**Curso GIT Completo - do Básico ao Avançado**

**>> 26/01/2022 <<**

**Cap. 3 - Controle de Versão:**

Os softwares de Controle de Versão permitem o gerenciamento das versões, com documentação, podendo ser qualquer tipo de arquivo (.xml, .xls, .docx, .css, .txt, etc.).

* Muito difundido em empresas de software;
* Pode ser usado em pequenos ou grandes projetos;
* Usado para controlar as etapas de qualquer tipo de trabalho feito em um computador (trabalhos acadêmicos, textos diversos do Word, entre outros diversos tipos de arquivos).

**Funcionalidades:**

* Histórico de versões;
* Alternar entre versões;
* Backup – a versão anterior fica armazenada;
* Trabalho em Equipe;
* Desenvolvimento em paralelo em ambientes diferentes;
* Ramificação – podem-se criar diversas versões a partir de um ponto;
* Junção das funcionalidades após finalizadas;
* Rastreabilidade – identifica quais foram as mudanças efetuadas e quem as efetuou.

**Cap. 4 – Tipos de Controle de Versão:**

**Centralizado:**

* O histórico das alterações fica centralizadas em um único servidor;
* O usuário dever ter acesso a esse servidor para acessar o histórico;
* Cada versão deve ser baixada individualmente;
* Quando uma versão está sendo utilizada por um usuário, outros não poderão acessá-la.

**Distribuído:**

* Distribuído nas máquinas que estão operando no repositório;
* Também possui servidor;
* Todo histórico do projeto é trazido para a máquina local;
* Melhor performance de programação;
* Réplica de repositórios em cada máquina, assim, caso haja algum problema com o servidor, o projeto pode ser novamente obtido com os usuários envolvidos.

**Características do GIT:**

* Operações locais;
* Navegação pelo histórico;
* Criação de Branch (ramificação) - troca de versões sem que seja necessário comunicar o servidor para tal, o que dá maior eficiência ao programador;
* Somente adição de conteúdo ao histórico (não há remoção de itens do histórico);
* Integridade – um arquivo, uma vez rastreado, será rastreável permanentemente, até ser removido do projeto, e todas as alterações ficarão no histórico;
* Autonomia – várias pessoas realizam alterações sem dependência de arquivos (porém, quando os usuários forem integrar o projeto, devem verificar se há conflitos).

**Cap. 5 – Instalação do GIT:** <https://git-scm.com> e seguir com a instalação padrão.

**Cap. 8 – Configuração do Usuário e alguns Comandos:**

* git --version: Retorna a versão do GIT instalada;
* git config --global user.name “nome\_do\_usuario”: Configura o nome do usuário;
* git config --global user.email “e-mail\_do\_usuario”: Configura o e-mail do usuário;
* git --list: Lista as configurações do GIT.

**Cap. 9 – Criação de um Repositório Local:**

* dir: Lista o que tem na paste;
* git init: Inicializa um repositório vazio;
* “master”: É a Branch (ramificação) principal do projeto – a partir dessa branch, poderão ser feitas novas ramificações;
* dir -a: Lista os arquivos e pastas ocultos;
* A pasta “.git” armazena todas as informações do repositório (todo o histórico);
* Comando “cd ..”: volta à pasta acima da atual.

**Cap. 10 – Versionamento - Introdução:**

* git status: Retorna o estado do arquivo;
* git add nome\_do\_arquivo: Faz o rastreamento do arquivo;
* git add --all ou git add . ou git add -A: Adiciona todos os arquivos da pasta atual.

**Cap. 11 – Versionamento – Salvando Alterações:**

* git commit -m “Mensagem para o Commit”: Salva uma nova versão do banco de dados local do GIT.

**Cap. 12 – Funcionamento do GIT:**

**Working Directory (working tree):**

* Criado a partir do “git init” – é a pasta onde estamos trabalhando com o projeto;
* Arquivos Modificados;
* Arquivos Excluídos;
* Arquivos Adicionados.

**Staging Area:**

* Área de preparação;
* Staged – adicionado com “git add .”;
* Contém os arquivos preparados para serem versionados.

**Committed:**

* Arquivos salvos (versionados) e prontos para fazer upload.

**Cap. 13 – Visualizar Alterações sem abrir o projeto:**

* git diff: Retorna as informações sobre as alterações no diretório;
* git diff –cached ou git diff --staged: Retorna as diferenças na área de preparação (cached ou staged).

**Cap. 14 – Histórico de Alterações:**

* git log: Lista os commits feitos no projeto;
* git log --oneline: Lista os commits de forma resumida;
* CHA: Algoritmo que gera um número de associação com o commit, como, por exemplo: 23d199865904a9123043123f0de753754dea1acc;
* HEAD: Aponta a última modificação em uma Branch;
* Para mais variações de log, verificar o arquivo “Git+log+-+Variações.pdf”.

**Cap. 15 – Usando Commits Anteriores:**

* git checkout numero\_do\_CHA: Aponta o HEAD para o commit referente ao CHA e volta ao estado do projeto do commit informado;
* Não é necessário informar o número completo do CHA;
* Importante para verificação de alterações e cada commit do projeto;
* git checkout master: Aponta o HEAD para a última versão da Master.

**Cap. 16 – Desfazendo Alterações:**

* git checkout nome\_do\_arquivo: Desfaz as alterações feitas no arquivo informado. Também recupera o arquivo informado, caso tenha sido excluído e se queira recuperá-lo (antes de um novo commit);
* git reset --hard: Desfaz todas as alterações de todos os arquivos do projeto, desde que não tenha sido feito um novo commit.

**Obs.:** O “git checkout” não funciona em arquivos novos.

**Cap. 17 – Desfazendo Alterações Não Rastreadas:**

* git clean -f: Força a remoção de arquivos não rastreados com “git add”, caso haja, pois o “git reset” não funcionar é nesse caso.

**Cap. 18 – Ignorando Arquivos (arquivo “.gitignore”):**

Há momentos em que não queremos versionar arquivos;

* Por exemplo, arquivos de configuração da máquina local; ou
* Arquivos de senha; ou
* Arquivos .NET (.dll), etc..

**Criando um Arquivo .gitignore:**

* Na pasta do projeto, crie um arquivo .txt vazio e com qualquer nome;
* Entre no CMD e vá até a pasta do projeto;
* Renomeio o arquivo com: ren nome\_do\_arquivo.txt .gitignore (para Mac e Linux, utilizamos o comando “mv” ao invés do “ren”;
* Dentro desse arquivo, inserir informações sobre o que o Git deve ignorar quando for fazer os versionamentos. Por exemplo: “teste.bmp” (ignora esse arquivo), ou “NomeDaPasta/\*.bmp” (ignora todos os arquivos .bmp da pasta “NomeDaPasta”.

**Obs.:** O arquivo “.gitignore” poderá ser criado quando for criado o repositório no GitHub.

**Quais Arquivos Ignorar?**

* No site do <https://github.com/github/gitignore>, podemos verificar os arquivos .gitignore para todos os tipos de projetos, para todos os tipos de plataformas;
* Copiar e colar para o arquivo local .gitignore, conforme necessário.

**Cap. 19 – Clonando Repositórios:**

* git clone nome\_da\_pasta/ novo\_nome\_de\_pasta: Clona a pasta de um projeto do computador local, cria uma nova pasta com o nome desejado e insere essa clonagem nessa nova pasta;
* git clone <https://github.com/DevMasterTeam/GitCompleto.git>: Clona um repositório do GitHub (a partir da URL informada);

**Obs.:** O repositório <https://github.com/DevMasterTeam/GitCompleto.git> contém Templates Bootstrap para baixar gratuitamente.

**Cap. 20 – Introdução ao GitHub:**

* Repositório remoto de armazenamento com controle de versão;
* Amplamente utilizado ao redor do mundo – mais de 40mi de usuários e mais de 2,1mi de empresas;
* Comprado pela Microsoft em 2018.

**Cap. 21 – Criando Repositório:**

* Criar conta no GitHub;
* Em Repositories, acessar New;
* Repository Name: Nome único para o seu repositório;
* Description (opcional): Descrição mais detalhada do projeto;
* Public: Permite o acesso a outros usuários ao seu repositório;
* Private: Não permite o acesso a outros usuários;
* Initialize this repository with a README: Possibilita a criação de um arquivo README utilizando uma linguagem de marcação semelhante ao HTML;
* Add .gitignore: Cria um arquivo “.gitignore”, configurado automaticamente, dependendo da aplicação utilizada no projeto, uma vez que podemos indicar com qual linguagem estamos trabalhando (podendo, ainda, ser vazio);
* Choose a license: Indica o tipo de licença do seu software (caso seja Public);
* Creating a new file: Possibilita a criação de um arquivo diretamente no repositório do GitHub, sem a necessidade de se programar localmente e, depois, fazer o push;

**Cap. 22 – Versionar Alterações no Servidor:**

* git remote add origin url\_do\_repositorio: Associa o repositório Git local com o repositório GitHub remoto;
* git remote set-url url\_do\_repositório: Altera a associação do repositório local;
* git push -u origin master: Faz o primeiro push para o branch “master” do repositório do GitHub remoto;
* git push: depois do primeiro push, podemos utilizar apenas esse comando para com
* tinuar fazendo o versionamento;

**Cap. 23 – Baixar a Última Versão do Repositório:**

* Devemos sempre verificar como a equipe se houve alterações no projeto, antes de iniciarmos novas alterações;
* É importante sempre trabalharmos sobre a última versão do projeto;
* git pull ou git pull origin master: Baixa o repositório Git mais atualizado.

**Cap. 24 – Star, Fork e Pull Request:**

**Alterar projeto de público para privado:**

* Entrar na sua conta GitHub;
* Entrar no repositório que deseja fazer a alteração;
* Entrar em Settings;
* Na área “Danger Zone”, selecionar “Change Repository Visibility”;
* Selecionar “Make public”;
* Confirmar e digitar o nome do repositório a ser alterado;
* Selecionar “I understand, change repository visibility”.

**Star e Watch:** Ao visualizar um projeto de terceiros, podemos querer acompanhar suas mudanças, sem ter que entrar nele de tempos em tempos para ver se houve alterações. Para isso, usamos o Star:

* Acesse o repositório de terceiros desejado;
* Clique em “Star”: Possibilita o fácil acesso ao projeto “Starred”;
* Watch: Podemos receber notificações sobre o projeto favoritado, sendo que essas notificações são customizáveis;
* Podemos, ainda, escolher não sermos notificados sobre nenhuma mudança.

**Fork:** Traz o projeto completo de terceiros e replica ele em seu perfil, como se tivesse sido feito um clone do repositório, porém, sem ser necessário fazer o download para a máquina do usuário:

* Podemos fazer alterações e realizar push;
* No caso de clonarmos o repositório para a máquina local, não teríamos a possibilidade de fazer o push, pois a URL do repositório não está no nosso perfil do GitHub, mas em perfil de terceiros.

**Pull Request:** Solicita ao desenvolvedor original uma autorização para efetuarmos mudanças no código.

**Cap. 25 – Issues, Milestones, Labels:**

* Issues: Permite-nos informar aos desenvolvedores se há qualquer tipo de problema no projeto a ser tratado;
* Labels: Indica Marcações referentes às Issues abertas;
* Milestones: Indica um marco às Issues, informando quando esses problemas deverão ser tratados, permitindo um melhor cronograma de trabalho no projeto.

**Cap. 26 – Arquivo README:**

* Carrega as informações sobre o projeto;
* Pode conter informações Legais;
* Pode conter a Documentação do projeto;
* Selecionar “Add a README”;

**Ferramente Dillinger -** <https://dillinger.io>:

* Online;
* Permite a criação de arquivos README facilmente;
* Disponibiliza um exemplo para nos basearmos;
* Podemos alterar esse exemplo e copiar e colar para dentro do nosso arquivo README.

**Cap. 27 – Bitbucket:**

* Acessar <https://bitbucket.com>;
* Criar Chave SSH: seguir os passos que o site informa;
* Criar autenticação de dois fatores (opcional);
* O Bitbucket possui algumas diferenças de pull e push para repositórios.

**Cap. 30 – Conceitos de Branch:**

**Branch:** Ramificação que permite que o projeto siga por “n” caminhos a partir de um mesmo ponto, onde as funcionalidades podem ser desenvolvidas separadamente, sem impactar funcionalidades estáveis no projeto.

**Cap. 32 – Criando uma Branch:**

* git branch: Lista as branches do repositório local (o asterisco indica em qual branch estamos no momento);
* git branch nome\_da\_nova\_branch: Cria uma branch;
* git checkout nome\_da\_branch: Altera a branch de trabalho;
* git checkout -b nome\_da\_branch: Cria uma branch e altera a branch de trabalho para a recém criada.

**Cap. 33 – Enviando Branch para o Repositório:**

* git push –-set-upstream origin nome\_da\_branch ou git push -u origin nome\_da\_branch (dentro da branch desejada): faz o mapeamento da branch para o servidor.

**Cap. 34 – Atualizando Branch:**

* git pull: Atualiza as branches;

**Cap. 35 – Remover Branches Locais:**

* git branch -d nome\_da\_branch: Remove a branch (deve-se estar em outra branch para fazer isso);
* git branch -D nome\_da\_branch: Força a remoção da branch, caso necessário.

**Cap. 36 – Remover Branches Remotas (servidor):**

* git push --delete origin nome\_da\_branch: Remove do servidor;
* IMPORTANTE alinhar com a equipe, pois pode haver outras pessoas utilizando a branch que estamos apagando.

**Cap. 37 – Renomear Branch:**

* git branch -m novo\_nome\_da\_branch: Renomeia a branch atual;
* git branch -m nome\_da\_branch\_para\_alterar novo\_nome\_da\_branch: Renomeia uma branch na qual não estamos;
* IMPORTANTE alinhar com a equipe antes de alterar nomes de branches.

**Cap. 38 – Mesclando Alterações (Merge):**

* Criar funcionalidades separadas da branch Master;
* Comando Merge – integra a branch;
* git merge nome\_da\_branch\_para\_integrar: Faz o merge, caso não haja conflitos, devendo-se, sempre, estar na branch principal (master).

**Cap. 39 – Resolvendo conflitos:**

* Quando há conflitos, o Git indica o arquivo que está ocorrendo esse conflito no GitBash, depois de um pull;
* Abrir o arquivo no editor;
* Os conflitos aparecem entre <<<<<<< >>>>>>>;
* No Visual Studio Code, os conflitos ficam melhor visualizáveis, pois são destacados com cores;

**Cap. 40 – Resolução de Conflitos Usando kdiff3:**

* Ferramenta que auxilia a identificação e resolução de conflitos;
* Baixar e instalar a ferramenta kdiff3;
* git config –-global –-list: Lista as configurações globais do Git;
* git config –-global –-add merge.tool kdiff3: Adiciona o kdiff3 às configurações do Git;
* git config –-global –-add mergetool.kdiff3.path “C:\Program Files\KDiff3\kdiff3.exe”: Executa o kdiff3 no Git a partir do caminho do executável kdiff3.exe;
* git config –-global –-add mergetool.kdiff3.trustExitCode false: outra configuração que o professor não explicou exatamente para que serve;
* git mergetool: Verifica os arquivos que necessitam de merge – abre uma interface para verificar e selecionar o que eu quero fazer – o professor não explicou direito, também;
* git diff --staged: Retorna as diferenças da área staged.

**Cap. 41 – Exemplo de SCM (Source Code Management):**

* SCM – Source Code Management – Administração do Código-fonte;
* Faz-se as modificações na branch do desenvolvedor (que não é a branch Master);
* Testa as modificações na branch do desenvolvedor;
* Ir para a branch “master” e fazer o merge da branch do desenvolvedor com o comando git merge nome\_da\_branch\_do\_desenvolveror.

**Cap. 42 – Pull Request:**

* Após alterações e testes no projeto, sempre na branch do desenvolvedor;
* git commit -a -m “Informacoes\_da\_Commit”: Adiciona o que foi alterado para a área “staged” e efetua o commit. Serve como um atalho pois, ao invés de colocarmos “git add .” e, depois, “git commit -m “Informacoes\_da\_commit””, podemos fazer os dois comandos em uma única linha;
* git push -u origin nome\_da\_branch: Dentro da branch desejada, faz o mapeamento da branch para o servidor;
* Acessar o GitHub;
* Entrar no repositório e selecionar “Pull requests”;
* Selecionar “New pull request”;
* É uma requisição de mudança que é criada e alguém tem que revisar.

**Cap. 43 – Criação e Listagem de Tags:**

* Tag é um ponteiro para commits específicos;
* A Tag pode fazer a marcação de qualquer commit;
* “master” é uma Tag padrão que marca, somente, o último commit;
* Marca commits “chaves” do projeto, por exemplo, alguma versão que tenha sido desenvolvida e esteja Estável;
* Facilita a localização de commits-chaves;
* git tag -a nome\_da\_tag -m “Mensagem\_sobre\_a\_Tag” (as tags devem ter um nome curto, de preferência – exemplo: “v1.0”): cria a Tag;
* git tag: faz a listagem das Tags.

**Cap. 44 – Enviando Tag para o Repositório:**

* git push origin nome\_da\_tag: Envia a tag para o repositório;
* Após serem enviadas ao servidor, as tags aparecerão no GitHub na aba Releases;
* **PARA TRANSFORMAR A TAG EM RELEASE:**

**Cap. 45 – Utilizando Tags:**

As Tags são ramificações do projeto, onde podemos fazer testes e commits sem que isso impacte no projeto final. Caso seja decidido que as alterações são relevantes para o projeto final, devemos replicar essas mudanças direto na branch.

* git chechout nome\_da\_tag: Vai para a Tag informada;
* Quando fazemos checkout para uma Tag, entramos no modo Detached, que significa que não há atrelamento ao HEAD (normalmente, a branch “master”) ou a nenhuma outran branch;
* Assim, podemos efetuar mudanças no projeto e, inclusive, fazer commits, porém, eles não impactarão nos outros commits preexistentes;
* Quando saímos da Tag, com checkout, para outra branch, há uma mensagem informando que o há commits deixados para trás, com número de identificação e comentário.

**Puxando o commit da Tag (que foi deixado para trás):**

* git branch nova\_branch número\_de\_identificação ou git checkout -b nova\_branch número\_de\_identificação: Cria uma branch a partir do commit, identificado pelo número;
* Assim, o commit é atrelado, e podemos fazer o push completo, sem perda de commits;
* As alterações não influenciam em nenhuma outra versão do projeto.

**Cap. 46 – Removendo Tags:**

* git tag -d nome\_da\_tag: Deleta a tag local;
* git push -–delete origin nome\_da\_tag: Deleta a tag no repositório.

**Cap. 47 – Tags em Commits Antigos (commits específicos):**

* Entrar no commit desejado com “git checkout número\_do\_commit”;
* git tag -a nome\_da\_tag -m “Mensagem\_sobre\_a\_Tag”: Cria a Tag;
* git push origin nome\_da\_tag: Envia a tag para o servidor;
* git checkout nome\_da\_tag: Entrar na tag para utilizá-la.

**Sintaxe alternativa para Criação de tags em commits específicos:**

* git tag -a nome\_da\_tag número\_do\_commit -m “Mensagem\_sobre\_a\_Tag”: Cria a tag direto no commit informado, sem precisar, necessariamente, entrar nele com “git checkout”.

**Cap. 48 – Stash: Uso e Criação:**

Quando fazemos alterações ao projeto a partir de uma branch, essas alterações devem ser comitadas ou descartadas para que possamos alterar de branch. Caso não queiramos seguir nenhuma dessas duas opções, podemos colocar as alterações em Stash.

* git stash save “Mensagem” ou git stash: Guarda as alterações na memória do Git, sem a necessidade de se fazer commit. Assim, o commit pode ser feito posteriormente, e o programador pode migrar entre branches;
* Podemos fazer diversas alterações e diversos “git stash” diferentes, que serão armazenados na memória como uma lista;
* git stash list: Lista todas as alterações em Stash.

**Cap. 49 – Stash: Listando e Removendo:**

* git stash apply: Aplica o Stash mais recente;
* git stash pop: Aplica o Stash mais recente e remove os demais;
* git stash drop stash@{1}: Exclui um stash, onde “1” é o número do stash a ser excluído;
* git stash apply stash@{1}: Aplica o stash entre chaves;
* git stash pop stash@{1}: Aplica o stash entre chaves e dropa os demais.

**Cap. 50 – Desfazendo Commits:**

* git reset -–hard: Volta ao commit atual, excluindo as alterações não comitadas;
* git reset –-hard HEAD~2: Volta quantos commits necessários, sendo que o “2” representa quantos devem-se voltar;
* **ATENÇÃO: as alterações serão perdidas;**
* git commit –-amend: Adiciona alterações ao commit. Esse comando irá abrir um editor no GitBash. Devemos:
  + Alterar o nome do commit, que está na primeira linha do editor;
  + Apertar ESC;
  + Digitar “:wq” – “w” vem de “write” e “q” vem de “quit”.

**Cap. 51 - Fetch:**

Fetch é um comando que já foi utilizado através do “git pull”. Quando executamos o “git pull”, o Git executa “git fetch” e “git merge”.

* git fecth: Traz as atualizações do servidor remoto para o repositório local;
* git merge: Aplica o merge, caso não haja conflitos.

A utilização desses dois comandos separadamente possibilita que tenhamos a visualização das diferenças entre seu repositório local e o repositório online.

**Cap. 52 - Rebase:**

Rebase traz commits de outras branches para a branch atual. Isso possibilita uma estrutura mais linear de commits, pois evita criar commits paralelos à branch “master”, como acontece com o comando “git merge”.

Normalmente, utiliza-se como abaixo:

* Entrar na branch desejada;
* git rebase master: Traz os commits da branch “master”;
* Voltar para a branch “master”;
* git rebase branch\_desejada: Traz os commits.

**Cap. 53 - Alias:**

Alias é um comando de abreviação para a utilização dos comandos do Git.

* git config –-global alias.s status: Faz com que a letra “s” possa ser utilizado em substituição da palavra “status”. Podemos aplicar esse princípio com outros comandos;
* git config –-global –-unset alias.s: Desfaz o alias da letra “s”.

**Cap. 54 - Remote: - git remote -v.**

**Cap. 55 – Grep:**

* Filtra as listagens;
* git branch | grep dev: Retorna as branches que contenham o valor “dev”;
* git tag | v0: Retorna as tags que contenham o valor “v0”;

**Cap. 56 - Sourcetree:**

Ferramenta gráfica utilizada junto ao Git.

* Site: <https://www.sourcetreeapp.com>;
* Permite a administração de branches e commits de um projeto de forma gráfica.

**Cap. 57 - GitKraken:**

Ferramenta visual para utilizar o Git;

* Site: <https://www.gitkraken.com>;
* Permite a administração de branches e commits de um projeto de forma gráfica.

**Cap. 58 – Visual Studio:**

O Git tem integração com o Visual Studio.

**Cap. 59 – Android Studio:**

O Git tem integração com o Android Studio.