro do Index, nesse caso é preciso voltar ele dando um git reset.

Git checkout (ID) 0.1.beta -Permite eu acesso um código em um determinado commit, e vê como ele era no inicio

Git checkout master – Muda minha branch para o master

Git checkout -b (nome da branch) - Corrigir código como era antigamente, cria uma nova branch a partir da branch desconectada.

Git merge (nome da branch) - Aplicar mudança nova feita na branch no máster

Git clean – n - Mostra arquivos que serão removidos.

Git clean -f - Remover arquivos que estão antracked

Git clean -d – Remove arquivos dentro de pasast antracked

Git rm nome do arquivo - Remover arquivos que estão trackeados pelo git

Observação: Ele trackeia para você depois fazer o reset do arquivo se caso necessário.

Git rm -r – Remove o login dentro do arquivo que você deseja

Git rm nome do arquivo - - cached - Deixar de trackear arquivo

Git rebase -i HEAD ou master~qtd – Copia ou substitui commit

**ESTUDO GIT ( BOSON TREINAMENTOS)**

O que é Controle de Versão?

Controle de Versão (CV) trata do gerenciamento de versões de um projeto, incluindo documento, código-fonte, programas, etc.

Cada alteração nos arquivos de um projeto deve ser rastreada.

CV registra essas alterações e fornece meio de desfazê-las, se necessário.

Controle de Versão

Tarefas comuns a trabalhos de criação de conteúdo:

\*Criar

\*Salvar

\*Editar

\*Salvar novamente

\*Abrir para revisão

\*Editar novamente

\*Salvar outra vez

\*Comparar arquivos

\*Compartilhar com colegas

\*Colegas editam

\*Colegas salvam

O que é Sistema de Controle de Versão?

\*Ferramenta para realização de controle de versão efetivo.

\*Permite navegar pelas alterações com facilidade e realizar rollback para versões anteriores.

\*Permite o trabalho em grupo sobre um projeto – várias pessoas podem trabalhar em cópias dos projeto (branches) e unir suas alterações posteriormente (merge).

**GIT – Sistemas de Três Estados**

Existem três estados nos quais um código pode se encontrar:

**\*Estado Modificado**: Snapshot atual no qual o trabalho é feito no momento. Arquivos no Diretório de Trabalho.

- Arquivos adicionados, modificados ou removidos são marcados como modificados.

- Indica que um arquivo sofreu alterações.

- Indica também que estão diferentes em relação à última versão que está armazenada no banco de dados.

**\*Estado Preparado**: Os arquivos modificados são marcados em sua versão atual, pronta para ser armazenada no banco de dados. Arquivos no Stating Area.

-Arquivo é enviado para a área de preparação antes de ser consolidado.

- Os arquivos entram neste estado após uma instrução git add

- Assim, se tornam preparados para serem versionados.

**\*Estado Consolidado**: Dados são salvos no Repositório (Banco de dados) onde o histórico é armazenado. Este estado é compartilhado. Arquivos no Diretório do Git.

- Os arquivos preparados são salvos em definitivo após a execução de uma instrução git commit

- Uma nova imagem é salva no histórico, contendo o estado dos arquivos salvos.

- Arquivos não podem ser consolidados se ainda estiverem no estado modificado.

**GIT – Comandos para logar usuário, usar pastas , criar pastas, fazer commits e etc..**

**Saber Versão** = >git –version

**Cadastrar Usuário** – git config -- global user.name “ (nome)”

**Cadastrar E-mail** – git config – global email “(colocar e-mail)

**Visualizar diretório no bash** - pwd

**Mudar diretório e levar até o raiz**- cd .. pwd cd..

**Limpar Tela no GIT** - Ctrl + L

**Visualizar pasta e tambem arquivo diretório** – ls

**Visualizar informações completas** – ls -ls

**Entrar no diretório criado** – cd pasta-git

**Usar pasta e dizer ao git que é o diretório** – git init .

**Criar Arquivo direto do Git** – touch (nome do arquivo )

**Olhar status do Git** – git status

**Adicionar arquivo para rastreamento (stating)** – Dois modos

gitt add (nome do arquivo )

git add \* ( adiciona todos os arquivos presentes na pasta)

**Fazer commit (utilizando uma mensagem explicativa sobre sua alteração feita no repositório)** – git commit -m “(mensagem)!”

**Verificar log do git** – git log

**Adicionar arquivo no modo preparado** – git \*

**Sair do git** – exit

**GIT – Comando para obter ajuda**

Pedir ajuda no GIT – git help

Limpar tela – clear

Ajuda de subcomando mas específica – git help -a

Pedir ajuda mas detalhada – git gelp nome do comando

**Significado dos comandos no GIT**

“Principais comando de porcelana”

Add - Adicionar conteúdo de arquivo ao índice

Am - Aplique uma série de patches de uma caixa de correio

Archive - Crie um arquivo de arquivos de uma árvore nomeada

Bisect - Use a pesquisa binária para encontrar o compromisso que introduziu um bug

Branch - Listar, criar ou excluir filiais

Bundle - Mover objetos e árbitros por arquivo

Checkout - Trocar de ramo ou restaurar arquivos de árvore de trabalho

Cherry-pick - Aplicar as alterações introduzidas por alguns compromissos existentes

Citool - Alternativa gráfica para git-commit

Clean - Remover arquivos não rastreados da árvore de trabalho

Clone - Clone um repositório em um novo diretório

Commit - Registrar alterações no repositório

Decribe - Dê a um objeto um nome legível humano com base em um árbitro disponível

Deff - Mostrar mudanças entre compromissos, cometer e trabalhar árvore, etc.

Fetch - Baixe objetos e árbitros de outro repositório

Format-pach - Prepare patches para envio de e-mails

Gc - Limpar arquivos desnecessários e otimizar o repositório local

Gitk - O navegador do repositório Git

Grep - Imprimir linhas que correspondam a um padrão

Gui - Uma interface gráfica portátil para Git

Init - Criar um repositório git vazio ou reiniciar um já existente

Log - Mostrar registros de compromisso

Maintenace - Executar tarefas para otimizar dados do repositório do Git

Merge - Junte-se a duas ou mais histórias de desenvolvimento juntas

Mv - Mover ou renomear um arquivo, um diretório ou um symlink

Notes - Adicionar ou inspecionar notas de objeto

Pull - Buscar e integrar com outro repositório ou uma filial local

Push - Atualizar árbitros remotos junto com objetos associados

Range-diff - Compare duas faixas de compromisso (por exemplo, duas versões de um ramo)

Rebase - Reaplicar se compromete em cima de outra dica base

Reset - Redefinir o HEAD atual para o estado especificado

Restore - Restaurar arquivos de árvore de trabalho

Revert - Reverter alguns compromissos existentes

Rm -Remover arquivos da árvore de trabalho e do índice

Shortlog -Resumir a saída 'git log'

Show -Mostrar vários tipos de objetos

Sparse-checkout -Inicializar e modificar o checkout esparso

Stash -Esconder as mudanças em um diretório de trabalho sujo afastado

Status -Mostre o status da árvore de trabalho

Submodule -Inicializar, atualizar ou inspecionar submodulos

Stiwch - Trocar de ramo

Tag -Criar, listar, excluir ou verificar um objeto de marca assinado com o GPG

Worktree - Gerenciar várias árvores de trabalho

**Comandos auxiliares / Manipuladores**

config -Obter e definir opções de repositório ou global

fast-export -Git exportador de dados

fast-import - Backend para importadores de dados Git rápidos

filter-branch -Reescrever filiais

mergetool-Executar ferramentas de resolução de conflitos de fusão para resolver conflitos de fusão

pack-refs - Embalar cabeças e etiquetas para acesso eficiente ao repositório

prune -Podar todos os objetos inalcançáveis do banco de dados de objetos

reflog -Gerenciar informações de reflog

remote -Gerenciar conjunto de repositórios rastreados

repack -Embalar objetos descompactados em um repositório

replace -Criar, listar, excluir árbitros para substituir objetos

**Comandos Auxiliares / Interrogadores**

Annonate -Anotar linhas de arquivo com informações de compromisso

Blame -Mostrar qual revisão e autor última modificação cada linha de um arquivo

Bugreport -Coletar informações para que o usuário faça um relatório de bugs

Count-objects -Conte o número de objetos não embalados e seu consumo de disco

Difftool -Mostrar alterações usando ferramentas diff comuns

Fsck -Verifica a conectividade e validade dos objetos no banco de dados

Gitweb -Interface web do Git (frontend web para repositórios de Git)

Help -Exibir informações de ajuda sobre o Git

Instaweb -Navegue instantaneamente no repositório de trabalho em gitweb

Merge-tree -Mostrar fusão a três sem tocar no índice

Rerere - Reaproveitamento da resolução registrada de mesclagens conflitantes

Show-branch -Mostrar filiais e seus compromissos

Verify-commit -Verifique a assinatura do GPG dos commits

Verify-tag -Verifique a assinatura gpg das tags

Whatchanged -Mostrar logs com diferença que cada compromisso introduz

**Interagindo com os outros**

Archimport -Importar um repositório de arco GNU em Git

Cvsexportcommit -Exporte um único compromisso com um checkout cvs

Cvsimport -Salvar seus dados de outro SCM que as pessoas adoram odiar

Cvssever -Um emulador de servidor CVS para Git

Imap-send - Enviar uma coleção de patches de stdin para uma pasta IMAP

P4 - Importar e submeter-se aos repositórios perforce

Quiltimport - Aplica um patchset de colcha no ramo atual

Request-pull -Gera um resumo das alterações pendentes

Send-email -Envie uma coleção de patches como e-mails

Svn -Operação bidirecional entre um repositório de subversão e git

**Comandos de baixo nível / Manipuladores**

Apply -Aplique um patch em arquivos e/ou no índice

Checkout-index -Copiar arquivos do índice para a árvore de trabalho

Commit-graph -Escrever e verificar arquivos de grafia de git

Commit-tree -Criar um novo objeto de compromisso

Hash-object -Computar o ID do objeto e, opcionalmente, criar uma bolha a partir de um arquivo

Index-pach -Construir arquivo de índice de pacote para um arquivo embalado existente

Merge-file -Executar uma fusão de arquivos de três vias

Merge-index -Executar uma fusão para arquivos que precisam de fusão

Mktrag -Cria um objeto de tag com validação extra

Mktree -Construir um objeto-árvore a partir de texto formatado de árvore ls

Multi-pack-index -Escreva e verifique os índices de vários pacotes

Pack-objects -Crie um arquivo embalado de objetos

Prune-packed -Remova objetos extras que já estão em arquivos de embalagem

Read-tree -Lê informações sobre árvores no índice

Symbolic-ref -Leia, modifique e exclua árbitros simbólicos

Unpack-objects - Desempacotar objetos de um arquivo embalado

Update-index -Registre o conteúdo do arquivo na árvore de trabalho para o índice

Update-ref -Atualizar o nome do objeto armazenado em um árbitro com segurança

Write-tree -Criar um objeto de árvore a partir do índice atual

**Repositórios**

É um local de armazenamento onde o projeto e suas alterações são mantidos.

Na prática, é uma pasta comum no sistema.

Para cada projeto gerenciado pelo Git, é um repositório é configurado.

Para cada projeto nov, seguimos o seguintes passos:

1. Criar um diretório para armazenar o projeto
2. Entrar no diretório (opcional)
3. Inicializar o repositório Git

**Diretórios**

É criado um subdiretório oculto .git, que guardará os dados de alteração e snapshots.

A área fora do diretório .git é o Diretório de Trabalho, onde ficam armazenados os arquivos que você irá manipular. Ou seja, é a área onde interagimos diretamente com os arquivos do projeto.

**Staging Area**

Local para onde os arquivos vão antes de um snapshot ser criado.

Somente os arquivos da staging área vão para um snapshot.

Selecionamos os arquivos desejados no diretório de trabalho.

**Commits**

Snapshot do projeto em um determinado momento, com informações sobre o autor do conteúdo e quem o realizou .

O estado anterios do projeto é denominado “pai”. Commits são ligados entre si por conexões pai-filho.

O conjunto de commits relacionados entre si por paternidade é chamado de branch (ramo).

Um commit pode ter dois pais, se for criado pela (mescla) de dois branches.

**Identificação do Commit**

Identificamos um commit pelo seu nome, que é uma string de 40 caracteres obtida pelo hash (SHA1) do commit.

**Exercício Passo-a- passo**

**\*Criar um novo repositório, de nome exercício01. (iniciar o repositório!)**

1º Criar diretório no GIT – Comando : mkdir nome do diretório

2º Verificar se diretório deu certo – Comando: ls

3º Entrar no diretório – Comando: cd nome do diretório

4º Mostrar caminho completo até o diretório – Comando: pwd

5º Inicializar transformar um diretório comum em um repositório no GIT – Comando: git init

6º Verificar diretório oculto – Comando: ls -a

7º Criando um arquivo e escrevendo com o editor notepad++ - Comando: start notepad++ nome do arquivo

8º Colocar arquivo na stating área – Comando: git add nome do arquivo

9º Verificar se o arquivo foi adicionado – Comando: git status

10º Dar um commit ou seja colocar mensagem sobre o projeto – Comando: git commit -m “nome da mensagem”

11º Criar um arquivo vazio sem o editor – Comando: touch nome do arquivo

12º Mostra arquivo com os dados – Comando: ls -l

13º Adicionar todos os arquivo na stating área de uma vez – Comando: git add \*

14º Renomear arquivo – Comando: mv nome do arquivo nome novo do arquivo

15º Adicionar arquivo com sua extensão – Comando: git add nome da extensão , ex: txt, e outros.

16º Tirar arquivo da stating área – Comando: git rm –cached nome arquivo

17º Sair do diretório – Comando: cd ..

18 º Excluir toda pasta do diretório – rm -rf nome da pasta

\*Criar um arquivo de nome projeto.txt no diretório, e digitar algum texto nele.

\*Colocar o arquivo na staging área.

\*Efetuar commit no projeto, acrescentando uma mensagem curta.

\*Criar dois novos arquivos, de nomes tarefa.txt e pendencias.txt.

\*Colocá-los na staging área e comita o projeto.

\*Renomear o arquivo pendencias.txt para restante.txt

\*Adicionar texto ao arquivo tarefa.txt.

\*Verificar o status do diretório.

\*Colocar na área de stage os arquivos restante.txt e tarefa.txt.

\*Tirar o arquivo tarefa.txt da staging área.

\*Efetuar commit no projeto.

\*Verificar o status do diretório.

1º Criar diretório no GIT – Comando : mkdir nome do diretório

2º Verificar se diretório deu certo – Comando: ls

3º Entrar no diretório – Comando: cd nome do diretório

**Estudo GIT – João e Cayo**

**Comandos Essenciais**

Vê e-mail e usuário que está usando – git config - - list

Iniciar repositório Local – git init

Trazer repositório existente localmente – git clone

Juntar Repositório Remoto com Repósitorio Local – git remote add origin URL

Limpar seu terminal GIT – clear

Clonar meu repositório Remoto no repositório local – git clone URL

Clonar meu repositório sendo numa branch diferente – git clone -b <nome da branch> <URL>

Trazer o que tenho no meu remoto para o meu repositório local sem clonar – git pull origin máster

Saber o que esta acontecendo no seu git – git status

Para rastrear os arquivos não rastreados – git add \* = todos os arquivos ou git add <nome do arquivo>

Comitar os arquivos quer dizer que meu arquivo ou código modificado no arquivo esta salvando – git commit -m “comentário”

Mandar o que eu alterei no repositório local para o repositório remoto – git pull origin máster

Criar branch localmente - git checkout -b “nome da nova branch”

Mudar de branch – git checkout “nome da branch”

Arquivos que não precisa subir para o git , cria-se uma extensão - .gitignore

\*Git FLow é o fluxo de branchs

\*Git fork é vamos dizer um clone no repositório de outra pessoa feita no git remote, dai você faz uma alteração e pra que a pessoa aceite você envia um pull request pra vê se ela aceita.

Rodar projeto sem o ReadME – git push -u origin máster

Adicionar ReadMe no terminal – git add “ReadMe.md”

Observações:

\*README de um projeto é preenchido pelo um analista de negócios.

\*Sempre usar o Origin , comando padrão para identificar o repositório remote;

\* Git Pull – Trazer

\*Untracked diz arquivos não rastreados;

\*Após alterar um arquivo, novamente tem que rastreá-lo;

\* O git só salva uma alteração no código se você comitar.

\*Colocar na mensagem do comitt , algo objetivo e descritivo do que você está fazendo.

\*Git Push – Mandar ao remoto

\*Não é necessário sempre da o push após o commit, mas sim após terminar o seu objetivo.

\* Um projeto pode ter mas de uma branch , pois branch é a arvore de tipos diferentes tipos, onde cada pessoa vai ser designada a uma branch.

\*Quando cria uma branch localmente, automaticamente a ferramenta git troca a branch para criada.

\*Não usar espaços e também não usar acentos.

\*Master é a branch já para a produção.

\*Pull requast é de suma importância , para que após avaliação, se git aprovado pode subir para a produção.

\*Develop branch criada da master

\*Release branch de desenvolvimento

\*feature branch criada da develop para alterações

\* Fluxo = Master -> Develop -> feature -> Develop -> release -> máster

\* Hotfixes bug na máster

\* Se for mudar de git hub para o lab é necessário a apagar as credenciais no seu computador, em Gerenciar credências.

**Estudo Scrum – Unipe**

**Exercício - Adicionar um commit do GIT HUB a pasta local (GIT PULL) e Enviar arquivo na pasta local para o GIT HUB (GIT PUSH)**

1º Abro o GIT Dash na pasta criada localmente.

2º Inicio o GIT – Comando: git init

3º Puxar o commit criado no GIT HUB – Comando: git pull remote origin master

4º Adicionar pasta no git dash – Comando: git add “nome da pasta local”

5º Commitar pasta adicionada – Comando: git commit -m “nome da pasta local”

6º Enviar pasta local para o repositório no GIT HUB – Comando: git push remote origin master