Curso de Git & Github

Cursos em vídeo – prof. Gustavo Guanabara

Mateus Machado

[Nome da empresa]  [Endereço da empresa]

“Esta página foi intencionalmente deixada em branco”

Primeiro explicando do porque é comum vermos textos semelhantes ao de cima e do porque temos página em branco no começo de Livros.

Razões

Processo de impressão: Os livros não são impressos página por página, mas sim em grandes folhas que são dobradas e cortadas, formando cadernos. O número de páginas em um caderno é sempre um múltiplo de 4 (4, 8, 12, 16, 20, 24, 28, 32, 36, 40 etc.) Se o conteúdo do livro não preencher todos os cadernos, páginas em branco são adicionadas para completar a formação.

**Facilidade de encadernação:** As páginas em branco ajudam a proteger o livro durante o processo de encadernação, evitando que a primeira página impressa seja danificada.

**Design e estética.** Páginas em branco podem ser usadas para criar um respiro visual no livro, separando seções ou capítulos.

**Intenção do autor ou editor:** Em alguns casos, as páginas em branco podem ser uma escolha deliberada ou autor ou editor, seja por motivos estéticos, para criar um momento de pausa para o leitor, ou até mesmo como parte de um conceito artístico.

**A frase “Está página foi intencionalmente deixada em branco”**

Essa frase é uma forma de o editor ou autor reconhecer que a página em branco não é um erro de impressão, mas sim uma escolha intencional. Isso pode ser feito para evitar que o leitor pense que houve uma falha na produção do livro.

Curiosidades

Algumas editoras usam as páginas em branco para adicionar informações extrar, como um espaço para o leitor escrever anotações ou um índice remissivo.

em edições especiais ou de luxo, as páginas em branco podem ser usadas para inserir ilustrações ou outros elementos gráficos.

Git e Github

Primeiramente é fundamental entender que Git e Github são coisas diferentes.

Git - É um software de Controle de versão ou (VCS)

Github - É uma plataforma de rede Social para programadores

Porém vamos nos aprofundar nestes termos.

Git

Sendo git um software de Controle de versão de versão, então isso faz do git uma ferramenta de *Versionamento.* O git pode ser utilizado para versionar outras coisas além de software vejamos algumas:

Documentos

**Textos:** Artigos, livros, roteiros, documentos de texto em geral podem ser visionados com Git, permitindo que você acompanhe as alterações, revise versões antigas e colabore com outras pessoas na escrita.

**Imagens**: Diagramas, ilustrações, fotos, e outros tipos de imagens podem ser versionados, facilitando o acompanhamento de edições e o trabalho em equipe em projetos visuais.

**Apresentações**: Slides de apresentações também podem ser versionados, garantindo que você tenha acesso a versões antigas e possa colaborar com outras pessoas na criação de apresentações.

**Outros tipos de arquivos**

**Dados**: Arquivos de dados, como planilhas, bancos de dados r arquivos de configuração, podem ser visionados para acompanhar alterações e facilitar a colaboração em projetos de análise de dados.

**Websites**: O código HTML, CSS, e JavaScript de um website pode ser versionado com Git, permitindo que você acompanhe as alterações, teste novas versões e volte a versões anteriores se necessário.

**Projetos de design:** Arquivos de design, como PSDs, AI e SVGs, podem ser visionados para facilitar o acompanhamento de edições e o trabalho em equipe em projetos de design.

O git permite que você veja todas as alterações feitas em um arquivo ao longo do tempo, facilitando a identificação de erros e a recuperação de versões antigas.

Versionamento

A palavra *versionamento* nos leva a outra, *versões* e para um programador é muito importante levar em consideração as versões, já que é completamente normal pra um programador lidar com muitas *versões* de uma mesma coisa, no caso softwares.

Exemplo:

Criar vários repositórios de um programa e para cada alteração significativa fazer um arquivo zip como “backup” para caso as versões futuras tenham algum tipo de erro, isso se torna um problema ainda maior quando tem mais de uma pessoa trabalhando em um mesmo projeto.

Porém vamos começar do inicio para entender tudo acerca dessa ferramenta tão útil para quem é programador.

História do software de versionamento

O primeiro software de versionamento que se tem registro, foi criado no ano de 1972, e o primeiro tipo de repositório que havia era do tipo *central* e para esse funcionar, tinha que enviar os arquivos diretamente para o repositório, mas não somente por internet, e toda a organização era feita pelo próprio servidor onde eram armazenadas esses versões, o servidor era capaz de reconhecer quais foram as atualizações significativas, levando em consideração todos os projetos, se havia mais de um programador trabalhando neste. Sendo assim essa modalidade de versionamento ou VSC ficou conhecida como *LINEAR* ou ***CENTRALIZADO***. Esse modelo evoluiu tanto que acabou surgindo outro modelo, uma versão mais aprimorada deste primeiro, esse é o modelo *DISTRIBUÍDO.*

Ao invez de fazer o commit para o repositório central, você agora baixa um software, onde você faz o commit diretamente na sua máquina (repositório local), desta maneira não é necessário ter o poder de processamento de um servidor, e nem precisa estar conectado a ele.

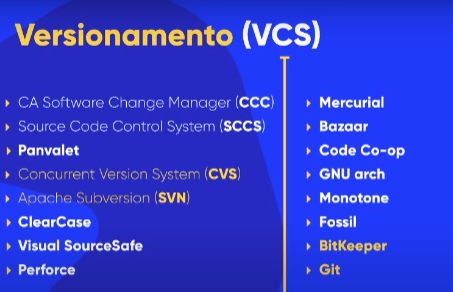
Mas isso gera uma dúvida, e no caso de precisar do commit dos outros programadores que estão do mesmo projeto?

Neste caso tem se o conceito de repositório remoto, que vai operar como um google drive, ou um deposito em nuvem.

Push

É uma expressão usada para quando o desenvolvedor quer pegar o versionamento que tem na máquina (repositório local) e encaminhar par aum repositório remoto.

Exemplos de alguns softwares que surgiram ao longo do tempo para versionamento de código:



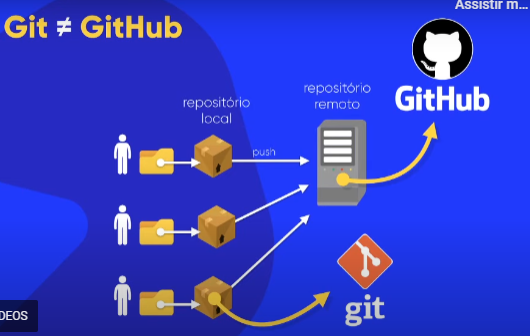
Algumas vantagens de se utilizar um software de controle de versão (versionamento).

Controle de histórico – Caso tenha algum erro nas versões futuras, basta voltar na última versão que antecedeu o erro para conseguir dar prosseguimento com o projeto, pode parecer uma coisa simples, mas quando se trata de trabalhar com mais de um desenvolvedor vai ser muito útil, já que cada um vai contribuir de uma maneira para aquela software funcionar, sendo assim isso se torna extremamente útil.

Ainda sobre trabalho em equipe, o git permite trabalhar com **ramificações de projeto,** ou seja “distribuir” funções para cada membro que forma aquela equipe.

Com isso também traz a questão da segurança, como cada membro vai ficar responsável por uma parte especifica do código, então isso não da margem para um desenvolvedor que está trabalhando em front, tentar mexer em algo de back-end, o que quando for fazer o *murge* do projeto, ou seja, juntar o projeto com cada ramificação, não dê problema.

Uma maneira fácil de visualizar o que é git e github:



O que é Github?

Github nada mais é que um repositório remoto.

O github não é o único repositório remoto que suporta git, existem outros:

Gitlab

Bitbucket

Phabricator

Gogs

Kallithea

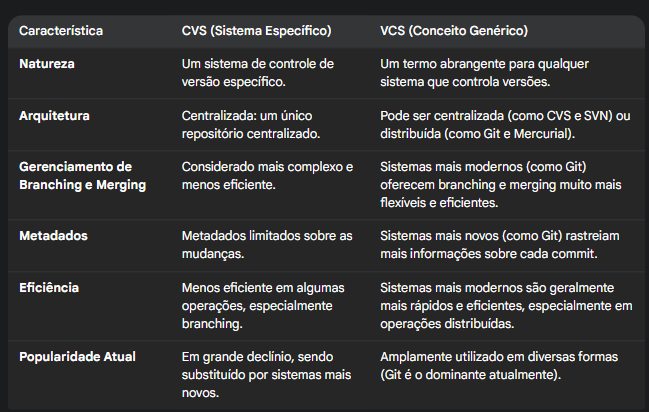
Diferença entre CVS e VCS

Pense assim:

VCS (Version Cotnrol System) É a categoria, o conceito geral de um sistema que rastreia as mudanças em arquivos ao longo do tempo, permitindo que você reverta para versões anteriores, compare alterações, trabalhe em equipe de forma eficiente, etc. Exemplos de VCS incluem Git, Subversion (SVN), Mercurial e, claro o próprio CVS.

CVS (current Versions System) é um dos sistemas de controle de versão mais antigos e historicamente significativos. Ele foi amplamente utilizado antes da ascensão de sistemas mais modernos como Git e SVN.

Veja mais a respeito a seguir:



Quem veio primeiro o Git ou Github?

- O CVS era um software centralizado ou seja ele precisa de um servidor para estar funcionando.

- Ele também é Open Source

- Na época de seu lançamento ele era o mais popular, por volta de 1985

- Porém ele tinha alguns problemas como, consistência e velocidade.

Alguns exemplos de coisas que o CVS era falho, era em coisas simples, como alterar um item de pasta, renomear uma pasta ou repositório, esse tipo de funcionalidade o CVS não permitia

Porém ele foi durante muito tempo um software extremamente popular, isso até o ano 2000 quando surgiu um software da fundação apache que é o SVN. O SVN é o subversion.

Se formos analisar o SVN em relação ao CVS vamos notar que eles são parecidos, ainda que o SVN foi lançado 15 anos depois a intenção era ser parecido com o CVS porém aqueles “problemas” que foram citados no CVS seriam corrigidos

Inclusive o SVN ele é ativo e atualizado até os dias de hoje!

E como a proposta do SVN era ser semelhante ao CVS então ele era muito parecido em muitas funcionalidades, algumas que não são tão agradáveis para os usuários que migraram do CVS para o SVN, porém alguns dos problemas que o CVS tinha foram corrigidos no SVN.

Alguns problemas que foram corrigidos, como instabilidade, menos tempo de espera ao fazer commits

No mesmo ano que surgiu o SVN 2000, surgiu outro software que tinha um paradigma diferente dos vistos anteriormente.

Os dois primeiros CVS e SVN eram centralizados. Porém no ano de 2000 surgiu outro que utilizada o sistema **distribuído**.

Sistema de Versionamento Centralizado (CVCS):

**Um único repositório Central:** Existe um servidor central que contém a “verdadeira” versão do projeto e todo o seu histórico.

**Check-out/Check-in:** Os desenvolvedores fazem um “check-out” (retiram) uma cópia dos arquivos do servidor central para trabalhar localmente. Após fazerem suas alterações, eles fazem um “check-in” (enviam) suas modificações de volta para o servidor central.

**Histórico no servidor:** Todo o histórico do projeto reside unicamente no servidor central, As cópias locais dos desenvolvedores geralmente contêm apenas a versão atual dos arquivos com os quais estão trabalhando.

**Dependência do servidor:** Se o servidor central falhar ou fica inacessível, os desenvolvedores não conseguem obter o histórico completo do projeto ou colaborar de forma eficaz.

**Branching e Merging:** O branching (criar linhas de desenvolvimento paralelas) e o merging (combinar as alterações de diferentes branches) geralmente são mais complexos e menos eficientes em sistemas centralizados.

**Exemplos de Sistemas Centralizados:**

* CVS (Concurrent Versions System)
* Subversion (SVN)
* Team Foundation Version Control (TFVC) (no modo centralizado)

**Sistema de Versionamento Distribuido (DVCS):**

**Repositório completo em cada cópia local:** Cada desenvolvedor tem uma cópia completa do repositório, incluindo todo o histórico do projeto, em sua máquina local.

**Clone/push/pull:** Os desenvolvedores “clonam” (copiam) o repositório inteiro para suas máquinas. Para compartilhar suas alterações, eles “puxam” (pull) as mudanças de outros e “empurram” (push) suas próprias mudanças para outros repositórios locais de colegas ou um repositório central remoto).

**Histórico distribuído:** O histórico do projeto é distribuído entre todas as cópias locais. Se o servidor central remoto falhar, qualquer cópia local pode servir como um backup completo do projeto.

**Independência do servidor (parcial):** Os desenvolvedores podem continuar trabalhando localmente e commitando suas alterações mesmo sem uma conexão com o repositório remoto. Eles sincronizam suas mudanças quando a conexão é restabelecida.

**Brancing e Merging:** O branching e o merging são geralmente muito mais rápidos, fáceis, e eficientes em sistemas distribuídos, pois essas operações são realizadas principalmente localmente.

**Exemplos de Sistemas Distribuídos:**

* Git
* Mercurial
* Bazaar

Tendo entendimento isso, e voltando a história sobre o software de versionamento, no ano de 2000 surgiu um software chamado de BitKeeper que como mencionado anteriormente tinha uma abordagem diferente dos dois até então utilizados, que era se utilizando do sistema Distribuido. O fundador desse software *Larry Mc’voil* criou uma empresa chama de BitMover. (O Larry antes de fundar a empresa, fez parte do desenvolvimento do kernel (Nucleo do sistema – feito em C) do sistema Linux). Voltando a história o Larry queria ganhar dinheiro com o BitKeeper, porém como ele fazia parte do desenvolvimento do kernel do Linux, ele também queria ter uma **Versão comunidade** do Bitkeeper.

E claro por ser uma versão Comunidade, não eram 100% dos recursos que eram disponibilizados, pois se não ficaria inviável ter recursos para manter a fundação.

Um termo que não foi comentado anteriormente quanto ao SVN é o termo CVS-like ou (igual CVS), o SVN tentava ser parecido com o CVS porém com algumas correções nos algoritmos e recursos do CVS logo o SVN é um CVS-LIKE;

Porém a Bitkeeper não tentava ser parecida com o CVS logo ela é considera um CVS-free;

Dentro da versão de comunidade de BitKeeper um dos maiores clientes foi o sistema Linux, e inclusive o kernel do Linux era disponibilizado através do sistema Bitkeeper.

Em 2004 um dos desenvolvedores que trabalharam neste software livre, o Andrew ele criou um programa para a comunidade do Linux utilizar. Esse programa se chamava SourcePuller, Explicando de uma maneira resumida, esse seria um cliente de BitKeeper.

Enfim o Andrew, se utilizando de uma engenharia Reversa do Bitkeeper comunity conseguiu destravar os recursos que não deveriam ser liberados na versão comunidade do BitKeeper, somente na versão completa, algo como um \*crack do programa. Claro que o fundador do BitKeeper não ficou contente com esse situação, foi quando teve uma rixa entre o Andrew e o MC’voil.

Então em 2005 foi lançada uma nova versão do BitKeeper com uma Nova licença, onde não tinha mais acesso aos Metadados do Bitkeeper, algo que existia na versão comunidade antes. E com isso a questão do versionamento em Si, a questão de voltar versões de um software de um programa de um commit foi dificultado por conta dessa rixa que existiu. Foi somente disponibilizada na versão comercial do BitKeeper.

Então o Linus Torvalds, um dos criadores do Kernel do Linux, ficou revoltado com o MC’voeld por conta dessa decisão de não deixar mais o software de versionamento guardar versões de software, sem o acesso pago. Então o Torvalds disse que faria seu próprio software de versionamento e esse criou o GIT.

E o Git foi criado em 2005 por conta desse evento que ocorreu, e o Torvalds criou esse software para ser Distribuido, Open Source, e uma versão usável desse software foi feita em 10 dias

Qual o significado de GIT

Não tem um significado semântico por traz da palavra GIt, na verdade o criador do git menciona qu esse nome é fácil de pronunciar, fácil de lembrar tem apenas três letras, então é algo extremamente simplificado!

O Linuz Storfield é alguém extremamente brincalhão por conta disso ele fez essas piadas com relação a criação do Git, mas não tem um real significado, o mesmo vale para o Linux, sistema esse que também foi desenvolvido por ele,

Quando surgiu o Github??

O gihub apesar da semelhança com o Git, foi criado em 2008, 3 anos depois da criação do Git, seus criadores foram:

- Chris Wanstrath

-PJ Hyett

-Tom Preston-Werner

-Scott Chacon

Algo de se comentar sobre o GitHub, é que ele não é um software livre, ele é um software proprietário, desses 4 elementos que pudemos conhecer acima!

Em 2009 Já haviam sido registrado mais ded 46 mil repositórios no Github, sendo que inicialmente, no primeiro ano ele não deixava repositórios privados, somente públicos.

Sendo assim o GitHub ficou em 14° na revista Forbes que nada mais é que uma lista de empresas de armazenamento em nuvem que vão crescer bastante, e o GitHub, aquele projeto que começou pequeno ficou em 14° lugar.

Em 2018 O Github recebeu o maior ataque de DDOS da História (negação de serviço), o que fez o site cair evidentemente; porém ao invez de desmotivar os criadores do Github naquele mesmo ano a Microsoft comprou o Github por 7.7 milhões de Dollares

Tendo entendido a origem do git e do gitbhub, vamos ver como podemos estar se utilizando dessas ferramentas.

O que é versionamento e qual sua importância na programação?

Versionamento, ou **Controle de Versão,** é o processo de gerenciar e rastrear as alterações em um conjunto de arquivos ao longo do tempo.

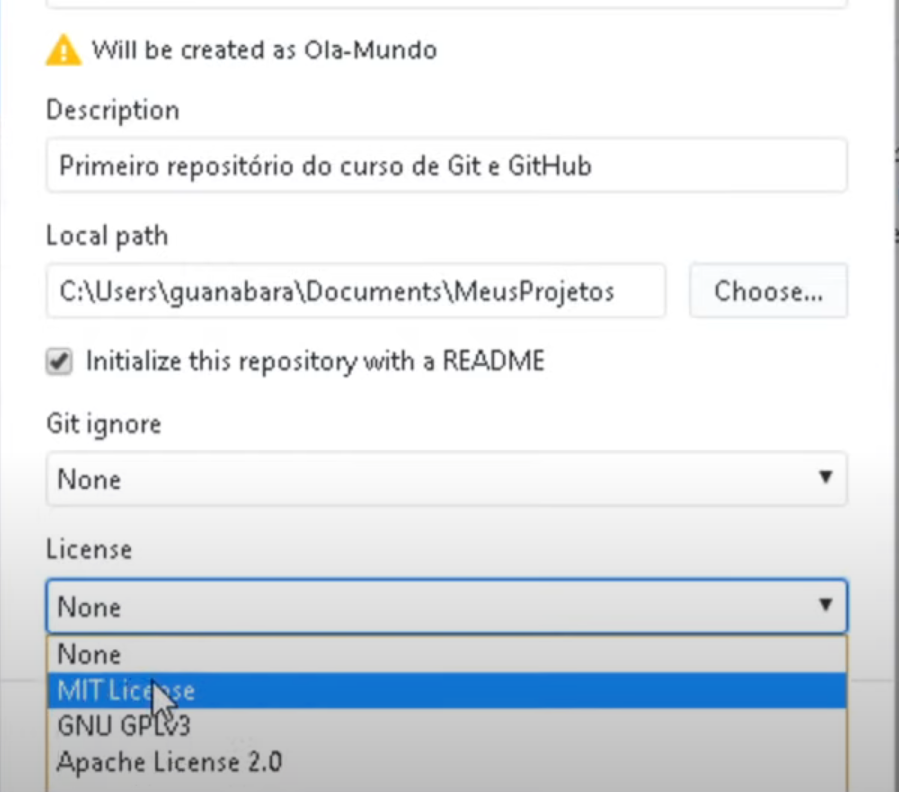
Pense em um documento que você edita. Você pode criar “Versão 1.0”, “versão 1.1 (com revisões)”, “versão 2.0 (final)”, etc. O versionamento faz isso de forma automatizada, inteligente e muito mais poderosa para arquivos de código.

Ele permite que você salve “instantâneos” (snapshots) de seu projeto a cada alterações significativa. Esses instantâneos são as versões.

Qual sua importância?

A importância é imensa, tanto para projetos individuais quanto para equipes:

Histórico completo: Você tem um registro de cada alteração feita, por quando. É como uma “máquina do tempo” para o seu código.



Em suma, quando for criar um repositório no github vai aparecer dessa maneira.

O local path, é onde estará armazenado localmente o arquivo que estará sendo desenvolvido, antes de ser upado para o github.

**Gitignore** é um arquivo que desempenha um papel fundamental no controle de versão com Git.

O arquivo .**gitignore** é um arquivo de texto simples que você cria e coloca na raiz do seu projeto Git. Ele contém uma lista de arquivos e pastas que o Git deve **ignorar** ao rastrear alterações. Em outras palavras, o Git vai “olhar para o outro lado” e não incluir esses arquivos nos seus “commits”.

Quanto a sua importância

A importância do .gitignore é crucial para manter o seu repositório de código limpo, organizado e eficiente. Ele evita que você inclua no controle de versão arquivos que não deveriam estar lá.

Imagina o seguinte cenário sem o .gitignore

Você está desenvolvendo um projeto web com JavaScript. Você tem o seu código-fonte, mas também tem uma pasta node\_modules com centenas de dependências (bibliotecas) que foram baixadas, arquivos de cache, logs de erro, arquivos de configuração locais (que contêm senhas ou chaves de API),etc.

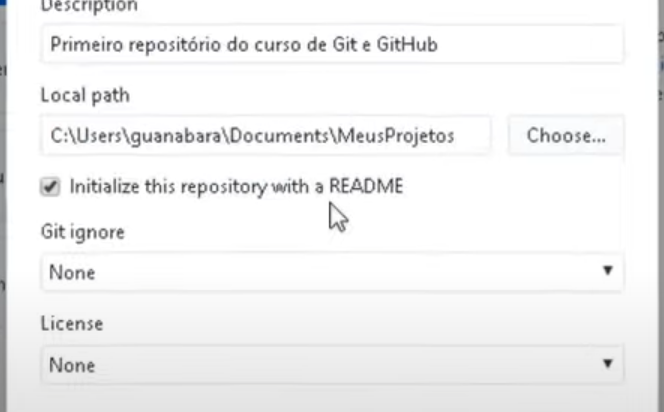
Se você fizesse um commit de tudo, o seu repositório ficaria

Enorme: O tamanho do respositório aumentaria exponencialmente, dificultando o download para outros desenvolvedores.

Bagunçado: O histórico de commits ficaria cheio de alterações irrelevantes.

Inseguro: Informações sensíveis (como senhas) poderiam ser enviadas para o repositório remoto (Github, por exemplo) tornando-as públicas.

Com conflitos: Arquivos de configuração específicos para o seu ambiente local (como uma porta de servidor ou um caminho de arquivo) gerariam conflitos constantes quando outros desenvolvedores baixassem o código.



A opção “Read me” é de suma importância quando for criar um repositório no github.

O arquivo Readme.md (onde .md significa Markdown, uma linguagem de marcação simples para formatação de texto) é o cartão de visitas do seu projeto. Ele é o primeiro arquivo que as pessoas veem quando acessam o seu repositório no GitHub.

Marcar essa opção é uma **Boa prática fundamental** por várias razões:

Primeira impressão e contexto:

Para Visitantes: É a primeira coisa que alguém que encontra seu repositório (um recrutador, um colega, um potencial Colaborador) vai ler. Ele fornece um resumo rápido do que é o projeto.

Para você mesmo: Ajuda a solidificar a ideia do projeto e a ter um ponto de partida para a documentação.

Documentação Essencial:

O que o projeto faz? Qual o propósito do seu código?

Em um portfólio do Github, um README bem escrito demonstra profissionalismo, organização e a capacidade de documentar seu trabalho. Recrutadores e gerentes de projeto valorizam muito isso.

Ele transforma um simples conjunto de arquivos em um projeto compreensível e apresentável.

Navegação e busca

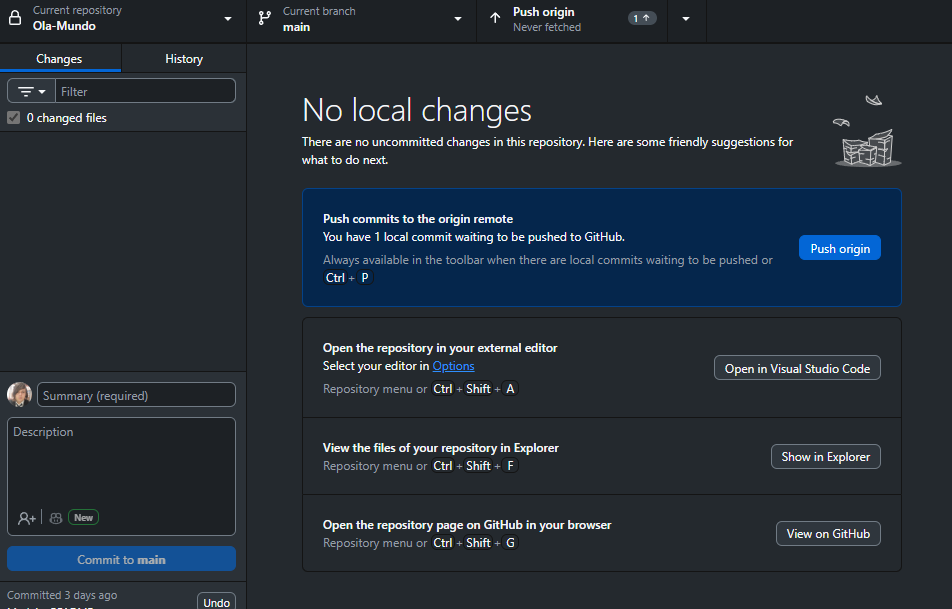
O conteúdo do README é indexado por motores de busca dentro do Giuthub, o que pode ajudar outras pessoas a encontrarem seu projeto.

Ponto de partida para colaboração:

Se você pretende que outras pessoas colaborem, o README é o guia inicial para que elas entendam o projeto e saibam como começar a trabalhar nele.

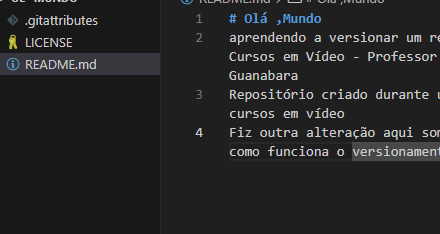
Dando continuidade com o projeto

Feito o projeto inicial para teste ele deve ficar da seguinte maneira:

Isso na plataforma do github desktop

Foi criado já a pasta, e pelo VS CODE e o github serem da Microsoft, eles são integrados, o que justifica o fato de poder utilizar o VS code para fazer alterações dentro do arquivo.

Lembrando que como foi adicionado a função “README” direto no momento em que foi criado o repositório

Quando for abrir o arquivo no VS code, vai aparecer da seguinte maneira: 

No VS code vai aparecer também a seguinte situação

Criando um repositório no git e github

O **GIT** é um sistema de controle de versão (VCS) distribuído e de código aberto. Pense nele como uma “máquina do tempo” para o seu código. Ele é uma ferramenta instalada localmente no seu computador que rastreia e gerencia todas as alterações feitas no arquivos do seu projeto ao longo do tempo.

Termos e conceitos importantes do Git.

**Repositório (Repo):** É o seu projeto. É um diretório com uma pasta oculta chamada .git que armazena todo o histórico de versões do projeto.

**Commit**: É um “instantâneo” ou um ponto de salvamento do seu código. Cada commit representa um conjunto de alteraç~eos feitas em um determinado momento e tem uma mensagem que descreve o que foi modificado.

**Branch (Ramo):** É uma linha ded desenvolvimetno paralela. Você criar um branch para trabalhar em uma nova funcionalidade sem afetar a versão principal do código (geralmente chama de **main** ou **master**).

O **Github** é uma plataforma baseada na web que usa o Git. Ele atua como um **repositório remoto**, um servidor na nuvem onde você pode hospedas o seu repositório Git. Elçe é uma plataforma de colaboração e comunidade para desevolvedores. O github Foi adquirido pela Microsoft em 2018.

Colaboração: Facilita o trabalho em equipe através de ferramentas como o Pull Requests (PRs), que são um pedido para “mesclar” suas alterações em uma branch principal.

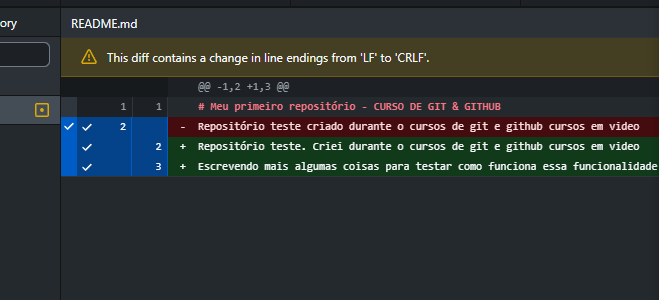
Comunidade: É um local para descobrir projetos de código aberto, colaborar com outros desenvolvedores e construir um portólio.

Quando você criar e abrir o seu repositório no Visual Studio Code, Algumas coisas vão acontecer.

A primeira delas vai ser o seguinte botão:

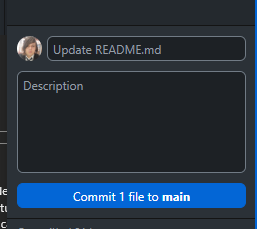


(Controle de versão). A funcionalidade disso seria de fazer alterações no código fonte do projeto, sendo assim qualquer pessoa pode fazer uma alteração e isso vai constar na páginas “changes” ou mudanças no Github Desktop como pode ser visto à seguir:



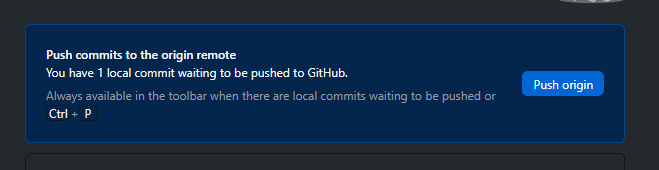
Neste exemplo o que está em vermelho era o que tinha no código, e o que está em verde foi a atualização, a adição que foi feita no código. Neste caso estamos se utilizando de mensagens para ficar fácil entendimento, porém imagine em um contexto onde isso seria códigos!

Mas somente fazer essa alteração faz com que a alteração seja versionada, apenas alterada, para versionar



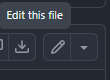
Seria neste campo que fica abaixo no github desktop.

Outra funcionalidade interessante, que quando feita a alteração e commit de um código, ele vai enviar salvar “localmente” esse commit, para enviar esse commit para o repositório github e para isso de maneira intuitiva vai aparecer da seguinte maneira:



Essa mensagem aparece muito por conta de já ter um repositório local que foi feita a alteração, porém que não foi enviada para o servidor para arquivar essa alteração.

Dentro do site do Github ainda tem uma funcionalidade de editar arquivos que vai ser mostrado pelo seguinte ícone:



Neste ícone de “lápis” você pode fazer alguma alteração. Não é muito recomendado por conta de ser um editor de código extremamente simples, porém dependendo do contexto pode ser uma solução interessante.

Commitar é sempre fazer uma alteração no repositório

Com isso surge a seguinte questão, e se for feita alguma alteração no repositório remoto, porém não tiver salva no repositório local?

Neste caso tem uma maneira de acessar e trazer do repositório remoto as alterações feitas, que seria através do ícone representado à seguir:



Esse ícone esta no programada do Github Desktop, você clicando essa opção, você checa se tem alguma alteração que foi feita no repositório remoto.

Depois de verificado o ícone vai ficar da seguinte maneira:

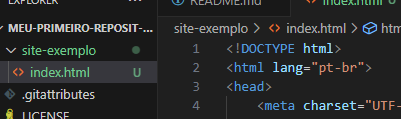


Sabendo que Push e “empurrar os arquivos do repositório local para o repositório remoto, então Pull é trazer commits, mudanças que tenha um repositório remoto mas que ainda não tem no repositório local

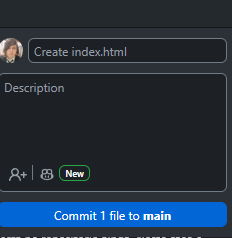
E pronto dessa maneira vamos ter uma integração do VSCODE com o Git com o Github. É algo extremamente simples, mas que pode fazer uma grande diferença para quem está aprendendo a programação.

Feito isso basta seguir o passo padrão de cada programação, que é criar uma pasta com o nome do arquivo, criar um código em alguma linguagem de programação que seja do seu interesse.

No VS code vai aparecer da seguinte maneira quando for feita uma adição no repositório:



O U seria “untracked” isso porque foi feita uma adição que não está no repositório ainda. Neste caso e preciso fazer o commit para o repositório local através do método mostrado anteriormente



A parte de trazer as informações que foram adicionadas, será feira de maneira automática pelo próprio Git

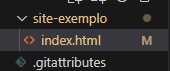
Para dentro do Github Desktop

E a pasta que acabou de ser criada vai aparecer da seguinte maneira depois de feito o Push:



Isso cria uma situação

Como antes o arquivo index do exemplo não estava sendo monitorado ele aparecida como U ou Untracked, porém uma vez que ele foi comitado, agora ele vai aparecer da seguinte maneira



O u, tornou-se um M porque antes era untracked agora é um arquivo modificado (isso porque foi feito mais alterações nele)

Algumas teclas de atalho

Quando você terminar de fazer uma alteração e quiser fazer um commit para o repositório local basta adicionar o nome do commit e apertar ctrl + Enter para que seja feito o commit para o repositório local

Para exportar arquivos para o repositório remoto (Push) **ctrl + P**

**Clonar um repositório**

Introdução ao conteúdo.

Clonar (git clone)

Clonar e o processo de criar uma cópia exata e completa de um repositório Git existente (geralmente de um servidor remoto como o Github) para o seu computador local.

Como funciona?

Você usa o comando **git clone <url\_do\_repositorio>** no seu terminal. Isso baixa todo o histórico de commits, todas as branches e todos os arquivos do projeto para um novo diretório na sua máquina. O Git automaticamente configura a cópia local para ter uma conexão com o repositório original (chamado de **origin**).

Importância:

Iniciar o trabalho. É a maneira padrão de começar a trabalhar em um projeto que já existe.

Sincronização: permite que você tenha uma copia para fazer alterações, testar e depois enviar (push) suas modificações de volta para o repositório remoto.

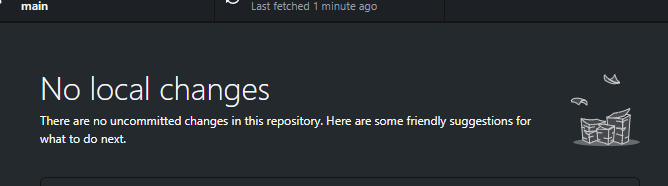
Boas práticas quando for trabalhar

Sempre que for começar a trabalhar, o ideal é sempre fazer um fetch no github para saber se está tudo de acordo, se as versões do repositório remoto e local estão sincronizadas



[opção acima]

Mão havendo nenhuma mudança, na interface do github vai aparecer da seguinte maneira:



E como fazer para clonar um repositório

“Só lembrando que repositório é uma área onde tem um código que está versionado, esse repositório podendo ser público ou privado”

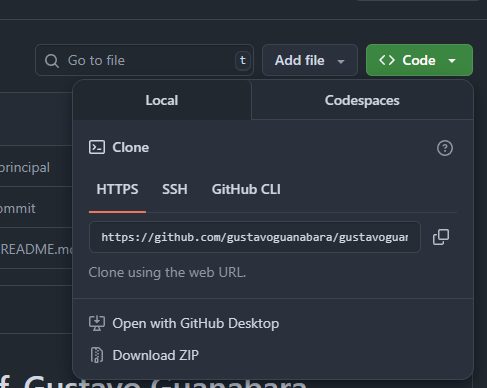
Explicando o que é clonar um repositório, é pegar um repositório que esta publico e baixar ele na máquina de tal maneira que o git faça o versionamento local

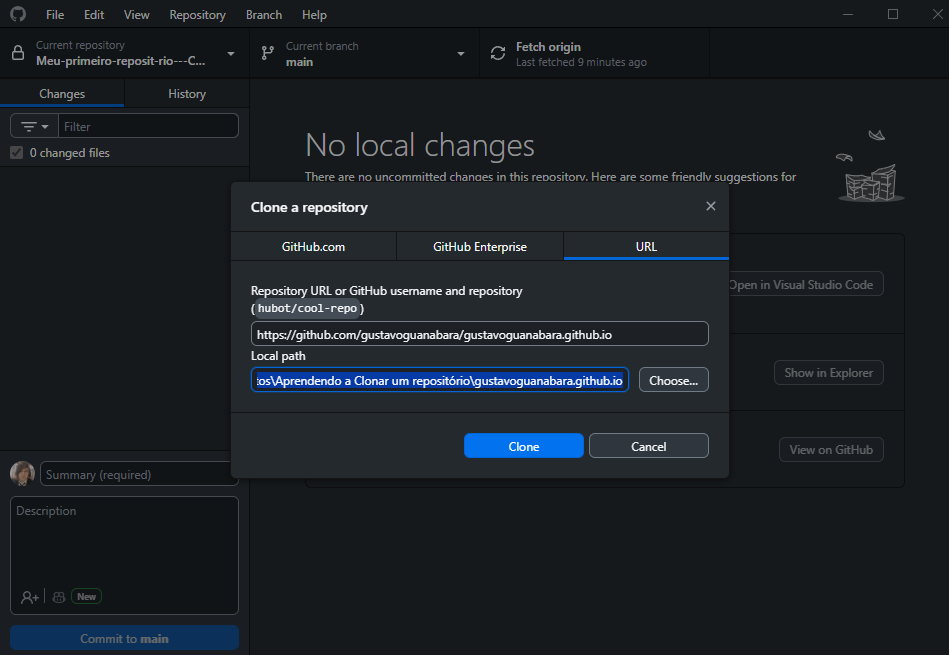
No github em questão que você queira estar clonando para o seu repositório local

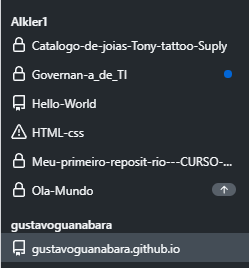
Vai aparecer a seguinte caixa

Na opção <> CODE basta escolher a opção “Open Whith Github Desktop”

Escolhendo essa opção vai aparecer uma segunda caixa de texto do navegador pedindo autorização para abrir esse repositório com o Github desktop, confirmando vai aparecer a as informações de origen do repositório, no caso de quem está sendo clonado, e para onde esse arquivo que está sendo clonado vai ficar localmente na máquina;

Nesse processo de clonagem de repositório, o Github traz tudo que inclui no repositório de origem, se no repositório de origem de arquivos de imagem, de audio, de video, independendo do tamanho ou formado, será clonado da origem e feito um clone local do repositório

É dessa maneira que vai aparecer o documento após a autorização de deixar o navegador acessar direto o Github Desktop.



Detalhe para a maneira como aparece no Github quando for acessar outros repositórios, aparece marcado de quem é o repositório, neste exemplo Alkler1 é o repositório próprio, e Gustavoguanabara é outra pessoa, então é essa separação de quem é o “proprietário” fornecedor do código original.

Detalhe para como pode ser utilizado isso, é uma excelente maneira de ter essa separação de código.

Feito isso é possivel escolher aonde esse repositório que foi clonado de terceiros vai ser salvo, bastando apenas mudar o nome no “save path” colocando na ordem da pasta em questão que o arquivo será salvo

Alterações em repositórios

Tendo entendido isso, vamos para outra questão...

No caso de identificação de um erro, ou de uma atualização que se faça necessária no repositório que foi clonado, é possivel fazer uma atualização e upar para o repositório principal essa atualização?

No caso a maneira de fazer isso seria através das ISSUES que são feedbacks que o usuário pode fazer para o dono do repositório principal

Exemplo

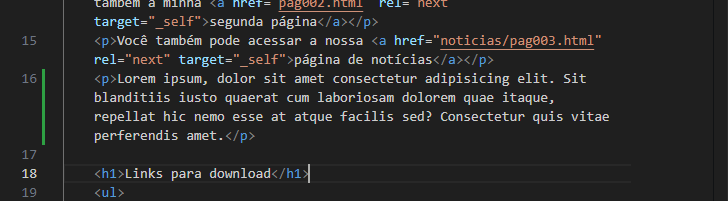
No caso de feita alguma alteração no código do repositório que foi clonado, vai aparecer um ícone da seguinte maneira:



Esse ícone que representa os Branches, ou ramificações. Isso vai significar que havia uma versão anterior do repositório no Git, e o Git já identifica que tem uma alteração.

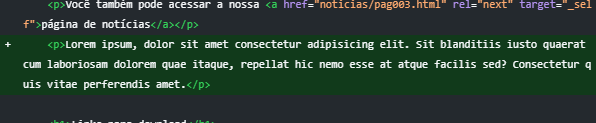
Se utilizando o código do repositório do curso de HTML e CSS do professor Gustavo Guanabara como exemplo, vamos observar que foi adicionado uma linha que não estava no código original

GRIFADA EM VERDE

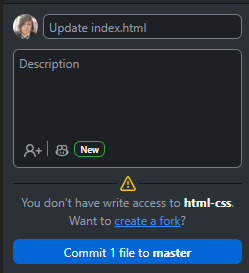


Quando feita essa adição no repositório principal, ela que está marcada tanto no VS CODE quanto

No GithubDesktop, e vai aparecer da seguinte maneira



Porém se tratando de uma alteração feita no repositório de outro usuário, o github não permite fazer o commit dessa atualização, neste caso vai aparecer a seguinte mensagem:



Que é basicamente que o meu usuário não tem permissão de fazer um commit para esse arquivo por conta do arquivo não ser do meu usuário, e esse não ter autorização para fazer alterações.

Até porque se todos pudessem fazer alterações no repositório, certamente haveriam projetos que poderia ser prejudicados com isso.

Então você pode, atravez de issues, branches e forks, sugerir alterações no repositório principal de quem o fez

E o Fork é exatamente isso, no caso do responsável pelo projeto decidir não aceitar a atualização que fora feita, você pode através do Fork, continuar o projeto a sua maneira, tendo como base esse projeto “principal” do criador que recusou sua alteração.

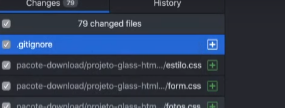
Como versionar projetos Antigos

É mais simples do que parece, para poder versionar projetos antigos, basta ir no Github desktop, criar um novo projeto, quando for criado esse novo projeto, ele vai “criar uma pasta de versionamento” que seria basicamente o local onde os arquivos serão adicionador para então ser versionado.

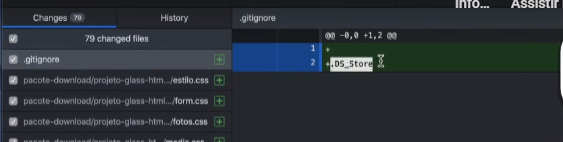
Isso já levanta uma outra questão também interessante, que é o fato de que, por se tratar de uma “pasta já versionada” poderia ser adicionado qualquer arquivo dentro desse projeto, porém de certa forma é ideal tomar cuidado com o tipo de arquivo que vai ser armazenado dentro dessa pasta, já que o ideal é sempre manter a organização e deixar arquivos relacionados ao projeto!

Sobre o Gitignore

O Gitignore é algo um tanto importante, vamos entender como funciona:



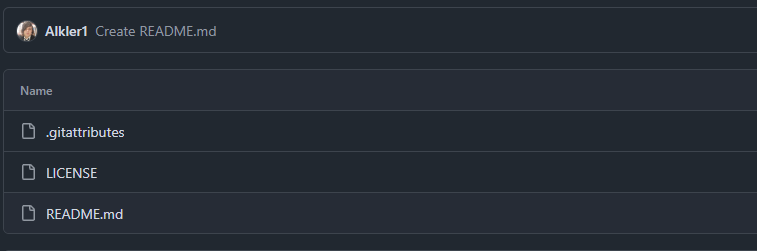
Quando for feito o versionamento de um item, o que vai acontecer é que pode ser que tenham itens, que talvez não sejam interessantes de estar subindo para o GitHub, sendo assim você pode adicionar arquivos que você não quer que sejam upados para o Github, ao invez de aparecer “excluir” ou “delete” ele encaminha o item em questão para dentro de GitIgnore, que é o arquivo que estará disponível apenas na sua máquina, funciona como se fosse um cofre de arquivos que você quer apenas deixar na sua máquina; E isso acaba sendo interessante, por exemplo no caso de ter um manual, um guia, dicas recomendações, observações acerca do projeto, mas mais pra si mesmo!



No exemplo do video em questão, havia um Arquivo chamado DS\_Store entre os arquivos do projeto que foi utilizado como exemplo.

A partir do momento que se coloca o item em questão dentro de Gitignore, isso cria o arquivo Gitignore, para poder armazenar esse arquivo que será dispensado!

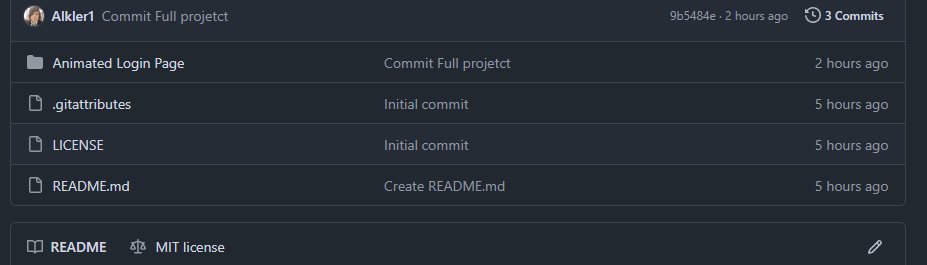
Em um primeiro momento o código vai ficar da seguinte maneira:



Isso porque isso seria essa “paste versionada” onde vão ser adicionados os itens que serão upados.

Indo em Push, para adicionar os itens que irão preencher a pasta.

Feita a devida atualização o repositório no Github vai passar a ser exibido da seguinte maneira:



Conceitos comuns em HTML

**Brenches:** Uma *Branch* é uma linha de desenvolvimento isolada. Imagine que o seu projeto é uma árvore. A *branch* principal (**main** ou **master**) é o tronco, e uma nova *branch*é um galho que você cria a partir desse tronco para trabalhar em uma nova funcionalidade ou correção de bug, sem afetar o código principal.

**Fork**: é uma cópia completa de um repositório de outra pessoa na sua própria conta do Github. É como pegar o projeto de alguém e fazer uma versão sua para poder mexer nele à vontade.

**Para que serve**? É o principal método de colaboração em projetos de código aberto. Se você quer contribuir para um projeto que não é seu, você faz um *fork* dele, trabalha na sua cópia depois sugere as mudanças para o dono do projeto.

O que é Issues?

Issues, é um problema, uma questão ou um levantamento de alguma coisa vamos elaborar isso com um exemplo para explicar também o que é um pull request.

Quando você tem um problema em um código e precisa de ajuda para resolver um problema, você tem uma Issues (problema).

Quando você já é alguém mais experiente, e você vê uma issues, você baixa uma Fork do código que está com esse problema, cria uma Branche onde você pode mexer no código na sua máquina, e Pull requeste é enviar esse problema para inicialmente criou o código problemático para ver se aquela solução que você encontrou para resolver o problema, é uma solicitação que faz sentido.

Se a solução que foi encontrada para o problema fizer sentido, e o desenvolvedor concordar com ela, sua ramificação [branche] passa a fazer parte do código principal do projeto; E você vai levar o devido crédito pela solução, entrando como colaborador do projeto que foi ajudado a resolver.

Uma maneira de visualizar Issues é se utilizando do próprio VS Code.

O VS Code é um programa da Microsoft para desenvolvedores e por ser uma ferramenta Open Source, o código dela é aberto e disponibilizado dentro do próprio Repositório da Microsoft, veja a seguir:

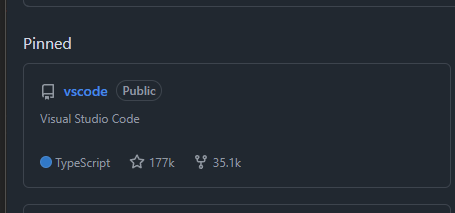
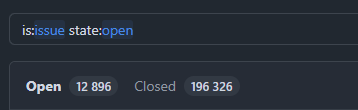


Imagem do Repositório do Proprio VS CODE de código aberto, disponível na página da própria Microsoft no Github



Seção Issues dentro do repositório



Aqui é onde você pode pesquisar algum erro que pode ter acontecido, exemplo somente. Mas você pode pesquisar caso tenha tido alguma falha, bug, ou problema no Github, então outros desenvolvedores vão tentar resolver, ou dar soluções para o problema em questão (caso esse ainda não tenha)

É recomendado pesquisar por código de erro

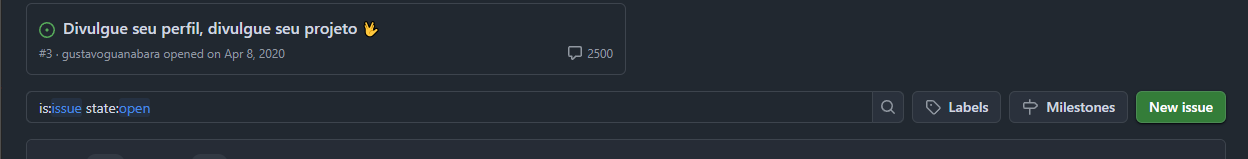
Outro detalhe que é importante Salientar, que antes de abrir uma Issue para um erro, sempre procure Issues para ver se alguma responde a sua questão. Porque entra na questão da Issue Duplicada, muitas dessas Issues “Closed” na imagem acima não são problemas que o VS CODE tem, são Issues, que foram fechadas por conta de serem ou Spam, ou ser uma Issue duplicada que já teve um respaldo. Então é sempre muito importante procurar a resposta de maneira adequada antes de criar uma Issue.

Outro ponto para justificar a falta de necessidade de ficar criando Issues, é por conta de, no caso de ser um Issue que são foi resolvida, isso passa uma visão ruim, de falta de vontade ou mesmo preguiça de procurar uma informação que já está disponível; Isso não é bem visto por desenvolvedores, é de mal tom fazer isso, porque na primeira vez que essa Issue apareceu, toda a comunidade teve um cuidado para tentar chegar a uma conclusão para resolver aquele problema, Ai alguém ignorar esse esforço coletivo que teve por conta de falta de vontade de pesquisar ou de procurar a informação, não é bem visto pela comunidade.

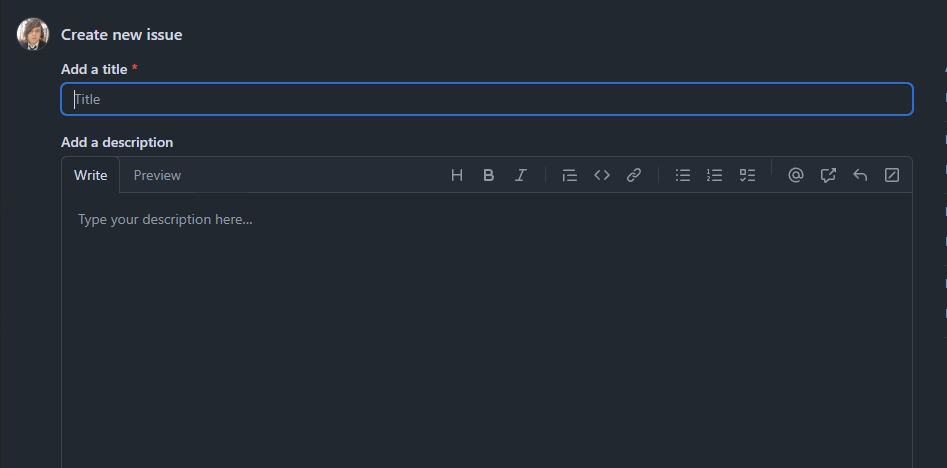
Dito isso, sempre recomendado procurar de maneira meticulosa com calma se não há resolução para a Issue que você estava procurando, realmente não tendo, ai você abre uma Issue, para que a comunidade então ajude a resolver, de preferente explicando com o máximo de informações acerca do problema, pode ser com: print, imagem, exemplo prático, como aconteceu, condição para acontecer etc.

Como criar uma Issue?

Tudo bem, agora vamos simular que você tenha um problema, porém não conseguir achar a resposta para o problema que você tem, ou está com uma dúvida e queira esclarecer. Neste caso você vai acessar o repositório principal equivalente ao que está te gerando dúvida, depois de fazer a pesquisa adequada para consultar se vai ter a resposta para a sua dúvida. Você vai abrir a issue, isso dentro da janela Issue que já se encontra dentro do próprio repositório:

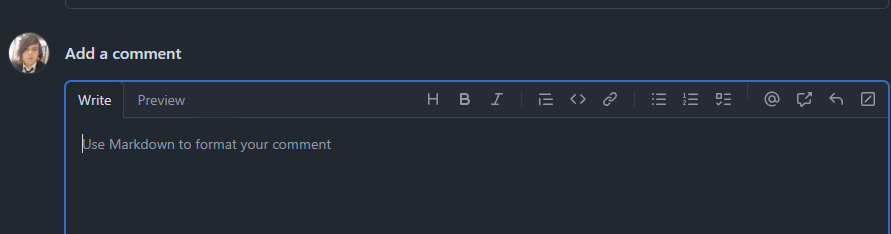


Quando for criar a Issue, vai aparecer da seguinte maneira:



Onde você vai dar o título pra sua Issue, o ideal é ser sempre direto ao ponto como “Onde está tal questão” Ou “Código de erro XXX em Python” etc.

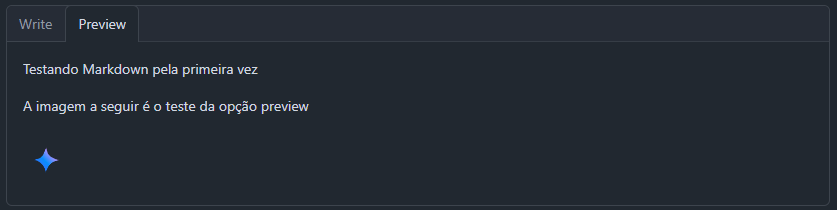
E quando for responder a mensagem, seja para ajudar, dar uma resposta, interagir, vai ser da seguinte maneira:



No ícone de Link você pode gerar uma frase destacada com Link dentro, como geralmente aparece um link mesmo:

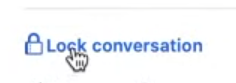
Clique aqui

Normalmente como escrito acima, o mesmo vale para imagens, que na opção Write, vai aparecer por extenso, porém na opção preview, você pode ver como vai ser exibido seu comentário:

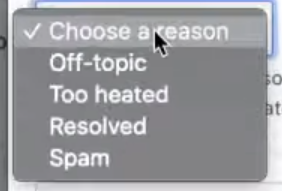


No caso da Issue ser resolvida, o Dono do repositório que foi feita a Issue, pode fechar a Issue, assim não será mais possivel fazer comentários.

Depois de fechar a Issue você pode ir na seguinte opção:



Que é onde vai ser comunicado o motivo do fechamento da Issue



Off-topic: é quando a Issue que foi aberta não tem haver com o repositório. Exemplo alguém fazer uma pergunta sobre Python em um repositório que tem apenas HTML e CSS. Esta fora do tema do repositório.

Too heated: seria muita “bagunça” dentro da Issue, quando estão falando de coisas não relacionadas da Issue de maneira excessiva.

Resolved: Resolvido o motivo que levou a criação da Issue

Spam: Quando estão Criando Issues com um tema que já foi resolvido.

Isso pode gerar uma dúvida; Porque somente o dono do repositório pode fechar uma Issue? Não deveria ser a pessoa que está com a dúvida que deveria fechar o repositório, baseado em se a resposta o ajudou ou não?

No geral, sim! Porém A pessoa quem gerou o repositório, provavelmente vai testar o que foi sugerido e determinar se acrescenta ou não no projeto, no caso de ser uma dúvida que tem uma resposta que somente o criador do repositório [main branches] vai ter, como por exemplo: (Porque usar o código Style dentro da CSS e não direto dentro da tag), é uma pergunta que o criador do repositório vai saber responder, por se tratar de uma escolha auto conciente, então ele sabe o motivo de ter feito então ele sabe qual a resposta que vai satisfazer.

A outra ocasião alguém gerar uma Issue, e você aguardar que a pessoa que fez a issue sinta se satisfeita com sua resposta, ou alguém da comunidade dê uma resposta satisfatória para aquela quem gerou a Issue. A resposta da Issue não está limitada somente ao criador, pessoas da comunidade que vejam a Issue podem tentar chegar a uma conclusão e passar para aquela quem gerou a Issue, sem necessariamente contar com a aprovação ou vistoria da criador do repositório.

O importante é fechar a Issue com uma resposta satisfatória. Se você ou alguém da comunidade foi capaz de resolver o motivo que gerou a Issue você pode fecha-la como “resolvida”.

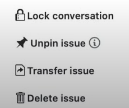
Exemplo de Issue feita:



Outra coisa interessante de se fazer com relação A Issues, é deixá-la pinada, ou fixada para assim essa tenha mais visibilidade, e ser exibida no topo das Issues. Isso pode ser feito na seguinte opção

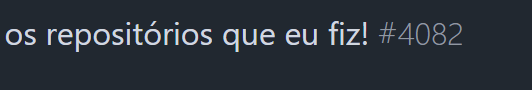


Que também fica na tabela de coisas a serem feitas com issues

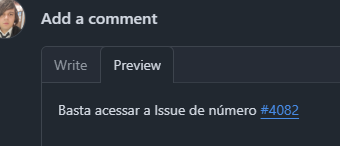


Outro detalhe interessante também sobre as issues, assim como é possivel marcas pessoas, usuários, é possivel marcas também Issues entre si, e não é através da URL da página, e sim através das tags da Issues.

Cada Issue que você cria vai gerar um número de tag, como pode ser visto a seguir:



Se você apenas colocar o número da tag dentro de uma resposta de uma Issue, isso vai gerar um Link:



Apertando em cima do número, você vai ser direcionada para a Issue equivalente ao número, bem como um link mesmo.

Linguagem markdown

**Markdown** é uma linguagem simples (como HTML, mas muito mais fácil) projetada para que você possa escrever usando um formato de texto plano que é fácil de ler e fácil de escrever. O objetivo principal dela é ser convertida em HTML de forma rápida e eficiente.

Pense no markdown como uma forma de **formatar texto** usando apenas caracteres simples, como asteriscos (\*), hashtags (#), e hífens (-).

Readme.md

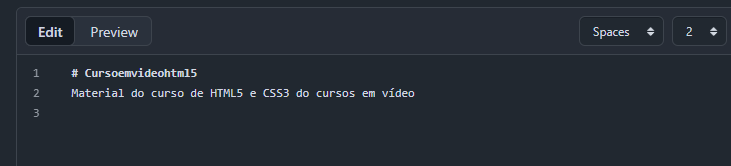
Se você notar dentro da pasta de um arquivo do Github, vai ter um arquivo chamado “readme.md”



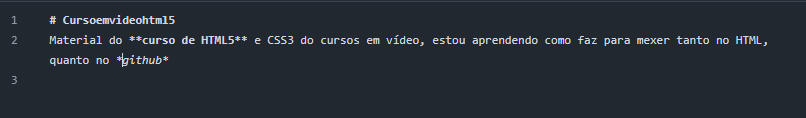
Esse MD, significa **Markdown**.

A proposta dessa **linguagem markdown** é ser uma linguagem semelhante ao que é o HTML. Porém o HTML é uma linguagem de dificl leitura, segundo aquele que criou o **markdown**, isso porque o HTML geralmente vai ter variável com nome de palavras, então acaba se tornando dificil ler a linguagem HTML, e por conta disso foi criado a linguagem **markdown.**

E é possivel ver essa linguagem, abrindo o arquivo README.MD



Por mais que seja uma linguagem de marcação, se você notar bem, diferente do HTML que teria uma tag como (Strong, H1, h2, P, Li,) nessa linguagem MD, não tem nada além da informação.



Ao invez de tags como “Strong” para deixar uma palavra em negrito, a maneira como é feito com a linguagem MB, é mais simples, é algo que não atrapalha a maneira de ler o conteúdo. No exemplo de cima foi apenas utilizado Asteriscos para conseguir deixar uma palavra em itálico, ou negrito.

Markdown

Seguindo o aprendizado sobre Markdown, vamos ver algumas coisas importantes de se saber.

Por exemplo para escrever algo em Negrito utilizando a linguagem de Markdown, basta se utilizar de uma escrita assim:

\_\_**negrito**\_\_ ou \*\***negrito**\*\*

O mesmo vale para itálico

\**itálico*\* ou \_*italico*\_

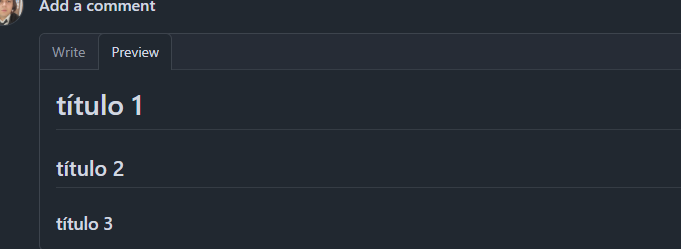
E para escrever títulos usando a linguagem de markdown vamos utilizar hashtags #

# título de nível 1

## título de nível 2

### título de nível 3

O resultado vai ficar da seguinte maneira:

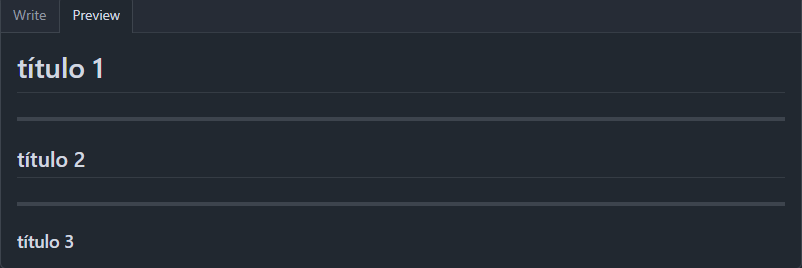
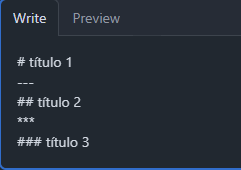


Algo bem semelhante ao que acontece com h1, h2, h3 no caso de HTML

Por padrão depois de um título vai ter uma linha bem discreta separando o título do restante do conteúdo.

Porém é possivel criar uma linha, caso você queira separar seções dentro do comentário, mas sem criar um título, e para isso utilizamos \*\*\* ou ---

O resultado vai ficar da seguinte maneira:



Algo semelhante com o próprio Whats’sapp é a questão de misturar os símbolos

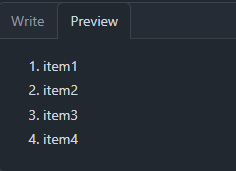
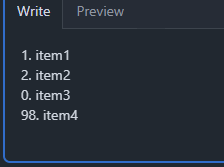
Como por exemplo para deixar mais destacado uma mensagem

\_\_\*~~***~~conteúdo em negrito e ítalico~~~~***\*\_\_

Sobre listas

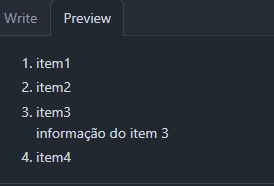
Assim também como HTML, tem listas na linguagem Markdown

Vamos ver a lista ordenada, e para isso podemos colocar um número, podendo ser na ordem ou não, a própria linguagem coloca em ordem



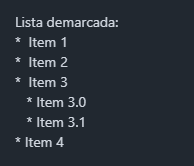
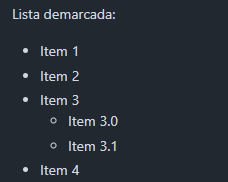
Ainda que a maneira como foi escrita, esteja fora de ordem; por se tratar de uma lista ordenada, a própria linguagem coloca os itens em ordem.

E para colocar um item de uma lista como sendo um subproduto de um dos itens, basta colocar 3 espaços antes do conteúdo

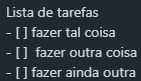
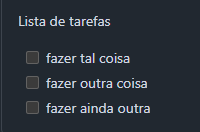


Lista demarcada:

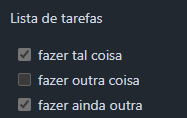
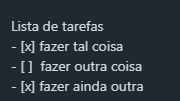
Para fazer isso basta utilizar \*



Lista de tarefas



E quando a tarefa estiver concluída, basta colocar um X dentro da tarefa, que a lista completa sozinha:



A linguagem Markdown, pode ser utilizada ainda fora go Github.

Documentação Técnica: Muitas ferramentas e bibliotecas de software usam arquivos **.md** para escrever sua documentação. É muito mais rápido escrever um guia de instalação ou um tutorial em Markdown do que em HTML ou em editor de texto completo.

READMEs Locais: mesmo projetos que não estão no Github frequentemente usam um arquivo **README.MD** no diretório principal para fornecer uma visão geral rápida do projeto para quem for trabalhar nele localmente.

Edição de conteúdo em CMS e Blogs

Muitos sistemas de gerenciamento de conteúdo (CMS) e plataformas de blog oferecem suporte nativo ou plugins para escrever posts usando Markdown.

**Jekyll, Hugo, Gatsby:** Estes são geradores de sites estáticos (muito populares entre desenvolvedores) que se baseiam em arquivos Markdown para criar as páginas do site.

**Plataformas de Blog:** O DEV Community, o Hashnode e muitas outras plataformas aceitam Markdown como a principal forma de formatar o corpo do seu artigo.

**Aplicativos de notas e produtividade**

A simplicidade e a capacidade de converter facilmente para HTML tornaram o Markdown a linguagem favorita para anotações e organização pessoal.

Aplicativos de notas: Aplicativos como **obsidian, Notion, VS Code** (Com plugins), **Typora** e **Joplin** usam Markdown como seu formato padrão de armazenamento de notas. Isso garante que suas notas sejam portáteis e fáceis de ler em qualquer editor de texto.

**Listas de tarefas**: Pode criar listas de tarefas formatadas de forma clara usando o Markdown em muitas dessas ferramentas.

Branches

Branche do inglês (*ramo*)

E a intenção é semelhante com a estrutura de uma arvore mesmo. O tronco da arvore, representaria o *branche Master*, que seria a estrutura principal.

Esse ramo master, é o que chamamos de ramo de produção, é o produto final, versão corrente. Por exemplo em um caso onde exista um programa que está em sua versão 4.0 e operando, essa versão seria a branche master, seria a versão principal. Claro que a equipe vai criar novas branches de atualização, que serão branches novas do projeto, que em lagum momento por vir a se tornar a branche principal.

No caso se alguém descobrir um erro, bug, ou abrir uma issue para a branche principal. Você pode criar uma branche somente para resolver esse problema/bug/issue, e depois commitar na branche master, que é o que é chamado de Murge

Origin

Origin, é tirar do seu repositório local e colocar em seu repositório remoto.

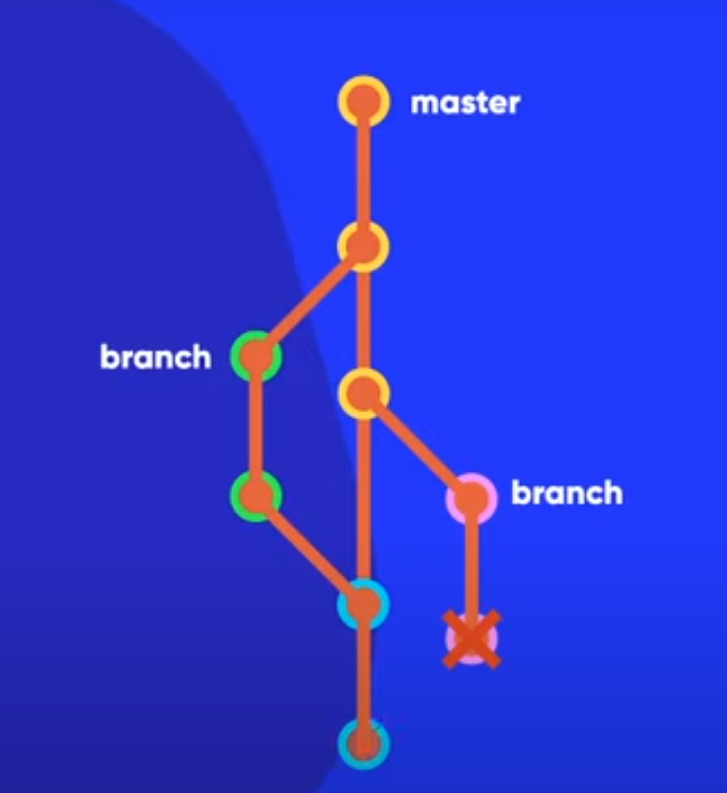
Quando é feito um Commit, você envia para o repositório local; quando é feito um push você está enviando para o repositório remoto.

Cada commit que foi feito na Branche Master, é um versionamento novo.

Quando estamos começando no mundo da programação, é muito comum a gente commitar na master, na branche Master, ou seja na ramificação principal do projeto.

Porém o ideal é fazer commits em branches separadas da master, porque quando você esta testando uma feature nova para a branche master, pode, e provavelmente vai ter muitos bugs, erros, coisas que precisam ser alteradas, então para não comprometer a branche principal com uma atualização falha, é feito dessa maneira. E quando essa atualização que foi feita em uma branche separada estiver pronta, já devidamente testada, ai basta fazer uma **merge** para a branche principal. Onde o arquivo da branche da ramificação separada vai passar a entrar como a branche principal.

O resultado vai ficar semelhante a imagem a seguir:



E você pode tentar desenvolver uma branche separada com uma feature nova, caso essa realmente não dê certo, você pode descartar ela, porque como se trata de uma ramificação ela não afeta a verão principal do programa que está sendo desenvolvido.

Vamos ver a aplicação desses conceitos em um repositório no Github.

Para tal, criamos um repositório no Github onde vamos utilizar como teste para ver o funcionamento das ramificações.

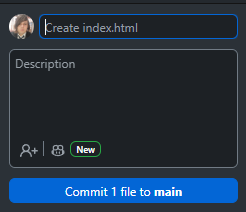
A partir do momento que for criado um repositório. Esse passa a ser a Branche Master, ou seja, qualquer alteração e commit feito agora será considerado um commit direto na branche master

Vai aparecer da seguinte maneira no próprio programa do Github desktop



Se referindo a Master. Neste exemplo está sendo chamado de “main” ou “principal” que seria o mesmo que a Branche Master.

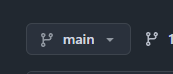
E quando for fazer um commit de um arquivo vai aparecer da seguinte maneira:



“commit 1 file to **main**”

Mesmo dentro do github ele deixa destacado que esse commit está sendo feito na **branche master**.

Feito o commit, vai aparecer da seguinte maneira no Github:



Essa seria a branche principal do arquivo, bem como foi feito o commit.

E aqui se segue uma dica muito importante, para qualquer atualização que seja feita, ou adição dentro do repositório do Github, sempre faça essas atualização em Branches separadas da master.

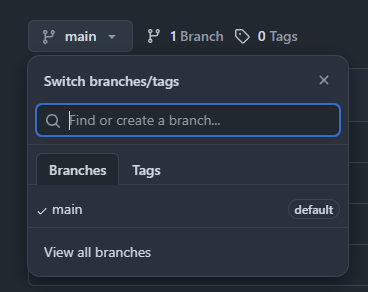
Para criar uma Branch, basta utilizar os recursos que estão descrito na barra de ferramenta do Github Desktop:



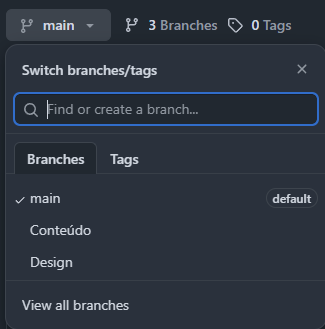
Na opção Branch, você pode estar criando uma Branch.

No exemplo em que estamos criando, vamos criar duas branches, uma chamada **conteúdo** outra chamada de **design**, isso porque estamos criando um exemplo baseado na programação de um site, por conta disso as nomenclaturas.

Em um primeiro momento, dentro do Github vai aparecer da seguinte maneira:

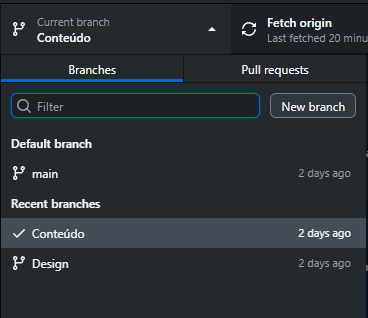


Isso porque a página do Github ainda não foi atualizada, mas assim que aparece todas as branches que foram criadas a partir do projeto.



Quando feita a devida atualização, passa a aparecer todas as branches que foram criadas do conteúdo

Depois de criadas devidamente as branches, elas vão aparecer da seguinte maneira:



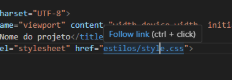
E ao invez de abrir cada branches de maneira individual para fazer alterações no código, basta  
escolher qual delas você quer fazer alterações, é bastante recomendado para deixar tudo devidamente separado quando for fazer alguma alteração dentro do código.

Um exemplo disso seria em um Site de jornalisto.

Enquanto uma equipe fica responsavel por fazer a parte de conteúdo, vai ter outras equipe responsavel por fazer a parte de Design. E quando todos terminarem suas devidas funções a gente faz uma Merge para adicionar tudo isso dentro da página principal (branche master).

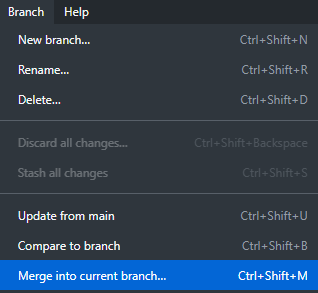
Ainda sobre essa aula, vamos pensar o seguinte. Imagina que você adiciona um link que está dentro de uma pasta. Porém a pasta ainda não existe, Segurando o CTRL + Clicando em cima do link.

Você pode solicitar que seja criada uma pasta que tem o nome que foi adicionado ao link, como pode ser visto a seguir

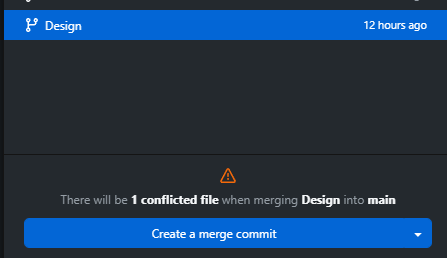


Existe uma ocasião que vamos descorrer que é o conflito de atualizações.

Porém por ora vamos ver como fazer uma ramificação “virar” a ramificação **master**. Para isso, depois de feita as atualizações e feito o commit das devidas branches. Para tornar uma das branches como a ramificação princial, isso pode ser feito atravez da barra de tarefas do proprio github desktop

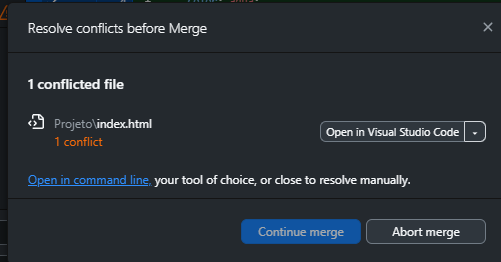


No cenario que a gente criou, tanto uma branche para **conteúdo** quanto uma branche para **Design,** então muito provavelmente vai dar conflito.



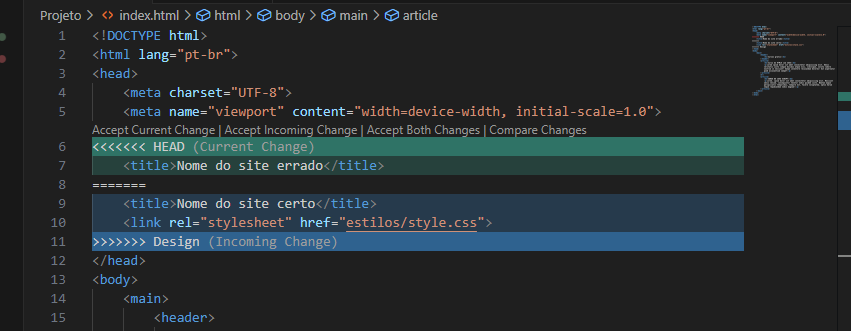
E por conta desse conflito que forçamos, não é possivel fazer o upload da branche, para a branche master.

E quando clicamos no **create a merge commit** o que vai aparecer vai ser essa tela:



Que informa que tem um conflito e em qual arquivo que está dando esse conflito.

Dentro do visual studio code vai aparecer o que está diferente em relação aos arquivos:



Neste exemplo que criamos o que está errado seria o titulo que não é conivente com o arquivo que já está no github.

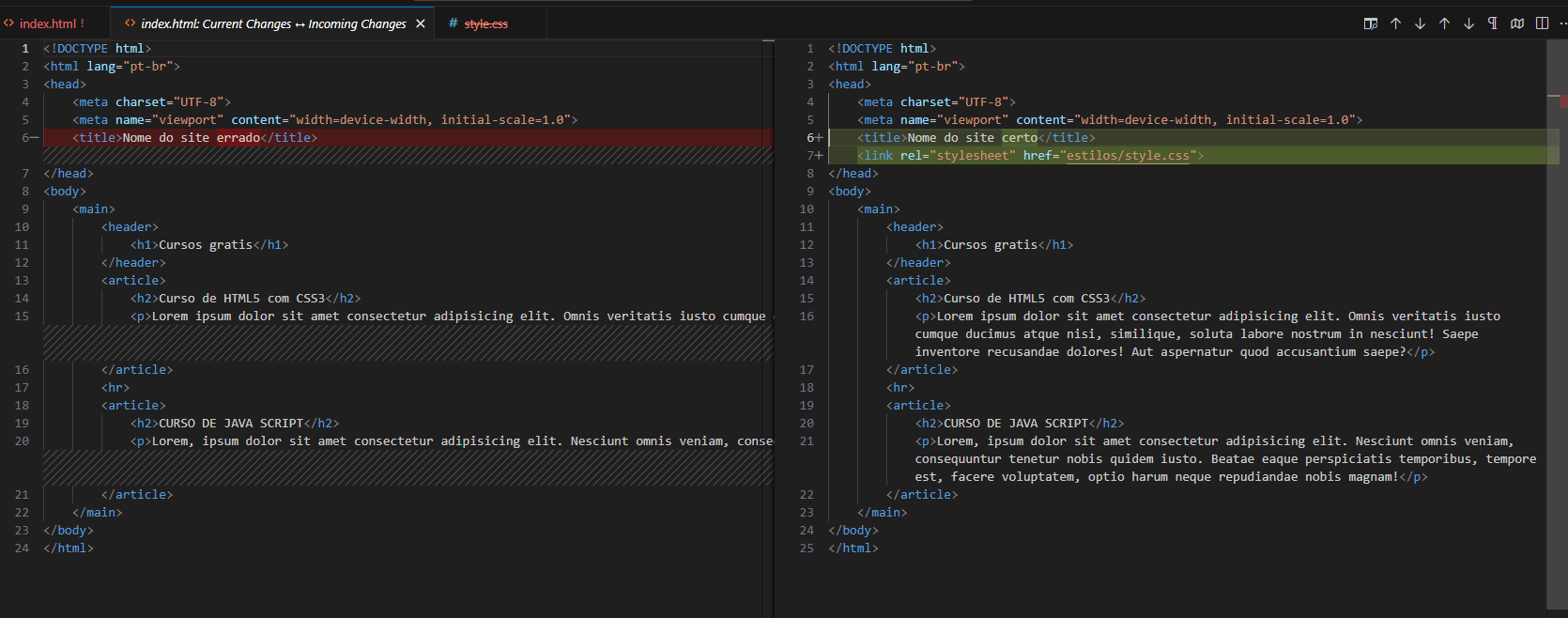
Sendo assim surgem algumas opções.

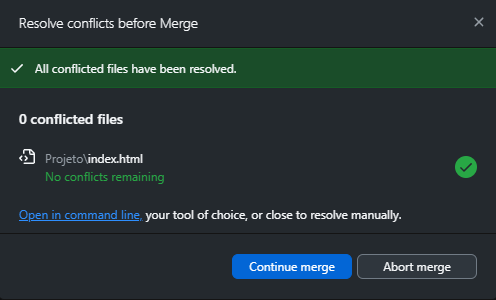
Accept Currrent Change: para aprovar a mudança da “segunda branche” em cima da primeira, reconhecendo que a segunda satisfaz mais ao desejado. (no exemplo em verde)

Acecept Incoming Change: aceitar manter do jeito que estava (no exemplo azul)

Accept Both Changes: aceitar as duas.

Compare Changes: comparer de maneira mais visível e intuitiva o que está causando o conflito.





Como não há mais conflitos, porque esse foram resolvidos, vai aparecer da seguinte maneira os arquivos.

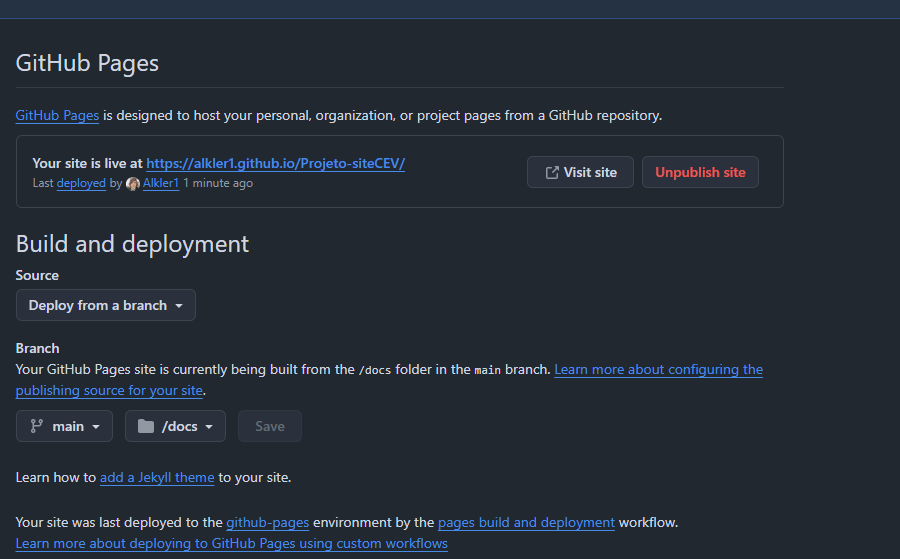
Quando você quiser selecionar mais de uma linha do Visual studio code para poder envelopar por exemplo, basta apertar a tecla Alt.

Tendo apertado a tecla Alt apertando Ctrl + Shift + P você pode envelopar o conteúdo que foi selecionado. Isso é, por exemplo no caso de fazer do conteúdo que esta selecionado, como parte de um código.

Hospedagem de sites no Github

Para fazer a hospedagem de um site através do Github Pages, basta acessar seu repositório, através do próprio site do Github, depois disso você procura pela opção **Settings** dentro do projeto que se deseja hospedar.





Basta escolher essas opções, em Branch, você escolhe qual dos itens do arquivo você quer tornar o site, ai ele gera um link.

Outro detalhe, quando é feito um branche para o site que está hospedado o github, demora de 30 segundos a 1 minuto para atualizar na página.

Outra coisa que é de se fazer, é deletar uma branche que não está mais sendo utilizando, se você utiliza uma branche como modelo para fazer uma atualização, mas depois não vai mais estar utilizando essa Branche, você pode deleta-la, pois ela cumpriu com a utilidade que tinha.

Outro detalhe O github pages, não funciona com coisas relacionadas a servidor, como MySQL, nodeJs. Somente coisas simples, como HTML, CSS, java Script, coisas da parte do site mesmo