GIT e GitHub

1. **História: GIT** foi criado por **Linus Torvalds** também criador do Linux. Devido à quebra da parceria entre a *"Bitkeeper"* e o Linux surgiu a necessidade de se criar um sistema de versionamento e repositório para o Linux, assim sendo, Linus criou o GIT e implantou algumas melhorias em seu sistema.

    - **Controle de versão:** O GIT é mais simples de voltar em versões passadas de arquivos pois seu sistema de salvamento é diferente. Os outros métodos salvam a diferença entre as versões de arquivo. O GIT trabalha com a integra das versões anteriores.
    - **Velocidade**
    - **Design simples**
    - **Suporte robusto a desenvolvimento não linear (milhares de branches paralelas)**
    - **Capaz de lidar com grandes projetos de maneira eficiente.**

2. **GitHub:** É um serviço de web compartilhado para projetos que utilizam o GIT para versionamento, ou seja, GitHub é um local na web/nuvem que armazena os projetos versionados em GIT.

3. **Configuração inicial do GIT:3**

    - **Criando usuário: git config –global user.name** "Nome"
    - **Adicionando e-mail de usuário: git config –global user.email** "e-mail"
    - **Definindo o editor padrão: git config –global core.editor code**
    - **Criação de pasta: mkdir nome-da-pasta**
    - **Acesso a nova pasta: cd nome-da-pasta/**
    - **Inicialização do repositório: git init**
    - **Retorno ao diretório inicial: cd..**

4. **Ciclo de vida dos arquivos GIT:**

    - **Untracked:** Arquivo que acabou de ser criado e o GIT ainda não o conhece.
    - **Unmodified:** Arquivo adicionado ao GIT porém ainda não modificado.
    - **Modified:** Arquivo já modificado e reconhecido pelo GIT.
    - **Staged:** Arquivo sofre versionamento.

5. **Fluxograma de processos GIT:**

    - **Add File –** adiciona o arquivo
    - **Edit File –** edição de arquivo
    - **Stage File –** etapa onde o arquivo fica pronto para o snapshot/versionamento.
    - **Commit –** Versionamento do arquivo, adicionado ao log com seu identificador.
    - **Remove the file –** Retorna ao "status 0" e deve-se adicionar novo arquivo/versão.

6. **Lista de comandos para o terminal do GIT:**

    - **git status – Comando base que mostra a situação atual do GIT.**
    - **git add – Adiciona arquivo para o próximo processo em GIT.**

- **git commit –m “Descrição do que foi editado/Implantado”** – versionamento (Utilizado caso se tenha feito o git add no arquivo editável anteriormente)
- **git commit –am “Descrição do que foi editado/Implantado”** – versionamento (Utilizado caso se deseje ir direto da etapa de edição para a etapa de final sem utilizar git add)
- **git log –** Detalhes do que aconteceu no GIT
- **git log –decorate –** Mais informações
- **git log –author “Nome ou iniciais”** ex: “Lucas Santiago”, “Lucas”, “Lu”, busca toda movimentação e edição realizada pelo “autor” author.
- **git shortlog** (mostra autores modificações e quais commits fez)
- **git shortlog –sm** (Lista autores e quantos commit o autor fez)
- **git log –graph** (mostra acompanhamento de situação através de linguagem gráfica)
- **git show tag/hash/numero** (Mostra os detalhes do arquivo correspondente a tag, hash ou numero)
- **git diff** (utilizado antes de realizar o commit para ver as diferenças entre o arquivo atual e o arquivo na versão anterior)
- **git diff –name-only** (Mostra somente o nome do arquivo que está sendo editado)

7. Mudanças e cancelamento de modificações

- **git reset (soft, mixed e hard) -** São utilizados quando já foi realizado o commit do arquivo, desta forma, estes três comandos podem ser utilizados.
- **git reset –soft** (volta o arquivo para etapa de Staged, já modificado e pronto para ser commitado)
- **git reset –mixed** (volta o arquivo para etapa de edição, ainda modificado porém não chegou a etapa de staged)
- **git reset –hard** (o arquivo volta 100% à versão anterior, porém é necessário tomar cuidado ao utilizar este método pois pode gerar problemas no histórico?)
- **git revert hash/tag/numero** (utilizado quando já se fez o commit porém não se deseja perder os dados do arquivo. Desta forma não se perde o histórico e possível atualizar o arquivo e realizar novamente o commit.)

Sintaxe: **git reset –soft/mixed/hard + identificador do arquivo anterior**.

Exemplo: git reset –soft a88f66a571aac3fc104cca1121a1c028cfb91c2c

- **git checkout nome-do-arquivo** (Utilizado quando o arquivo ainda se encontra na etapa de edição, volta o arquivo para o original antes da edição realizada)

8. Repositório remoto (GitHUB):

- ssh-keygen -t rsa -b 4096 -C "seu_email@exemplo.com" - **(1º passo, colar dentro do terminal)**
- Enter a file in which to save the key (/c/Users/*you*/.ssh/id_rsa) : [Press enter]
  (2ºpasso, só apartar enter e continuar)
- Enter passphrase (empty for no passphrase): [Type a passphrase]
  Enter same passphrase again: [Type passphrase again]
  (3º passo, pode colocar um password, mas normalmente só dar enter)
- **4º passo -** acessar**: cd ~/.ssh/** e digitar **ls** (mostra os arquivos)
- **5º passo –** pegar a chave: **cat id_rsa.pub**

- **6º passo –** abrir o arquivo e copiar o código: **code id_rsa.pub** (o comando code abre o arquivo no VScode, cada programa tem o seu prefixo.)

- **7º passo –** acessar conta do GitHUB: em configurações **colar a chave** no menu **"SSH and GPG key. "** E inserir um título (nome da máquina, ou qualquer um a escolha)

- **8º passo –** criar novo repositório no GitHUB**: no site** criar um **novo repositório** para receber os arquivos.

- **9º passo** – dentro da pasta desejada utilize 1: **git init** (caso não tenha iniciado o repositório ainda)**,** 2: faça o **commit de um arquivo** para subir).

- **10º passo** – copiar do site do GitHUB na tela de criação de repositório: **git remote add origin git@github.com:LucasSGomide/brincadeira.git**

- **11º passo** – enviar os arquivos para o repositório online: **git push -u origin master**

9. **Comandos git remote:**

   - **git remote** (mostra se já existe ou não o repositório)
   - **git remote –v** (mostra mais algumas informações)
   - **git push origin** (nome padrão) **master** (branch)

10. **Git Clone:**
    - O comando **git clone** permite que você copie qualquer repositório open source encontrado no GitHUB, e ele cria automaticamente uma pasta dentro do diretório que o terminal se encontra. Este repositório pode ser editado, modificado, etc. Porém, não é possível subi-lo para o GitHUB.
    - **Sintaxe: git clone git@github.com:LucasSGomide/projeto-calculadora-macros.git** (o link pode ser encontrado na aba download dentro do repositório no site do GitHUB.)

11. **Fork:**

    - Este comando permite clonar um repositório que não é seu, e ajusta-lo, ou fazer modificações, porém para subi-lo novamente para o GitHUB você deve fazer um **request** para o dono do arquivo.

12. **Branches:**

    - Um branch é um ponteiro móvel que leva um commit. Em outras palavras é uma espécie de aba em que um commit se encontra, o branch default é o master.

    - O branch te possibilita modificar o arquivo sem alterar o local principal (master), ele é facilmente desativado, permite que múltiplas pessoas trabalhem simultaneamente em diferentes branches e evita conflitos.

13. **Comandos dos branches:**

- **git checkout –b nome_do_branch** (cria um novo branch)
- **git branch** (mostra em qual branch você está trabalhando)
- **git checkout nome_do_branch** (muda de branch)
- **git branch –d nome_do_branch** (deleta o branch)

14. União de branches (merge):

- Utilizado normalmente quando é necessário mostrar no histórico (log) que foi realizado uma união de branches no arquivo. (Geralmente o último arquivo do programa)
- O merge funciona criando um novo commit no final do arquivo e fechando um "loop" assim sendo o arquivo unificado será expressado em um novo commit.
- **Vantagens:** não apaga os commits criados anteriormente
- **Desvantagens:** cria um commit extra e deixa o histórico poluído.
- **git merge nome_do_branch** (comando para realizar o merge)
- Para entender melhor pesquisar.

15. União de branches (rebase):

- É a forma de união utilizada com maior frequência, ele funciona enviando todos os commits que estão "desalinhados" ou fora do branch master para último da fila e realiza a união dos arquivos sem criar um novo commit.
- **Vantagens:** evita criação de commits extras, e mantém o histórico linear.
- **Desvantagens:** perde-se a ordem cronológica dos commits.
- **git rebase nome_do_branch**
- **Para entender melhor praticar e pesquisar.**

16. Extras:

- **gitignore** (utilizado para ignorar arquivos que não podem ser expostos ao público na hora de subir para o GitHUB, ex: senhas)
- Para fazer a utilização deve-se criar um novo arquivo **code .gitignore** e adicionar quais arquivos ou extensões ele deve ignorar. Ex: *.json (ignora todos os arquivos .json / db.xls ignora o arquivo db especificamente)
- No site do GitHUB existe uma database com arquivos padrão .gitignore de diversos tipos de linguagem e tipos de projeto. Github.com/github/gitignore

- **Git Stash** (utilizado quando não se quer commitar um arquivo porém é desejável trabalhar nele depois)
- **git stash nome_do_arquivo** (cria um stash)
- **git stash apply** (retorna os arquivos)
- **git stash clear** (limpa todos os stash)

- **git config - - global alias.s status (atribui a letra S a função de STATUS, ou seja cria um atalho, o comando git status** passa a ser **git s)**
- **git config - - list** (mostra todas as configurações, inclusive os atalhos criados)

- **git tag –a 1.0.0 –m “Criando tag 1”** (comando utilizado para criar TAG de versão no git, estas TAGs sobem para o github)

- **git push origin master - - tags** (sobe todas as TAGS criadas para o GitHUB)
- **git tag –d 1.0.0** (deleta a TAG 1.0.0)
- **git push origin: 1.0.0** (comando utilizado para apagar a TAG do repositório remoto)
- para utilizar o sistema de versionamento por TAGs, faz-se o commit padrão, adiciona a TAG, e por fim sobe a TAG.