**Introdução ao Git e ao GitHub**

**O que é o Git e sua importância**

**Contextualização**

Fazendo uma analogia com um jornalista/redator, considere que ele, com seu conhecimento, decide fazer uma reportagem e então escreve ela em um documento com o nome *Rascunho*. Antes de publicar, é necessário que esse documento passe por uma revisão, feita por parte do editor. O editor então revisa e pontua diversas correções.

O jornalista então volta à sua mesa, altera o nome deste arquivo (que antes era *Rascunho* e agora se chama *Antigo*), corrige esses pontos e novamente leva essas alterações ao editor.

O editor indica novas alterações a serem feitas. O jornalista então volta à sua mesa, altera o nome deste arquivo (que antes era *Antigo* e agora se chama *Antigo 2*), faz as correções necessárias e, mais uma vez, leva ao editor. Este, por sua vez, analisa e libera o arquivo, dizendo que está tudo correto. Agora esse arquivo que era um rascunho, passa a ser a *Reportagem*.

Perceba, que apesar das correções feitas, todas as alterações, pensamentos, anotações feitas nas reuniões de revisão foram mantidas nos arquivos anteriores chamados de *Antigo* e *Antigo 2,* onde se encontram de fato a essência da criação, tudo que, na visão do jornalista, é importante guardar para saber como se deu o andamento até o arquivo final.

Essa é a função do **Git**. Ele é um sistema de controle de versão distribuído, registrando o histórico de edições de qualquer tipo de arquivo.

Foi criado em 2005 por Linus Torvalds (criador do Linux), que teve alguns problemas durante a criação do sistema operacional Linux, de forma colaborativa, com pessoas espalhadas pelo mundo todo. Ele precisava de algo que pudesse criar versões do seu trabalho e que também suportasse pessoas do mundo todo trabalhando em um mesmo arquivo, às vezes, em uma mesma linha de código.

Embora hoje o Git seja uma versão de mercado, ele não é uma invenção de Linus Torvalds, pois já existia um sistema de versionamento na época. Porém, devido ao descontentamento em relação aos sistemas da época trouxe a necessidade de criar seu próprio sistema.

***Nota****:* um software sempre é feito de forma colaborativa.

Então, depois que você já tem um software que cuida dessa parte de versionamento do seu código, é preciso um local para guardá-lo. E com isso, outras empresas vieram com a solução, trazendo o **GitHub**, da Microsoft, que é uma plataforma de hospedagem de código-fonte e arquivos com controle de versão usando o Git. Ele permite que programadores, utilitários ou qualquer usuário cadastrado na plataforma contribuam em projetos privados e/ou Open Source de qualquer lugar do mundo. Por esse motivo, atualmente, os cursos unem Git e GitHub em um mesmo módulo, já que eles funcionam de forma complementar, mas diferente.

Apesar de essas ferramentas serem bastante utilizadas no mercado, elas não estão sozinhas. Assim como Git, existem outras plataformas, como: CVS, Subversion, Mercurial, Bitkeeper, entre outras. Já o GitHub tem como concorrentes: GitLab, BitBucket, entre outros.

**Benefícios de aprender essas tecnologias juntas:**

1. Controle de Versão;
2. Armazenamento em Nuvem: utilizando toda a estrutura Microsoft para armazenamento;
3. Trabalho em Equipe;
4. Melhorar seu Código;
5. Reconhecimento: como uma rede social;

**Navegação via command line interface e instalação**

**Comandos básicos para um bom desempenho no terminal**

A maioria dos softwares funcionam de forma responsiva ao usuário, dessa forma, temos o chamado GUI - Graphical User Interface ou Interface Gráfica do Usuário, que permite que os usuários usem os gráficos para interagir com um sistema operacional.

O Git, por sua vez, possui o design voltado para outro tipo de programa, sendo CLI - Command Line Interface ou Interface de Linha de Comando, não tendo uma interface gráfica.

O Terminal permite que você instrua o computador a efetuar as operações que você deseja (como copiar um arquivo, ou iniciar a execução de um programa). O Terminal é um interpretador de comandos. Cada comando é, em geral, dado em uma linha digitada no terminal.

*Nota:* Hoje existem programas que automatizam isso e acrescentam uma interface gráfica ao Git, tornando ele mais simples.

**Git Bash:** terminal extendido para otimizar o uso do Git;

**Comandos básicos:**

* Mudar de pastas;
* Listar as pastas;
* Criar pastas / arquivos;
* Deletar pastas / arquivos;

Os sistemas operacionais trazem consigo diferentes maneiras de interagir com o terminal e rodar os comandos. Isso tem a ver com a forma do terminal. O sistema Windows é derivado do *Shell* e os sistemas Unix são derivados do *Bash*. O terminal é chamado frequentemente de **linha de comando** ou **shell**. Até pouco tempo, esta era a maneira que o usuário interagia com o computador; entretanto, os usuários de Linux viram que o uso do shell pode ser mais rápido do que um método gráfico e que ainda merece algum mérito hoje.

\*shell: é a interface entre sistema operacional o usuário (você) e o núcleo do sistema(Kernel). O primeiro processo, executado automaticamente ao entrar no sistema (login) é o seu shell.

Quais são os tipos de shell?

Existem vários tipos de Shell, sendo os mais comuns o sh (chamado Bourne shell), o bash (Bourne again shell), o csh (C Shell), o Tcsh (Tenex C shell), o ksh (Korn shell) e o zsh (Zero shell). Normalmente, seus nomes correspondem ao nome do executável.

*Nota:* O windows 10 é: windows subsystem for Linux. Isso significa que ele é um sistema Linux rodando dentro do windows.

**Os comandos de navegação:**

São funções que permitem facilitações nas atividades do dia a dia, como solucionar problemas e obter informações sobre o computador e o sistema operacional. Para acessar o terminal vá no menu Iniciar e digite cmd. Uma tela preta se abrirá.

**Usuários Windows:**

* cd: command directory;
* dir: directory;
* mkdir: make directory;
* del / rmdir: remove directory;

**Usuários Unix (Linux / Apple):**

* cd: command directory;
* ls: list command;
* mkdir: make directory;
* rm -rf: remove -recursive force (não pede nenhum tipo de confirmação);

*Nota:* Todos os comandos citados acima possuem variâncias, eles possuem flags, que são complementos passados a esses comandos que acrescentam, modificam ou formatam a forma como esses comandos são devolvidos para o usuário.

**Listar pastas:**

**dir - directory:** comando de listagem de diretórios. Permite que você se situe dentro do sistema operacional e saiba em que local está. Ele traz uma lista de diretórios que existem na pasta na qual estamos situados (pasta C:\users\nome do usuário). No Linux, é utilizado o comando “ls” para isso.

**dir**: entra na pasta C:\users\nome do usuário

**Entrar e Sair de pastas:**

Tendo a lista de diretórios, é possível subir e descer de nível nesses diretórios, ir para uma pasta específica, etc. Para isso, existe o comando cd.

**cd - command directory:** é usado para trocar de um diretório para outro, ou seja, serve para você conseguir entrar em uma pasta e é igual para todos os sistemas operacionais.

cd / : te permite passar um caminho específico (exemplo: pasta C:). Mostra todas as pastas contidas em C:;

C:\> cd/Windows> entra na pasta windows;

C:\Windows>dir : te dá acesso aos diretórios desta pasta;

C:\Windows>cd .. : você retorna um nível de pasta;

**cls** - **clear screen:**  C:\>cls :Limpa a tela do terminal. Atalho Ctrl + L;

TAB - tecla autocomplete: usado para completar os nomes de pastas/diretórios, diminuindo o risco de erros de digitação;

Ex.: C:\>cd W(tab) - vai automaticamente completar Windows;

**Criar pastas e arquivos**

**mkdir**: make directory: comando usado para criar pastas. Cria-se por padrão uma pasta chamada “workspace”

Conceito: “Silence on Success” - se der tudo certo, o terminal não dirá nada;

Ao lado do nome “workspace”, utilizando o comando “echo”, você pode criar um arquivo dentro da pasta:

C:\>cd workspace (criando a pasta workspace)

C:\workspace>echo hello (o comando echo “ecoa” aquilo que escrevemos)

hello (resultado do eco)

OBS: o sinal de maior (>) se chama redirecionador de fluxo;

C:\workspace>echo hello > hello.txt (arquivo hello.txt criado)

**Deletando arquivos**

**del - delete:** comando utilizado para deletar arquivos;

C:\>del workspace (utiliza o comando del + o nome da pasta que contém o arquivo)

C:\workspace\\* Tem certeza (S/N)? s (você deve confirmar se deseja excluir tudo)

**Deletando diretórios**

**rmdir - remove directory:** remove o diretório com todos os arquivos;

C:\>rmdir workspace (utiliza o comando rmdir + o nome da pasta que contém o arquivo)

**Pasta, Diretório ou Repositório?**

**Pasta**: índice de arquivos agrupados e organizados;

**Diretório**: o diretório não contém arquivos dentro dele. Ele têm *informações* sobre a localização dos arquivos;

**Repositório**:

**Entendendo o funcionamento do Git por “baixo dos panos”**

**Tópicos fundamentais para entender o funcionamento do Git**

* **SHA1;**
* **Objetos Fundamentais;**
* **Sistema distribuído;**
* **Segurança;**

**SHA1**

A sigla SHA significa Secure Hash Algorithm (Algoritmo de Hash Seguro). É um conjunto de funções hash criptográficas, projetadas pela NSA (Agência de Segurança Nacional dos EUA).

Essa ferramenta de criptografia pega o seu arquivo, seja ele uma foto, uma imagem, uma frase, e embaralha ele de uma forma específica. Essa encriptação gera um conjunto de caracteres identificador de 40 dígitos. E esse conjunto é único, servindo como identificação.

**Ex.:** se você tem um arquivo de texto e roda esse algoritmo, a saída dele vai gerar um conjunto de caracteres de 40 dígitos.

Se você alterar uma vírgula no texto e rodar novamente, será gerado um novo conjunto identificador de 40 dígitos.

Agora, se você abrir o arquivo, desfazer a alteração feita anteriormente e rodar o arquivo novamente, o identificador gerado será igual ao primeiro.Por esse motivo ele é muito eficiente e o git usa ele como algoritmo de encriptação, permitindo identificar os arquivos de uma forma segura e rápida.

Basicamente, essa chave identificadora é uma forma curta de representar um arquivo, de representar o estado de um arquivo, pois sempre que rodamos um algoritmo, ele gera uma nova chave.

echo “ola mundo” | openssl sha1

> (stdin)= f9fc856e559b950175f2b7cd7dad61facbe58e7b

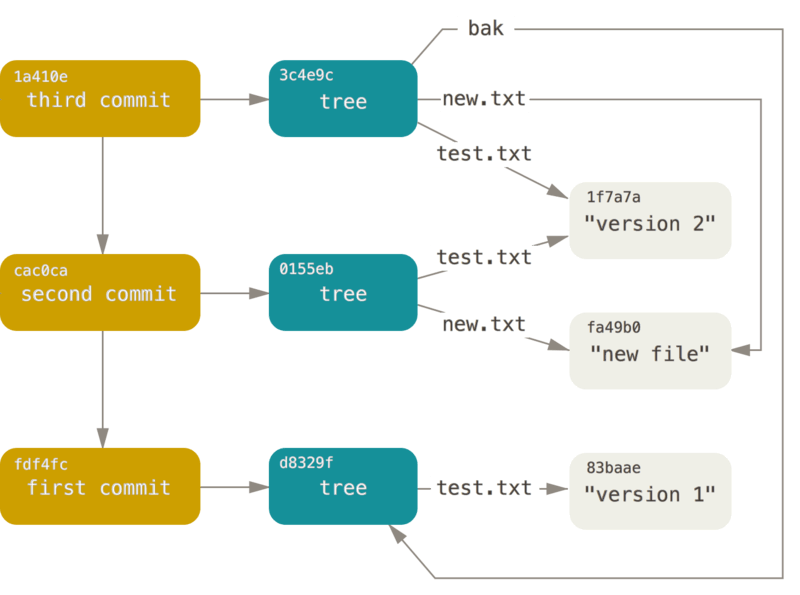
Essa é uma maneira muito eficiente do Git garantir e identificar que os arquivos sofreram modificação e identificar, também, que naquele arquivo, de uma versão para outra, ele sofreu ou não modificações garantindo que naquele arquivo tenha o conteúdo correto.

**Objetos internos do Git**

O Git é um sistema de arquivos de conteúdo endereçável. Isso significa que o coração do Git é um simples armazenamento chave-valor. Você pode inserir qualquer tipo de conteúdo nele, e ele lhe dará de volta uma chave que você pode usar para recuperar o conteúdo de volta em qualquer momento. Para demonstrar isso, você pode usar o comando de encanamento **hash-object**, que recebe alguns dados, armazena eles em seu diretório .git, e lhe devolve de volta a chave com o qual os dados são armazenados.

Os 3 objetos básicos e fundamentais do Git, responsáveis pelo versionamento do nosso código são:

* **Blobs (binary large objects)**: para arquivos binários;
* **Trees**: para pastas;
* **Commits**: para confirmações;



Fonte: https://git-scm.com/book/pt-br/v2/Funcionamento-Interno-do-Git-Objetos-do-Git

**Por que o Git é um sistema distribuído e seguro?**

Imagine que você tenha seu código, seu repositório, hosteado em um servidor (em uma nuvem, por exemplo o GitHub). Então, nesse caso, o código presente lá representa o estado final do seu código, sistema ou software, sendo a versão mais recente e atualizada.

Suponhamos que tenha 40 pessoas contribuindo nesse repositório e na máquina destas 40 pessoas (maintainers: pessoas que mantém esse código de forma regular) também tenham uma versão deste código. Então, pelo fato dos commits serem tão difíceis, até mesmo impossíveis de serem alterados, tanto a versão mais recente que está na máquina do servidor quanto essas 40 versões existentes distribuídas entre esses maintainers também são versões confiáveis. Então, se der um problema na nuvem do GitHub, e o código não existir mais, tem que acontecer algo também com essas 40 pessoas, pois as versões disponíveis deles também são confiáveis devido a estrutura que o git mantém e foi projetado. Por esse motivo ele é um sistema distribuído e seguro.

**Chave SSH E Token**

Chave SSH é uma forma de estabelecer uma conexão segura e encriptada entre duas máquinas. No caso, vamos nos conectar com o servidor do GitHub e configurar a nossa máquina local como confiável para o GitHub, estabelecendo essa conexão com duas chaves, uma chave pública e uma chave privada. Então pegamos a chave pública e colocamos ela lá. A partir deste momento, o GitHub vai conhecer nossa máquina, então todos os repositórios que tivermos em nossa máquina por esse processo SSH e formos enviar para lá, o GitHub já vai conhecer a nossa máquina, tendo uma conexão prévia estabelecida, permitindo que enviemos outros códigos sem precisar utilizar senha.

A sequência para criar essa chave é:

$ ssh-keygen -t ed25519 -C "email do usuário"

Generating public/private ed25519 key pair.

Enter file in which to save the key (/c/Users/debor/.ssh/id\_ed25519):

.ssh : dentro da pasta c: users, as pastas que iniciam com ponto são pastas ocultas;

id\_ed25519 : é o nome da chave gerada;

**Etapas no CLI:**

* Gerar chave;
* Navegar até a pasta e pegar o nome da chave, o conteúdo dela;
* Colocar a chave pública no GitHub e passar a chave privada para o agente;

**Token de Acesso Pessoal**

O que é esse Token de Acesso? Um token de acesso, de forma simplificada, seria uma chave de 40 caracteres que te permite ter acesso a algo. No caso do Github você usaria uma chave assim no lugar da senha da sua conta ao executar operações pela linha de comando ou na API.

**Primeiros comandos com Git**

**Iniciando o Git e criando um commit**

* Iniciar o GIT: (git init);
* Iniciar o versionamento: (git add);
* Criar um Commit: (git commit);

Quando estamos lidando com o terminal, sempre colocamos o nome do programa na frente do comando. Então, no caso, estamos chamando pelo terminal o GIT, e estamos chamando pelo terminal comandos específicos do GIT, por isso, todos os comandos têm o GIT na frente.

**Criando um repositório:** aula prática

**Adicionando um arquivo:** aula prática

Tipo de arquivo: markdown (.md);

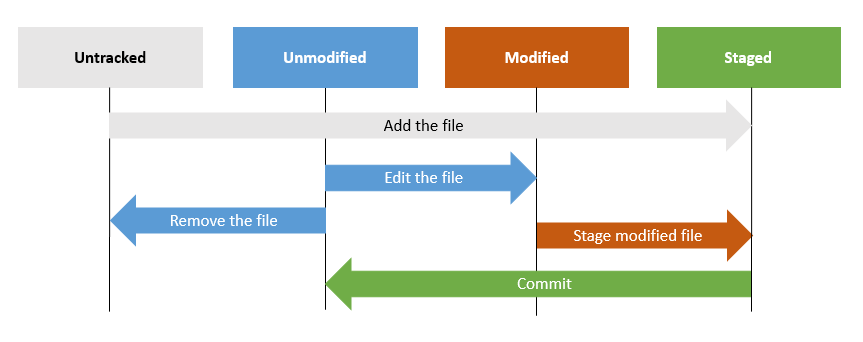
Ele é uma forma mais “humana” de escrever um arquivo HTML;

**Ciclo de Vida dos Arquivos no Git**

**Git Init:** cria uma pasta chamada .git e inicializa um conceito chamado de repositório. Usando o git init estamos de fato criando um repositório dentro do diretório (pasta)

**Tracked (rastreável)/Untracked (irrastreável)**

**—----UNTRACKED-------|—-------------------------------- TRACKED ------------------------------------**



**Tracked**: arquivos que o git tem ciência deles;

**Untracked**: arquivos que o git **não** tem ciência deles;

**Unmodified:** arquivos criados, sem nenhuma alteração;

**Modified:** arquivos unmodified com pequenas alterações;

**Staged:** arquivos prontos para fazer parte de um novo tipo de agrupamento;

Passo a Passo na criação de um arquivo:

**Comando Git Init:** inicia um novo repositório;

**Untracked:** entrada de um arquivo untracked que foi recém criado no repositório inicializado pelo git init (ex.: strogonoff.md ). O git ainda não sabe da existência dele;

**Comando Git Add:** move o arquivo untracked direto para o staged, na área em que ele está aguardando para entrar no palco;

**Unmodified → Modified:**  você tem um arquivo dentro do seu repositório que ainda não sofreu modificação, então você abre ele, faz uma alteração e automaticamente ele passa para modified. Como ele reconhece que esse arquivo foi alterado? o git compara o SHA1 dos arquivos e descobre que tem modificação.

**Modified → Staged:** Se rodarmos novamente o comando git add, agora no arquivo modified, esse arquivo vai para o staged, entrando também nessa área que está esperando a ação, aguardando para fazer parte de um outro grupo..

**Unmodified → Untracked:** se tivermos um arquivo unmodified, que não sofreu alteração nenhuma e removermos esse arquivo, ele retorna para untracked

**Staged → Commit:** quando um arquivo está no estado staged, ele está aguardando uma ação. Essa ação que ele espera é a de fazer parte de um commit.

Então quando colocamos esses arquivos no backstage, atrás do palco e esperando para entrar em ação, damos um commit neles, ou seja, envelopa todas essas modificações com significância, escreve uma mensagem para esse commit (contendo autor, data,etc), integra o commit de fato, que é o objeto commit e eles deixam de ser staged e vão para o estado commit. O commit então retorna esses arquivos para unmodified, recomeçando o ciclo.

Por que ele retorna? O git pega esses arquivos, finaliza as alterações e salva eles criando um snapshot (É como se ele pegasse uma câmera e tirasse uma foto do seu código nesse momento). Esse snapshot agora está salvo dentro do commit. Ele retorna então para unmodified pois está aguardando novas modificações. Você então roda o git add novamente e ele vai retornar para staged.

Portanto **unmodiified → modified → staged** é um ciclo que sempre ficará rodando entre esses 3 estágios, sempre voltando para *unmodified* quando você commita, se você abre ele e faz uma modificação ele passa para *modified* e se você adiciona ele a um commit ele vai primeiro para *staged*, quando você escreve o commit e está tudo certo ele volta para *unmodified*.

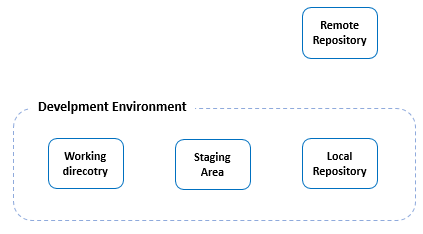
O que os repositórios significam?

Nesse momento nós temos 2 ambientes: o *ambiente de desenvolvimento* e o *servidor.*

**Ambiente de Desenvolvimento (working directory, staging area e local repository):** representa tudo que está na nossa máquina;

**Servidor (remote repository):** o git é um sistema distribuído então ele vai ter a versão dele no servidor, o *GitHub* e ele tem a versão que está na sua máquina. Portanto, a alteração que você fez no código na sua máquina, não repercute imediatamente na versão que está no seu repositório remoto, no remote repository.

A menos que você execute um conjunto de códigos específico, seu código não é empurrado ao seu repositório remoto.

****

**O git se organiza em dois ambientes:**

Funciona da seguinte forma: no seu ambiente de trabalho você tem seu repositório de trabalho (working directory), onde criamos o livro de receitas, e tem também o staging area (área de staging). Os arquivos vão sempre intercalar/transitar entre o working directory e o staging area a medida que você vai adicionando novos arquivos neste trabalho, modificando novos arquivos, etc.

Quando você faz um commit, ele passa a integrar o seu repositório local, e o seu repositório local, por sua vez, pode ser empurrado (push) para seu repositório remoto.

**Introdução ao GitHub**

Aula prática;

README: arquivo responsável por contar a história/finalidade do seu repositório;

**Resolvendo conflitos**

**Como os conflitos acontecem no GitHub e como resolvê-los**

Quando você tem seu código no GitHub e você baixa ele na sua máquina ou você acabou de enviar ele para lá, o que significa que o código é exatamente o mesmo que o da sua máquina, você tem seu arquivo lá, passando pelo versionamento distribuído.

Suponhamos então que outra pessoa vá lá e pegue seu código no GitHub e faça um clone dele. Nesse momento, ninguém mexeu nos códigos ainda, os dois estão sincronizados, em seguida, os dois abrem os arquivos para começar a editar de fato. É muito difícil duas fazerem exatamente a mesma alteração, sempre tem uma vírgula diferente, outro formato, um espaço a mais, etc. A partir de agora, os dois não estão mais em sincronia, serão 3 códigos diferentes, sendo que o seu código e o da pessoa que clonou tiveram edições na mesma linha de código. E é nesse momento que ocorrem os problemas.

Considere que a pessoa continuou e empurrou o código para o GitHub e você continuou trabalhando em outros arquivos e naquele mesmo arquivo. Agora o código que está no GitHub é o código dessa pessoa e seu código está desatualizado . Então o código que está no GitHub é diferente do que está na sua máquina e o GitHub está olhando e percebendo que o código que está lá é uma versão diferente, mais atual à sua. Quando você gerar um commit vai vir uma data, então a sua é uma versão mais antiga daquele código. Então você vai submeter o seu código para o GitHub e ele vai te falar exatamente isso.

O GitHub então vai dizer que você tem que puxar aquele arquivo mais atual e fazer as alterações a partir dele para poder empurrar de volta, para só então a sua versão ser a mais atualizada, a versão mais recente das mudanças que ocorreram. Nesse momento acontece o que é conhecido como **Conflito de Merge.**

**Conflito de Merge:** ocorre quando o github tenta juntar dois códigos, com alterações na mesma linha.

O github não vai fazer nenhuma alteração drástica nisso, não vai inferir nada. Ele vai deixar que você abra aquele arquivo no qual ocorreu a edição na mesma linha, de duas ou mais pessoas e você mesmo, de forma manual, deve resolver aquele conflito e só depois empurre o código para o github.