Curso de git

do Básico ao Avançado

Instalação git Windows

- Instalar git no Windows é muito **fácil!**
- Acessar o site: https://git-scm.com/downloads
- Fazer o download do executável;
- E seguir as instruções;

Instalação git Linux

- Instalar git no Linux também é muito **fácil!**
- Acessar o site: https://git-scm.com/download/linux
- Teremos que seguir as instruções dependendo da nossa distro;
- E iniciar a instalação com nosso gerenciador de pacotes;

Instalação do VS Code

- O VS Code é o editor que vamos utilizar no curso;
- Porém **não é uma obrigatoriedade**, use o de sua preferência;
- A grande jogada é que ele possui um terminal integrado, facilitando as nossas ações com o **git**;
- Além de ser um editor super atualizado e que aceita diversas linguagens e ferramentas de programação;

O que é controle de versão?

- Uma técnica que ajuda a **gerenciar o código-fonte** de uma aplicação;
- Registrando **todas as modificações** de código, podendo também reverter as mesmas;
- Criar versões de um software em diferentes estágios, podendo **alterar facilmente entre elas**;
- Cada membro da equipe pode trabalhar em uma versão diferente;
- Há ferramentas para trabalhar o controle de versão como: **git** e SVN

O que é git?

- O sistema de controle de versão **mais utilizado do mundo** atualmente;
- O git é baseado em **repositórios**, que contêm todas as versões do código e também as cópias de cada desenvolvedor;
- Todas as operações do git **são otimizadas para ter alto desempenho**;
- Todos os objetos do git são **protegidos como criptografia** para evitar alterações indevidas e maliciosas;
- O git é um **projeto de código aberto**;

Como tirar o máximo proveito

- Codifique **junto comigo**;
- Crie seus **próprios exemplos**;
- Crie **outros casos** utilizando os recursos aprendidos no curso e também mesclando eles;
- **Dica bônus:** ouça e depois pratique!

Introdução

Conclusão da seção

Comandos fundamentais

Introdução da seção

O que é um repositório?

- É onde o código será **armazenado**;
- Na maioria das vezes cada projeto tem **um repositório**;
- Quando criamos um repositório estamos iniciando um projeto;
- O repositório pode ir para servidores que são especializados em gerenciar repos, como: **GitHub** e **Bitbucket**;
- Cada um dos desenvolvedores do time pode baixar o repositório e **criar versões diferentes** em sua máquina;

Criando repositórios

- Para criar um repositório utilizamos o comando: **git init**
- Desta maneira o git vai criar os arquivos necessários para inicializá-lo;
- Que estão na pasta oculta **.git**;
- Após este comando o diretório atual **será reconhecido pelo git como um projeto** e responderá aos seus demais comandos;

O que é o GitHub?

- É um **serviço para gerenciar repositórios**, gratuito e amplamente utilizado;
- Podemos **enviar nossos projetos** para o GitHub e disponibilizá-lo para outros devs;
- O GitHub é gratuito tanto para projetos públicos como **privados**;
- Vamos criar uma conta em: https://github.com

Enviando repositórios para o GH

- Podemos facilmente **enviar nossos repos** para o GitHub;
- Precisamos criar o projeto no GitHub, inicializar o mesmo no git em nossa máquina, sincronizar com o GH e enviar;
- E esta sequência que parece ser complexa é facilmente executada **por poucos comandos**;
- Vale lembrar que só fazemos **uma vez por projeto** este fluxo;
- Porém alguns dos comandos utilizados vão ser úteis ao longo do curso;

Verificando mudanças do projeto

- As mudanças do projeto podem ser verificadas por: **git status**
- Este comando é utilizado **muito frequentemente**;
- Aqui serão mapeadas todas as alterações do projeto;
- Como: **arquivos não monitorados** e **arquivos modificados**;
- Podemos também dizer que é a **diferença** do que já está enviado ao servidor ou salvo no projeto;

Adicionando arquivos ao projeto

- Para adicionar arquivos novos a um projeto utilizamos: **git add**
- Podemos adicionar **um arquivo** específico como também **diversos de uma vez só**;
- Somente adicionando arquivos eles serão monitorados pelo git;
- Ou seja, **se não adicionar ele não estará** no controle de versão;
- É interessante utilizar este comando de tempos em tempos para não perder algo por descuido;

Salvando alterações do projeto

- As alterações salvas do projeto são realizadas por: **git commit**
- Podemos commitar **arquivos específicos** ou vários de uma vez com a flag **-a**
- É uma boa prática enviar **uma mensagem a cada commit**, com as alterações que foram feitas;
- A mensagem pode ser adicionada com a flag **-m**

Enviando código ao repo remoto

- Quando finalizamos uma funcionalidade nova, **enviamos o código ao repositório remoto**, que é código-fonte;
- Esta ação é feita pelo **git push**
- Após esta ação **o código do servidor será atualizado baseando-se no código local** enviado;

Recebendo as mudanças

- É comum também ter que **sincronizar o local** com as mudanças do remoto;
- Esta ação é feita pelo **git pull**
- Após o comando serão **buscadas atualizações**, se encontradas elas **serão unidas ao código atual** existente na nossa máquina;

Clonando repositórios

- O ato de baixar um repositório de um servidor remoto é chamado de **clonar repositório**;
- Para esta ação utilizamos **git clone**
- Passando a **referência** do repositório remoto;
- Este comando é utilizado quando **entramos em um novo projeto**, por exemplo;

Removendo arquivos do repo

- Os arquivos **podem ser deletados da monitoração** do git
- O comando para deletar é **git rm**
- Após deletar um arquivo do git ele não terá mais suas atualizações consideradas pelo git;
- Apenas quando for adicionando novamente pelo **git add**

Histórico de alterações

- Podemos **acessar um log** de modificações feitas no projeto;
- O comando para este recurso é **git log**
- Você receberá uma informação dos **commits realizados** no projeto até então;

Renomeando arquivos

- Com o comando **git mv** podemos renomear um arquivo;
- O mesmo também pode ser **movido para outra pasta**;
- E isso fará com que este novo arquivo **seja monitorado pelo git**;
- O arquivo anterior é **excluído**;

Desfazendo alterações

- O arquivo modificado pode ser **retornado ao estado original**;
- O comando utilizado é o **git checkout**
- Após a utilização do mesmo o arquivo sai do staging;
- Caso seja feita uma próxima alteração, ele entra em staging novamente;

Ignorando arquivos no projeto

- Uma técnica muito utilizada é **ignorar arquivos do projeto**;
- Devemos inserir um arquivo chamado **.gitignore** na raiz do projeto;
- Nele podemos inserir todos os arquivos que não devem entrar no versionamento;
- Isso é útil para **arquivos gerados automaticamente** ou arquivos que contêm **informações sensíveis**;

Desfazendo todas as alterações

- Com o comando **git reset** podemos resetar as mudanças feitas
- Geralmente é utilizado com a flag **--hard**
- Todas as alterações **commitadas** e **também as pendentes** serão excluídas;

Comandos fundamentais

Conclusão da seção

Branches

Introdução da seção

O que é um branch?

- Branch é a forma que o git **separa as versões dos projetos**;
- Quando um projeto é criado ele inicia na branch **master**, estamos trabalhando nela até este ponto do curso;
- Geralmente cada nova feature de um projeto **fica em um branch separado**;
- Após a finalização das alterações os **branchs são unidos** para ter o código-fonte final;

Criando e visualizando os branches

- Para visualizar os branchs disponíveis basta digitar **git branch**
- Para criar um branch você precisa utilizar o comando **git branch <nome>**
- Estas duas operações são muito utilizadas no dia a dia de um dev;

Deletando branches

- Podemos deletar um branch com a flag **-d** ou **--delete**
- **Não é comum deletar um branch**, normalmente guardamos o histórico do trabalho;
- Geralmente se usa o delete quando o branch foi criado errado;

Mudando de branch

- Podemos mudar para outro branch utilizando o comando **git checkout -b <nome>**
- Este comando também é utilizado para dispensar mudanças de um arquivo;
- Alterando o branch podemos levar alterações que não foram commitadas junto, **tome cuidado!**

Unindo branches

- O código de dois branches distintos pode ser unido pelo comando **git merge <nome>**
- Outro comando para a lista dos **mais utilizados**;
- Normalmente é por meio dele que recebemos as atualizações de outros devs;

Unindo branches

- O código de dois branches distintos pode ser unido pelo comando **git merge <nome>**
- Outro comando para a lista dos **mais utilizados**;
- Normalmente é por meio dele que recebemos as atualizações de outros devs;

Stash

- Podemos salvar as modificações atuais **para prosseguir com uma outra abordagem de solução** e não perder o código
- O comando para esta ação é o **git stash**
- Após o comando o branch será resetado para a sua versão de acordo com o repo;

Recuperando stash

- Podemos verificar as stashs criadas pelo comando **git stash list**
- E também podemos recuperar a stash com o comando **git stash <nome>**
- Desta maneira podemos continuar de onde paramos com os arquivos adicionados a stash

Removendo a stash

- Para limpar totalmente as stash de um branch podemos utilizar o comando **git stash clear**
- Caso seja necessário deletar uma stash específica podemos utilizar **git stash drop <nome>**

Utilizando tags

- Podemos criar tags nos branches por meio do comando **git tag -a <nome> -m “<msg>”**
- A tag é diferente do stash, serve como um **checkpoint de um branch**;
- É utilizada para demarcar estágios do desenvolvimento de algum recurso;

Verificando e alterando tags

- Podemos verificar uma tag com o comando **git show <nome>**
- Podemos trocar de tags com o comando **git checkout <nome>**
- Desta maneira podemos retroceder ou avançar em checkpoints de um branch;

Enviando e compartilhando tags

- As tags podem ser **enviadas para o repositório de código**, sendo compartilhada entre os devs;
- O comando é **git push origin <nome>**
- Ou se você quiser enviar mais tags **git push origin --tags**

Branches

Conclusão da seção

Compartilhamento e atualização

Introdução da seção

Encontrando branches

- Branches novos são criados a todo tempo e o **seu git pode não estar mapeando eles**;
- Com o comando **git fetch** você é atualizado de todos os branchs e tags que ainda não estão reconhecidos por você;
- Este comando é útil para utilizar o branch de algum outro dev do time, por exemplo;

Recebendo alterações

- O comando **git pull** serve para recebermos atualizações do repositório remoto;
- Cada branch pode ser atualizado com o git pull;
- Utilizamos para atualizar a master do repo como também quando trabalhamos em conjunto e queremos receber as atualizações de um dev;

Enviando alterações

- O comando **git push** faz o inverso do pull, ele envia as alterações para o repo remoto;
- Serve também para **enviar as atualizações de um branch específico** para um outro dev;
- Ou quando terminamos uma tarefa e precisamos enviar ao repo;

Utilizando o remote

- Com o **git remote** podemos fazer algumas ações como: adicionar um repo para trackear ou remover;
- Quando criamos um repo remoto, adicionamos ele ao git com **git remote add origin <link>**

Trabalhando com submódulos

- Submódulo é a maneira que temos de possuir **dois ou mais projetos em um só repositório**;
- Podemos adicionar uma dependência ao nosso projeto atual, porém mantendo suas estruturas separadas;
- Para adicionar o submódulo utilizamos o comando **git submodule add <repo>**
- Para verificar os submódulos o comando é **git submodule**

Atualizando submódulo

- Para atualizar um submódulo primeiro devemos **commitar as mudanças**;
- E para enviar para o repo do submódulo utilizamos **git push --recurse-submodules=on-demand**
- Este fluxo fará a atualização apenas do submódulo;

Compartilhamento e atualização

Conclusão da seção

Análises e inspeção

Introdução da seção

Exibindo informações

- O comando **git show** nos dá diversas informações úteis;
- Ele nos dá as informações do branch atual e também **seus commits**;
- As **modificações de arquivos** entre cada commit também são exibidas;
- Podemos exibir as informações de tags também com: **git show <tag>**

Exibindo diferenças

- O comando **git diff** serve para exibir as diferenças de um branch;
- Quando utilizado as diferenças do branch atual com o remoto serão exibidas no terminal;
- Podemos também verificar a diferença entre arquivos: **git diff <arquivo> <arquivo_b>**

Log resumido

- O comando **git shortlog** nos dá um log resumido do projeto;
- Cada commit será unido por **nome do autor**;
- Podemos então saber quais commits foram enviados ao projeto e por quem;

Utilizando o describe

- Com o comando **git describe --tags** podemos verificar todas as tags do nosso projeto;
- Com a opção --all recebemos também a referência das tags;

Análises e inspeção

Conclusão da seção

Administração do repositório

Introdução da seção

Limpando arquivos untracked

- O comando **git clean** vai verificar e limpar arquivos não estão sendo trackeados;
- Ou seja, todos que você **não utilizou git add;**
- Utilizado para arquivos que são **gerados automaticamente**, por exemplo, e atrapalham a visualização do que é realmente importante;

Otimizando o repositório

- O comando **git gc** é uma abreviação para **garbage collector**;
- Ele identifica arquivos que **não são mais necessários** e os exclui;
- Isso fará com que o repositório seja otimizado em questões de **performance**;

Chegando integridade de arquivos

- O comando **git fsck** é uma abreviação de File System CheckK;
- Esta instrução verifica a integridade de arquivos e sua conectividade;
- Verificando assim possíveis **corrupções em arquivos**;
- **Comando de rotina**, utilizado para ver se está tudo certo com nossos arquivos;

Reflog

- O **git reflog** vai mapear todos os seus passos no repositório, até uma mudança de branch é inserida neste log;
- Já o **git log**, que vimos anteriormente, apenas armazena os commits de um branch;
- Os **reflogs ficam salvos até expirar**, o tempo de expiração padrão é de 30 dias;

Recuperando arquivos com reflog

- Podemos avançar e também retroceder nas **hashs** do reflog;
- Para isso utilizamos o comando **git reset --hard <hash>**
- Caso você tenha algo que queira salvar, pode utilizar o **git stash** antes;
- **Lembrando:** o reflog expira com o tempo!

Transformando o repo para arquivo

- Com o comando git archive podemos transformar o repo um arquivo compactado, por exemplo;
- O comando é **git archive --format zip --output master_files.zip master**
- E então a master vai estar zipada no arquivo master_files.zip

Administração do repositório

Conclusão da seção

GitHub

Introdução da seção

Criando repositório

- No **GitHub** inicializamos os repositórios, e temos algumas informações importantes para preencher, vamos vê-las em detalhes;
- Algumas delas são: Nome do repo, descrição, licença;
- **Tudo poderá ser alterado ao longo do seu projeto**, mas é interessante conhecer os detalhes das informações para configurar um projeto;

A aba Code

- Na aba **Code** teremos acesso a informações importantes, como o próprio código fonte;
- Podemos checar também uma documentação do projeto pelo **README.md**;
- E os detalhes da **licença do projeto**;
- Criar branches, adicionar arquivos e muito mais!

A aba Issue

- Na aba **Issue** podemos criar tarefas ou possíveis bugs do projeto;
- Interessante para a organização se manter ciente do que ainda **precisa fazer ou corrigir**;
- Normalmente há um padrão para criação de novos issues;
- Podemos utilizar o **Markdown** no texto também (igual o README.md);
- A issue deve ter uma label e também um responsável;

A aba Pull Request

- Na aba **Pull Request** é onde os colaboradores do projeto enviam código para resolver as **issues** ou **adicionar novas** funcionalidades ao projeto;
- A ideia é que o código não seja inserido direto na master e sim passe por um pull request, **para ser analisado**;
- O pull request vem de um **novo branch** criado no projeto e enviado para o repo, com o incremento de código;

A aba Actions

- Na aba **Actions** é onde se cria as automatizações de deploy com integração em outros serviços;
- Incluindo **CI/CD** (Continuous Integration / Continuous Development);
- Ou seja, podemos criar uma rotina de atualizar a master automaticamente e outros processos;

A aba Projects

- Na aba **Projects** podemos criar um projeto e utilizar um quadro de tarefas;
- Este processo é conhecido como **Kanban** e pode ajudar a organizar seu time, criando notas que podem virar **issues**;
- Estrutura interessante: Backlog, Retorno de qualidade, Desenvolvimento, Teste, Finalizadas;
- A tela lembra muito o software **Trello**;

A aba Wiki

- Na aba **Wiki** podemos criar uma documentação mais extensa para o projeto;
- Como descrever funcionalidades, bugs conhecidos e não solucionados, entre outras funções;
- A ideia é que seja um **repositório de conhecimento** sobre o projeto;

A aba Insights

- Na aba **Insights** temos informações detalhadas do projeto, como:
- Quem são os contribuidores, commits, forks e muito mais;
- Interessante para entender como o projeto está andando e a **sua evolução desde o início**;

A aba Settings

- Na aba **Settings** temos acesso a diversas configurações do projeto;
- É onde podemos alterar o nome do repo ou remover/adicionar features;
- E também é nela que **adicionamos colaboradores** ao projeto;
- O repositório poder **removido** nesta aba;

Criando um Gist

- **Gist** são pequenos blocos de código que podem ser hospedado no GitHub também;
- Você pode armazenar uma solução que achou interessante para algum problema e não quer perder, por exemplo;
- E o link do Gist **pode ser compartilhado**;
- No fim das contas o Gist **acaba sendo um repositório** também;

Encontrando repositórios

- O GitHub não serve só para salvar os nossos projetos, podemos encontrar muitos **repos interessantes**;
- Podemos até aprender com isso também, olhando o **código fonte** de desenvolvedores experientes;
- E não para por aí: você pode dar **star** nos projetos que gostou ou **fork** nos que deseja continuar em um repo próprio;

GitHub

Conclusão da seção

Markdown

Introdução da seção

O que é Markdown?

- O **Markdown** é uma forma de adicionar estilo a textos na web;
- O arquivo **README.md** aceita Markdown;
- Você vai conseguir exibir: trechos de código, links, imagens e muito mais;
- Dando uma **melhor experiência para o usuário** nas suas documentações;

O que é Markdown?

- O **Markdown** é uma forma de adicionar estilo a textos na web;
- O arquivo **README.md** aceita Markdown;
- Você vai conseguir exibir: trechos de código, links, imagens e muito mais;
- Dando uma **melhor experiência para o usuário** nas suas documentações;

Cabeçalhos

- Os **cabeçalhos em markdown** são determinados pelo símbolo #
- Cabeçalhos são os famosos títulos ou **headings** do HTML
- # => h1, ## => h2, ### => h3 e assim por diante

Ênfase

- Temos símbolos que podem dar **ênfase ao texto**;
- Para escrever em **negrito**: **texto** ou __texto__
- Para escrever em **itálico**: *texto* ou _texto_
- **Combinando** os dois: _um **texto** combinado_

Listas

- Temos as listas **ordenadas** em **não ordenadas** em markdown;
- As listas não ordenadas começam os itens com: *** Item**
- As listas ordenadas com: **1. Item**

Imagens

- É possível **inserir imagens** em markdown também;
- Veja a sintaxe: ![Texto Alt](link imagem);
- A imagem pode **estar no próprio repo** ou **ser externa**;

Links

- Com o markdown podemos **inserir links** de forma fácil;
- A sintaxe é a seguinte: [Texto do link](link)
- Se for um link do GitHub pode inserir **de forma direta**:

  https://www.github.com

Código - GitHub

- Podemos inserir **código** no Markdown também;
- A sintaxe é: ``` código ```
- Esta sintaxe é do markdown **especial do GitHub**;

Task list - GitHub

- Podemos inserir uma **lista de tarefas** pelo Markdown;
- A sintaxe para tarefas concluídas: - [x] CSS do rodapé
- Para não concluídas: - [ ] CSS da página de contatos
- Esta sintaxe é do markdown **especial do GitHub**;

Markdown

Conclusão da seção

GitHub Pages

Introdução da seção

O que é GitHub Pages

- Uma forma de criar uma página estática nos **servidores do GitHub**;
- Ou seja, uma alternativa **gratuita** para hospedar nosso portfólio;
- Muito simples de colocar no ar, **não precisa de domínio ou servidor**;
- Muitas empresas utilizam para **apresentar o seu projeto** ou **a própria documentação**;

Como criar a página

- Você deve seguir alguns passos simples, veja:

1. Criar um repositório com o nome **nomedousuario.github.io**
2. Clonar o repositório no nosso computador
3. Adicionar o código do projeto na **branch master**
4. Enviar o código por meio de **push**
5. E pronto, você tem um site em **https://nomedousuario.github.io**

GitHub Pages

Conclusão da seção

A importância do commit

- **O problema:** commits sem sentido atrapalham o projeto;
- Precisamos padronizar os commits, para que o projeto cresça de forma saudável também no versionamento, isso ajuda em:
- Review do **Pull Request**;
- Melhoria dos log em **git log**;
- Manutenção do projeto (voltar código, por exemplo);

Branches com commits ruins

- Há uma solução chamada **private branches**;
- Onde criamos branches que **não serão compartilhados no repositório**, então podemos colocar qualquer commit;
- Ao fim da solução do problema podemos fazer um **rebase**;
- O comando será: git rebase <atual> <funcionalidade> -i
- Escolhemos os branches para excluir (**squash**) e renomear com (**reword**);

Boas mensagens de commit

- Separar **assunto** do corpo da mensagem;
- Assunto com no **máximo 50 caracteres**;
- Assunto com **letra inicial maiúscula**;
- Corpo com no **máximo 72 caracteres**;
- Explicar o **por que e como** do commit, e não como o código foi escrito;