Sistema de versionamento de código distribuído criado por Linus Torvalds.

Git e Github são tecnologias diferentes, mas que normalmente são utilizadas juntas no mercado.

Git – open source.

Github – repositório online onde vamos armazenar nosso código.

1. Controle de versão;
2. Armazenamento em nuvem – no repositório do Github usando a infraestrutura da Microsoft;
3. Trabalho em equipe;
4. Melhorar o código – expondo o código para o mundo;
5. Reconhecimento – espécie de rede social.

Interagimos com o software do git através do terminal. Ele é um **CLI** com uma Learn Interface, utilizado por guia de comando, diferente do **GUI** Graphic User Interface, que possui uma interface gráfica.

1. Mudar pastas
2. Listar as pastas
3. Criar pastas/arquivos
4. Deletar pastas/arquivos

Shell vs Bash

Linux

Linux: cd(change directory) Is mkdir rm -rf

ls 🡪 lista de diretórios

cd etc (pasta do sistema operacional – disco c no windows)

cd .. 🡪 retroceder nível de navegação

clear 🡪limpar tela (ctrl+L)

ex: cd etc 🡪 ls

mkdir workspace – criar diretório

echo hello > hello.txt 🡪 ls 🡪 cd .. 🡪 rm -rf workspace/ (remove reinforce workspace – **deletar pasta**)

Windows

Windows: **cd** (change directory) **dir** **mkdir** **del/rmdir**

dir 🡪 cd / (a / indica um caminho que vai para a base do diretório C)🡪 dir (mostrar as pastas presentes na pasta do diretório C) 🡪 cd Windows (para entrar na pasta Windows) 🡪 cd .. (para voltar).

**cls** (Clear Screen)

**mkdir** workspace – criar pasta com nome workspace

e

Silence On Sucess – ele cria uma pasta com mkdir workspace, apenas cria e não retorna uma mensagem. SOS

**del** workspace – excluir arquivos dentro da pasta workspace

**echo hello arquivo > hello.txt** – criar arquivo com nome hello

**rmkdir nomeDaPasta /S /Q** – Remove Directory ou remover **pasta** workspace.

Como o GIT funciona

Porque o GIT é um sistema tão seguro? Porque o GIT é um sistema distribuído?

Temos nosso código hosteado por um servidor na nuvem (GITHUB) com a versão mais recente e atualizada e, se houver 30 pessoa que fazem manutenção do código de forma regular, além da versão do servidor, cada maintainner possui uma versão do código confiável por causa da estrutura que o GIT mantém e foi projetado.

SHA1

Security Hash Algorithm. Algoritmo de criptação desenvolvido pela NSA, Nacional Security dos EUA. Gera um conjunto de 40 dígitos único. É uma forma curta de representar um arquivo. Comando: openssl sha1 texto.txt

Objetos fundamentais:

São três tipos básicos de objetos do GIT responsáveis pelo versionamento do código.

BLOBS, TREES e COMMITS.

Sistema distribuído

Segurança

BLOB

BLOB é o sha1 dos arquivos. Os arquivos ficam guardados dentro de um **objeto** chamado **BLOB** que contém metadados dentro dele. O objeto BLOB tem um tipo, um tamanho (da String do arquivo), \0 e o conteúdo:

1. **Passa a String echo pela função git hash 🡪 em seguida ele mostra o sha1 gerado;**
2. **Usa o openssl com o sha1 para passar os metadados para a String**

O GIT guarda os arquivos gerando a encriptação deles e armazena metadados do GIT, que são o tipo do objeto, o tamanho do arquivo, tamanho da String, entre outros...

TREES

Armazena e aponta para tipos de BLOBS diferentes ou outras árvores, porque os diretórios do sistema operacional podem conter outros diretórios então o GIT aponta para outras árvores como um elemento recursivo. Contém metadados (possui **\0**) e aponta para BLOBS que também contém o sha1.

O BLOB não guarda o nome do arquivo na TREE. Aponta para um BLOB que possui um sha1 e **guarda o sha** do arquivo.

Se mudar uma vírgula em um arquivo na qual a árvore está apontando, quando ela gerar um sha1 dos metadados, o sha da bolha vai ter mudado e, consequentemente, vai mudar o sha da árvore.

**COMMIT**

**É o objeto que dá sentido para a alteração que estamos fazendo.**

Escrevemos uma mensagem para o COMMIT que dá significado para os arquivos, esse objeto leva o nome do autor e timestamp (carimbo de tempo), que leva o nome do autor, data e mensagem de quando ele foi criado.

**Os COMMITS também possuem um sha1, encriptação e geração do hash de 40 caracteres identificador de seus metadados.** Se alterarmos uma blob que está sendo apontada para uma árvore, que pode estar sendo apontada por outras árvores e, se mudarmos um arquivo a mudança se reflete no COMMIT e **é mostrado uma linha do tempo de quais commits foram realizados** mostrando que o código no código mostrando que ele não teve interferência.

**Aula: Chaves SSH e Tokens**

Comandos básicos de interação do GIT com na máquina local para o CLI.

Chave SSH: forma de estabelecer uma conexão segura e encriptada entre 2 máquinas, se conectando com servidor do GITHUB e estabelecendo nossa máquina local como uma máquina confiável para o GITHUB. Serão geradas 2 chaves, uma pública, que será colocada no GITHUB e uma privada.

**CHAVE PÚBLICA**

ssh-keygen -t ed25519 -C [luizmrv@outlook.com](mailto:luizmrventura@outlook.com) 🡪 apertar enter 🡪 Escrever uma senha 🡪 Reescrever a senha 🡪 **cd /c/Users/luizmrv/.ssh/** 🡪 ls 🡪 **cat id\_ed25519.pub** (para poder visualizar o conteúdo URL da chave pública) 🡪 pwd 🡪 eval $(ssh-agent -s) //gera o Agent pid 987, esse número final varia, ele ta startando um projeto e o ssh-agent rodando de fundo nas threads do computador. 🡪 ssh-add id\_ed25519 🡪 enter 🡪 Digitar senha (passphrase) Coala2805

**PASSOS: 1. GERAR CHAVE; 2. NAVEGAR ATE A PASTA E PEGAR O NOME DA CHAVE; 3.COLOCAR NO GITHUB; 4. VOLTAR PARA O CLI E PASSAR COM O ssh-add id\_.pub;**

**Para clonar um repositório:**

1. Ir em um projeto hitbub em <> Code, copiar o link SSH

2. Ir no \*\*cli\*\* e digitar: **git clone URL**

3. Want to continue connecting, digitar: yes

🡪 Em seguida, ele clona

🡪Em seguida ele **clona o repositório,** se dermos **ls** podemos ver o link do repositório que nós não teríamos.

RESUMINDO:

1. Gerar Chave

2. Navegar até a pasta e pegar o nome/conteúdo da chave

3. Colocar no GITHUB

4. Voltar

**Token de acesso pessoal: ghp\_7TNuLWZadsYHhXj2SkLgIX0wSXOTmc4WeObC**

Geramos um token no github e sempre que formos fazer um COMMIT o GIT vai pedir o usuário e, na senha, utilizamos nosso TOKEN DE ACESSO PESSOAL. Podemos configurar esse arquivo como uma chave SSH se tivermos uma máquina de confiança.

1. Ir em um repositório do GITHUB em <> Code
2. Ir na pasta ( cd /c/Users/luizmrv/.ssh/ ) e digitar: git clone colarURLqueQuerClonar
3. **Colar o token e dar Sign in**
4. Repositório é baixado

Anotações:

ed25519 (tipo de criptografia da chave)

Enter passphrase - Coala2805

/c/Users/luizm/.ssh/id\_ed25519 already exists.

$ ls

id\_ed25519 id\_ed25519.pub

id\_ed25519.pub --> chave pública que vamos utilizar no github

ED25519 256 - Nome da criptografia

SHA256 - Tipo de criptografia usada

Inicializar o SSH agent (entidade que fica encarregada de pegar as chaves e lidar

com elas). - eval $(ssh-agent -s)

**Aula: Iniciando um COMMIT**

Primeiros comandos com GIT

1. Iniciar o GIT
2. Iniciar versionamento
3. Criar um commit

Para iniciar repositório: git init

Para mover arquivos e começar a dar início ao versionamento: git add

Para criar o promeiro commit: git commit

**CADASTRAR EMAIL E NAME**

cd workspace 🡪 mkdir livro-receitas 🡪 cd livro-receitas 🡪 **git init** 🡪 ls -a (-a para mostrar as pastas ocultas de gerenciamento do GIT, apenas ls não vai mostrar) 🡪 cd .git/ (vai entrar na pasta e mostrar a estrutura que tem dentro dela e a pasta \_objects é onde ficara os objetos) 🡪 cd .. 🡪 **git config --global user.email “luizmrv@outlook.com” 🡪 git config --global user.name Matt**

**Na primeira vez que usar o GIT** ele pede algumas configurações, um **username e email,** para que quando for gerado um commit ele tenha um autor atrelado a ele. **git config --global**

Depois de criar um arquivo na pasta executar no cli: **git add \* 🡪 git commit -m ”commit inicial”**

[master (root-commit) **40b25c9**] commit inicial 🡪 o que está em **negrito** é o início do sha1

**Relembrando:** São os objetos do GIT que dão significados as alterações e esses objetos carregam em si uma mensagem de texto com outros metadados como autor, hora e etc, essa mensagem é o **“commit inicial”**. Assim, depois, quando tivermos acesso à lista de alterações, conseguiremos facilmente identificar o arquivo que adicionamos, o commit que a gente fez e acesso à essa mensagem curta para conseguirmos entender o que o commit significa dentro das nossas alterações e projetos.

**Aula: Ciclo de vida dos arquivos no Git**

Quando usamos o git init, de fato estamos de fato criando um repositório no git, dentro daquele diretório ou pasta.

Tracked x untracked

Quando criamos um arquivo, ele está **untracked**, usamos o **git add \*** que fez com que o git movesse o arquivo **untracked para staged** (esperando para entrar no palco).

Untracked 🡪 unmodified 🡪 modified 🡪 staged . A cada vez que rodamos o git add no arquivo ele muda de área/grupo (‘status’).

Se o arquivo não tinha modificação e removemos ele, ele volta para untracked.

Quando ele entra na área **Staged, ele fica pronto para o commit**. **O commit retorna os arquivos para unmodified, para recomeçar o ciclo, após salvar o arquivo** (ou snapshot do arquivo) no commit.

Os arquivos vão ficar alterando sempre entre Working Directory and Staging Area na medida em que vamos adicionando novos arquivos e modificando no repositório. Quando fazemos o commit ele passa a se integrar com o repositório local que pode se integrar para um repositório remoto (nuvem).

**Commit:** Move os arquivos de **Staging Area 🡪 Unmodified,** **depois de ter criado uma cópia/snapshot para popular o repositório local** (**Local Repository), composto por commits.** Se o repositório local não estiver ‘commitado’ não temos como mandar o arquivo para um repositório remoto (GITHUB).

Prática:

Abrir cli na pasta 🡪 ls 🡪 git status 🡪 mkdir receitas 🡪 ls 🡪 **mv** strogonoff.md **/.receitas/ 🡪** ls 🡪 cd .. 🡪 git add strogonoff.md.txt **receitas/** (receitas é a pasta nova onde o arquivo foi colocado dentro) 🡪 git status (foi mudado para staged – “changes to be commited”) 🡪 **git commit -m “cria pasta receitas, move arquivo para receitas” 🡪 git status (NOTHING TO COMMIT - )**

1.git add \* 🡪 2.git commit -m “mensagem” 🡪 1.Pega os arquivos modificados e move para STAGED; 2. Pega todos os arquivos em STAGED e envelopa em uma mensagem, dando significância a eles, criando um objeto chamado COMMIT 🡪 Revisão, na pasta livro-receitas: echo > README.md 🡪 ls 🡪 git status 🡪 **git add \* 🡪 ls 🡪 git status 🡪 git commit -m “adiciona index”**

**git init**

**git add \* : Working Directory 🡪 Staging Area (MOVE PARA SER COMMITADO)**

**git commit -m “msg”: Staging Area 🡪 Local Repository**

**git push origin master**

**git pull origin master 🡪** executar e ver onde foi o conflito CONFLICT (content): ... 🡪 editar o arquivo de == onde foram feitas alterações que não estão presentes no Local Repository 🡪 git add \* 🡪 git commit -m “resolve conflitos” 🡪 **git push origin master**

**Aula: Trabalhando com o GITHUB**

Local repositor 🡪 Remote Repository (GITHUB)

cli: git config --list

reconfigurar email: git config --global --unset user.email 🡪 git config --global --unset user.name 🡪 git config --list

**Temos que apontar nosso repositório local para o repositório do github, usando a URL.git** em “...or push na existing repository from the command line”

git remote origin

git remote -v 🡪 comando para listar as listas de repositórios remotos cadastrados.

git push origin master 🡪 origin = apelido; master = apelido que demos para o link no repositório do github

**No GITHUB, em commits, podemos ver o histórico dos commits e, se clicarmos no sha, podemos ver o conteúdo contido no commit trackeado pelas BLOBS.**

**Como baixar um repositório**

< > Code 🡪 Copia o link 🡪 git clone url 🡪 ls 🡪 cd cpython/ 🡪 ls -a (mostra a pasta .git) 🡪 git remote -v (mostra os repositórios remotos na qual o repositórios estão apontados).

**git add status 🡪 git add . 🡪 git status 🡪 git commit -m “Anotações”**