

Git e github

Pedro Henrique Santos Araujo

Sumário

1. Git x github.....	2
2. O que é git e versionamento.....	2
3. Controle de versão sem os VCS'S.....	2
4. Versionamento centralizado ou linear.....	3
5. Versionamento distribuido.....	3
6. Softwares de versionamento além do git.....	4
7. Vantagens de uso do git.....	4
8. Aprofundando git x github.....	5
9. Github.....	5
10. Surgimento do git.....	6
11. Surgimento do github.....	6
12. Instalações necessárias.....	8
13. Criando repositório.....	8
14. Clonando repositório.....	11
15. Versionando projetos antigos.....	14
16. Issues e pull-requests.....	16
17. Linguagem Markdown.....	19
18. Branches.....	26

Git x github

Para dar início ao estudo desses dois termos, devemos comentar sobre duas definições mais gerais, mas normalmente errôneas em alguns sentidos, além disso, já podemos definir aqui que **git e github são ferramentas diferentes**. Enquanto o git é um software de controle de versão (VCS), o github é uma rede social para programadores, no entanto, como foi dito, essas definições são muito superficiais.

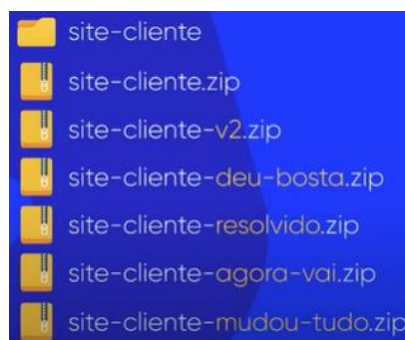
Git e versionamento

O git falando de modo aprofundado e certo, **é um software de controle de versão (VCS)**, ou seja, é uma **ferramenta de versionamento de software**. Por conseguinte, para entendermos de modo mais claro é necessário definir **versionamento**, esse termo se dirige as **versões** desenvolvidas de um software até sua versão “final”, esses VCS’s são de suma importância porque faz parte do **cotidiano dos programadores de lidar com diversas versões de um mesmo projeto**. Por fim, o git guarda e gerencia as versões de um mesmo projeto.

Controle de versão sem VSC

Para exemplificar e deixar ainda mais claro para que serve os softwares de versionamento, vamos seguir a seguinte linha de raciocínio:

1. Normalmente, sem o uso dos VCS’s cada versão de um site seria separada e organizada em pastas distintas, quando uma versão é finalizada a pasta é zipada e armazenada;
2. Para fins de segurança, estes arquivos zipados devem ser armazenados em backups seja na nuvem ou em dispositivos físicos, ambos com risco de perda de dados;



3. A complexidade desse tipo de controle se dá de forma simples de se entender, imagine um desenvolvimento de site, **cada setor é desenvolvido por um programador diferente**, com isso, cada um teria um **arquivo zipado e armazenado no google drive a cada versão feita**, isso geraria uma **confusão e**

desorganização, por isso, os softwares de versionamento são importantes para gerenciar cada modificação e nova versão feita;

Por fim, o versionamento é como se fosse uma máquina do tempo, para **voltar em qualquer ponto do desenvolvimento de um projeto**, o git, apesar de ser o mais famoso, não foi o primeiro e nem o único software de versionamento que não gera um zip a cada modificação, ele guarda cada modificação feita (linha por linha).

Versionamento centralizado ou linear

O funcionamento dos softwares de versionamento **centralizado ou linear**:

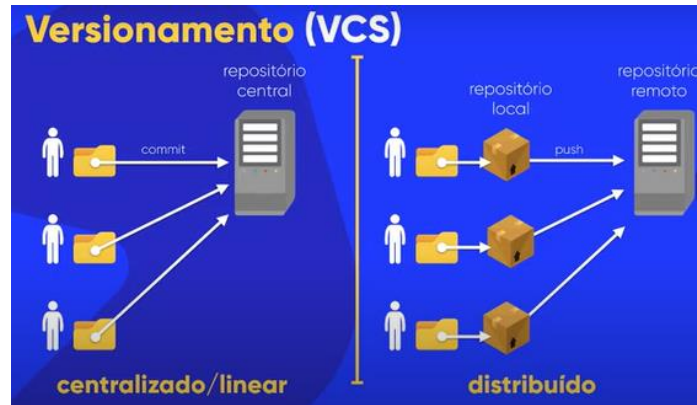
1. Inicialmente, o projeto está salvo em uma pasta com os arquivos organizados;
2. Depois de ter sua “primeira” versão concluída o software de versionamento não faz um arquivo zip como foi explicado anteriormente e sim um **commit** para ser mandado para um **repositório central**;
 - **Commit é o processo de tornar permanente um conjunto de alterações**, ou seja, ele efetiva alterações. Em outras palavras nós fazemos um commit quando pegamos uma pasta, ela é monitorada e mandada para o repositório central;
 - O **repositório central** é um servidor que armazena todos os arquivos de todas as versões referentes a um projeto;
3. Os primeiro VCS's tinham uma limitação, para dar um commit era necessária a **conexão constante** com o repositório central;
4. **Várias pessoas podem dar commits** em um mesmo servidor;
5. A organização dos arquivos é feita pelo **servidor**, ele se vira e **descobre qual versão veio primeiro, qual veio depois e quais são suas diferenças**;
6. Esse tipo de versionamento explicado acima é chamado centralizado ou linear;

Versionamento distribuído

O funcionamento dos softwares de versionamento **distribuído**:

1. Nesse modelo mais moderno, **não é necessário ter a conexão direta e constante** com o servidor, pois aqui não acontece o **commit para o repositório central e sim para os repositórios locais instalados nas nossas máquinas**, sendo não necessário o poder de processamento de um servidor;
2. **O git trabalha com o versionamento distribuído**;
3. No entanto, como não trabalhamos com o repositório central, temos um novo conceito que deve ser inserido. Para que várias pessoas trabalhem se comunicando pelos **repositórios locais**, ou seja, para que um repositório x trocando informações com outro y é necessário o **repositório remoto**;
 - Repositório remoto é como se fosse um google drive que **armazena as versões de um projeto**;

4. O versionamento acontece do seguinte modo: quando esses repositórios locais se encontram com um **conjunto grande de modificações para declarar uma nova versão** é feito um **commit** para os repositórios locais, a partir desse **commit** fazemos um **push** para o repositório remoto;
- Push é a **ação de pegar o versionamento atual no repositório local e jogar no repositório remoto**;



Softwares de versionamento além do git

Os mais famosos softwares de versionamento foram (os escritos em amarelos foram os mais importantes e vamos comentar muito em nossos estudos):



Vantagens de uso do git

Por fim, é importante destacar as vantagens de se trabalhar com git ou com qualquer outro software de versionamento de códigos:

1. **Controle de histórico:** permite ver cada alteração feita, em qual dia foi feita e por quem foi feita, além de ter acesso a todas as versões de desenvolvimento;
2. **Trabalho em equipe e ramificação do projeto:** permite a divisão do desenvolvimento em setores como design, banco de dados e backend, além de,

- posteriormente ocorrer a junção de todos esses setores formulando o projeto como um todo;
3. **Segurança:** a equipe ou integrante interage apenas com seu setor e só faz alterações e tem acesso aos seus próprios códigos, sem a interferência de outro setor;
 4. **Organização:** não existe aquela quantidade massiva de pastas compactadas, não há a perda de mídias, pois se você perder algo basta voltar em uma versão anterior;

Aprofundando git x github

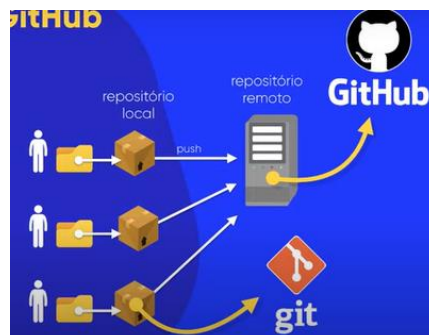
Como vimos anteriormente, o **git** funciona como versionamento distribuído, ou seja, com os conceitos de **repositórios locais e remotos** e, como vimos, o git não gera um arquivo zipado a cada nova versão, ele apenas **salva as novas alterações feitas**. Para exemplificar, imagine que inicialmente você tem 200 linhas de código, em sua segunda versão você tem 10 linhas de código adicionais, a função do git é apenas guardar essas 10 linhas e não copiar uma segunda vez as 200 linhas da primeira versão.

O **git** atua no **repositório local**, enquanto o **github** não só atua no **repositório remoto**, ele é um **repositório remoto** e é o mais famoso deles, mas não o único. Em outras palavras, o **github** é um **local para armazenar os arquivos referentes aos códigos de qualquer linguagem**.

GitHub

Além da ferramenta ser um repositório remoto para armazenar seus códigos, ele também oferece muito mais usos:

- Repositórios ilimitados;
- Hospedagem de códigos em qualquer linguagem;
- Funciona como uma rede social para programadores (comunidade);
- Serviço de hospedagem simples para sites (github pages integrados);
- Colaboração com outros projetos (forks);
- Teste de códigos de outros projetos;
- Contato com vagas de emprego na área;



Cabe ressaltar que existem outros repositórios famosos:



Surgimento do Git

Vamos seguir a linha cronológica em relação aos mais famosos acontecimentos em relação aos softwares de versionamento. **Em 1985 surgiu o CVS** (Concurrent Version System), VCS centralizado, open source (código fonte disponibilizado para manuseio) e o mais popular da época, no entanto tinha muitos problemas de consistência, velocidade e não permitia renomeação e realocação de pastas.

Em 2000 surgiu o SVN, centralizado, open source, muito parecido com o CVS até porque a ideia do SVN era se parecer ao máximo com o antecessor, mas com os problemas resolvidos, ainda é ativo até os dias atuais, é da fundação apache e, apesar de resolver alguns problemas do CVS ainda tinha muitos problemas.

Ainda em 2000 surge o BitKeeper, VCS distribuído, **proprietário (não é open source)**, tinha uma **versão comunidade** com muitas limitações, era completamente diferente com os dois anteriores e um de seus maiores clientes foi o **Linux**.



A partir da criação do BitKeeper, os acontecimentos que a sucederam foram todos responsáveis pela criação do Git.

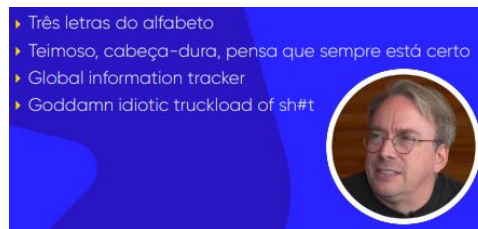
Em 2004, ocorreu um conflito em relação ao fato de o **Linux** ter a característica **open source**, mas trabalharem com um software de versionamento de código proprietário. Toda confusão começou quando **Andrel Tridgell** fez uma espécie de engenharia reversa para **destravar recursos que eram travados na versão comunidade do BitKeeper**.

Em 2005, com o fato que ocorreu antes, o BitKeeper adquiriu uma **nova licença que restringia ainda mais os conteúdos**, bloqueios como acesso a metadados e a versões anteriores foram impostos. Com isso, o **representante da Linux** e mais tarde criado do Git **Linus Torvalds** ficou irritado com a situação, pois ele mesmo fazia o uso da ferramenta para o desenvolvimento dos códigos Linux

Depois de toda essa confusão, em 2005 Linus Torvalds criou o mais famoso software de versionamento de software da atualidade, o Git. O mesmo, como foi dito, é distribuído, open source, foi feito em 10 dias e focado totalmente em performance.



Por fim, Linus já foi questionado diversas vezes sobre o significado de git e suas respostas foram diversas e peculiares, veja:



Surgimento do Github

O Github veio depois do Git, foi criado em 2008 por quatro amigos, é um software proprietário, surgiu apenas com a característica de hospedagem de códigos e foi baseado em Git.

A ferramenta teve um crescimento absurdo, veja no quadro abaixo:



- Em 2011 o Github ultrapassou o SourceForge que era o mais famoso até então em questão de armazenamento de códigos.
- Em 2016 ficou em 14º na lista Forbes Cloud Hundred ("Tecnologias de nuvem para ficar de olho").
- Em 2018 sofreu o maior ataque de DDoS da história, que, em poucas palavras, é um ataque virtual cujo objetivo é tirar o serviço prestado pelo Github.

Ainda em 2018 o Github foi comprado pela gigante Microsoft, a qual, ela mesma armazena projetos próprios no Github, inclusive um dos editores de códigos mais famosos do mundo, o **Visual Studio Code**. No entanto, apesar da compra, a **operação da ferramenta ainda se manteve do mesmo modo feito antes de ser compra**.

Em 2020, o próprio Github comprou o NPM (Node Package Manager) que é um gestor de pacote feito em JavaScript utilizando o **Node JS (é um software de código aberto do google que permite execução de códigos em JS fora de um navegador web)**.

Por fim, a Microsoft está dando muita atenção para os softwares de códigos livres, a mesma é **dona** do maior hospedeiro de códigos do mundo (**Github**), da maior base de dados de **programadores** do mundo e dona do maior gestor de pacotes de JS do mundo (**NodeJS**).



Instalações necessárias

Para continuar com os estudos de Git (VCS – Software de controle de versão) e Github (plataforma de hospedagem de códigos), são necessárias algumas instalações e configurações, todas as informações necessárias estão no vídeo de número 4, sendo elas: o navegador Google chrome ou Microsoft edge, o Git, o Github desktop (usado para não precisar digitar comandos) e editor de códigos Visual Studio Code.

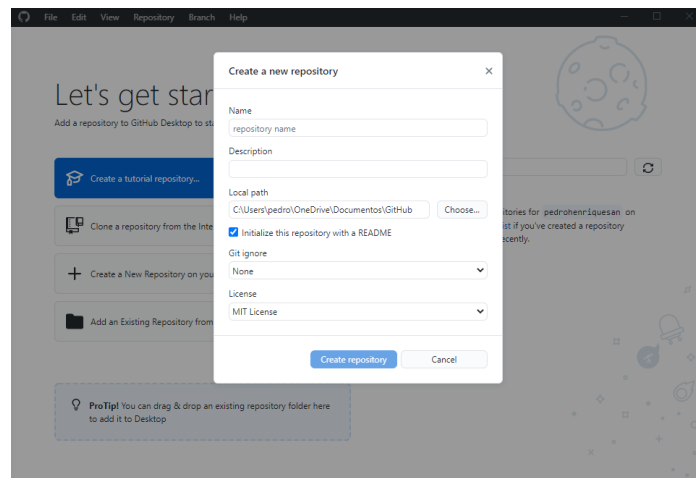
GitHubDesktopSetup-x64	06/11/2021 10:04
Git-2.33.1-64-bit	06/11/2021 10:02
Anteriormente nesta semana (2)	
VSCodeUserSetup-x64-1.62.0	04/11/2021 18:10
ChromeSetup	31/10/2021 12:54

Criando o primeiro repositório

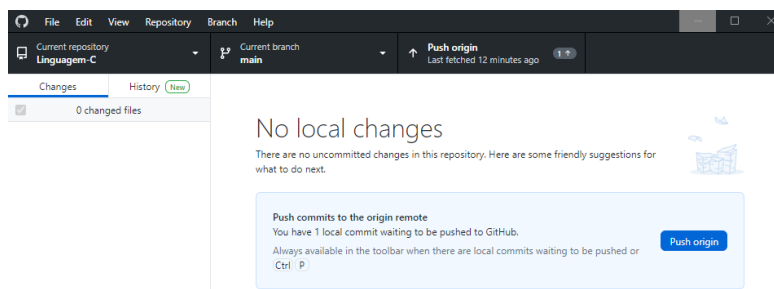
Primeiramente, é importante lembrarmos que o repositório local se encontra na nossa máquina (Git) e o repositório remoto se encontra no servidor (Github), além disso, lembre-se que quando é feita uma nova versão/alteração nós fazemos um **commit para o repositório local e depois um push para o repositório remoto**.

Em primeiro lugar, no nosso curso vamos trabalhar com a **ferramenta github desktop**, a qual facilita a criação e manipulação dos repositórios e **evita o uso do terminal** que é mais complexo.

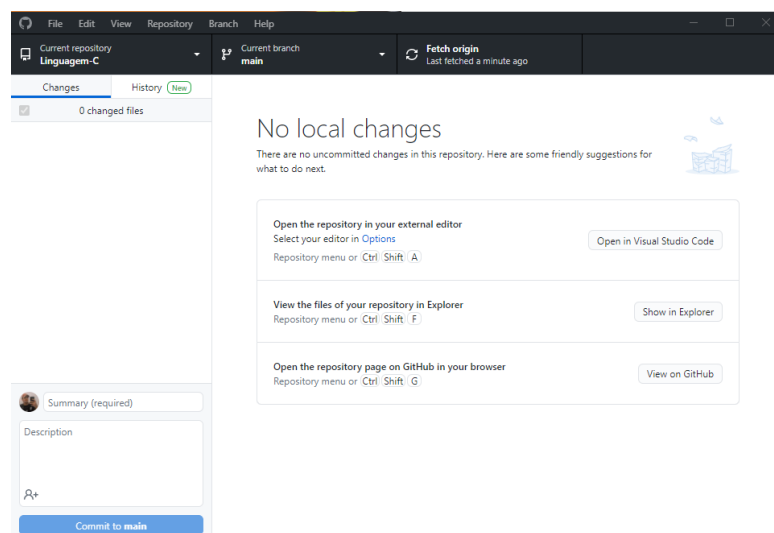
O primeiro passo é abrir o Github desktop instalado na última aula e escolher a opção de criar um novo repositório:



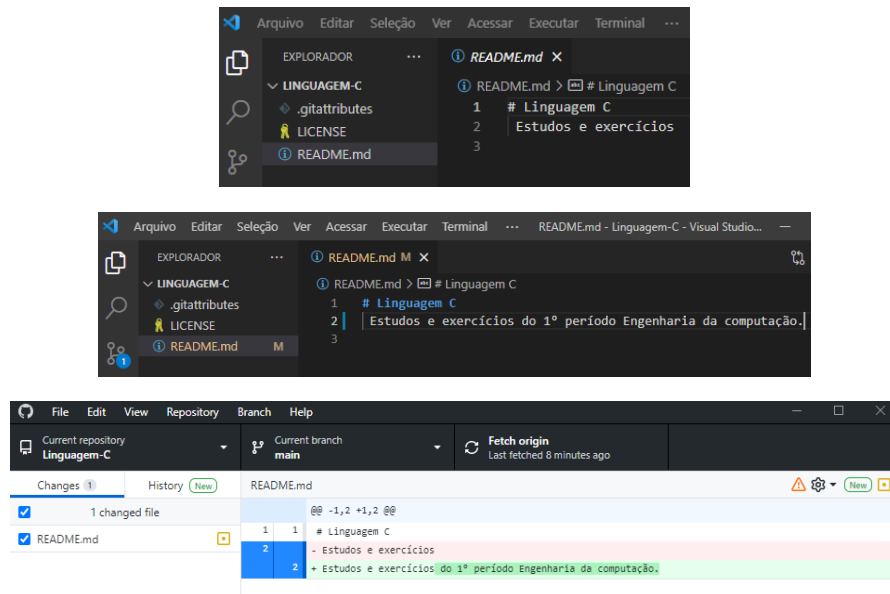
Quando selecionamos a opção e preenchemos todas as configurações e criamos o repositório não aparece nada em nossa página no github, pois como foi dito antes primeiro criamos o repositório local (Git) e depois fazemos o push para o Github. Após isso, **para a criação esteja visível em seu perfil é feito o push que o próprio Github desktop oferece:**



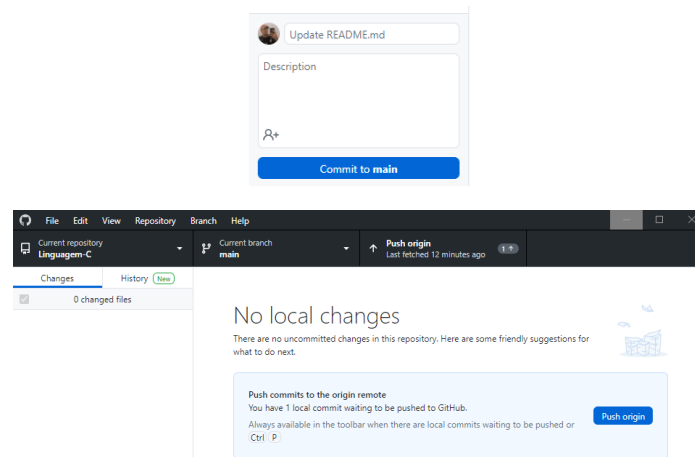
Com isso, o seu repositório está de fato criado no seu perfil do Github, agora vamos nos inserir na parte das alterações do código fonte/novas versões, na página do Github desktop tem a opção de abrir seu repositório com o VSCode, com ele aberto é possível editar qualquer parte, aqui vamos **exemplificar a modificação do “README”**:



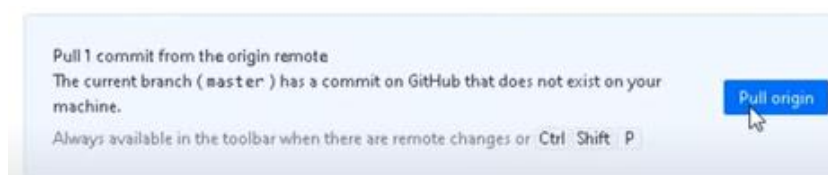
Quando o VS é aberto e alguma informação é alterada, tanto no VS como no desktop aparece uma notificação de que uma alteração foi feita, mas ainda não foi extraída, ou seja, **não foi feito um commit, a alteração foi feita, mas não foi versionada.**

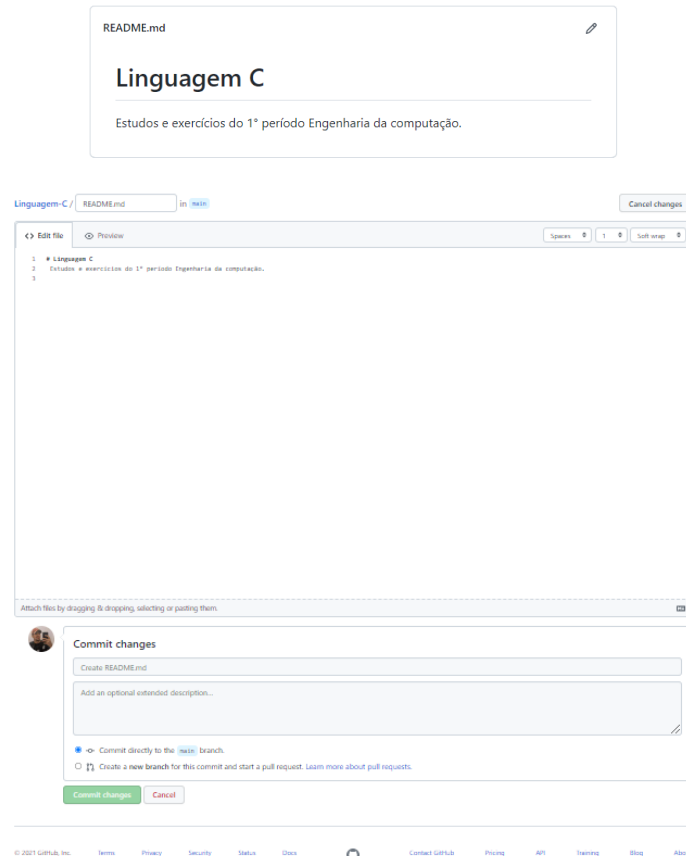


Para que essa alteração seja efetivada, **tem uma área na parte inferior esquerda do Github desktop referente aos commits**, lá você dá um título e uma descrição para esse commit de acordo com a modificação que foi feita, após isso, é necessário fazer o push para o repositório remoto que a própria ferramenta te oferece.



Por fim, até mesmo no Github possui um simples editor de código para alterações de emergência que não é recomendado de ser usado, mas é importante lembrar da existência do mesmo. Além disso, quando nós ou outras pessoas alteram o código fonte pelo editor do Github nós fazemos um **“pull” para trazer essas modificações para nossa máquina.**





Em resumo:

1. Primeiro crie o repositório pelo Github desktop;
2. Abre o mesmo com o VSCode pelo Github desktop e faça as alterações/criações no mesmo, pastas, arquivos e entre outros;
3. Faça o commit para o repositório local (Git);
4. Faça o push para o repositório remoto (Github);

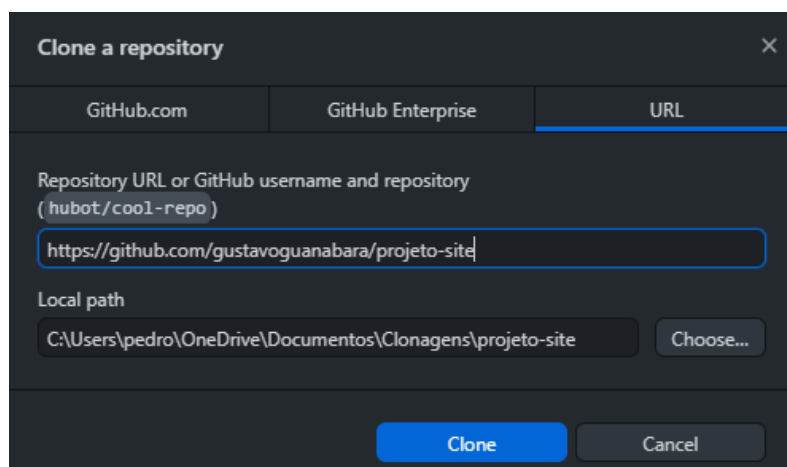
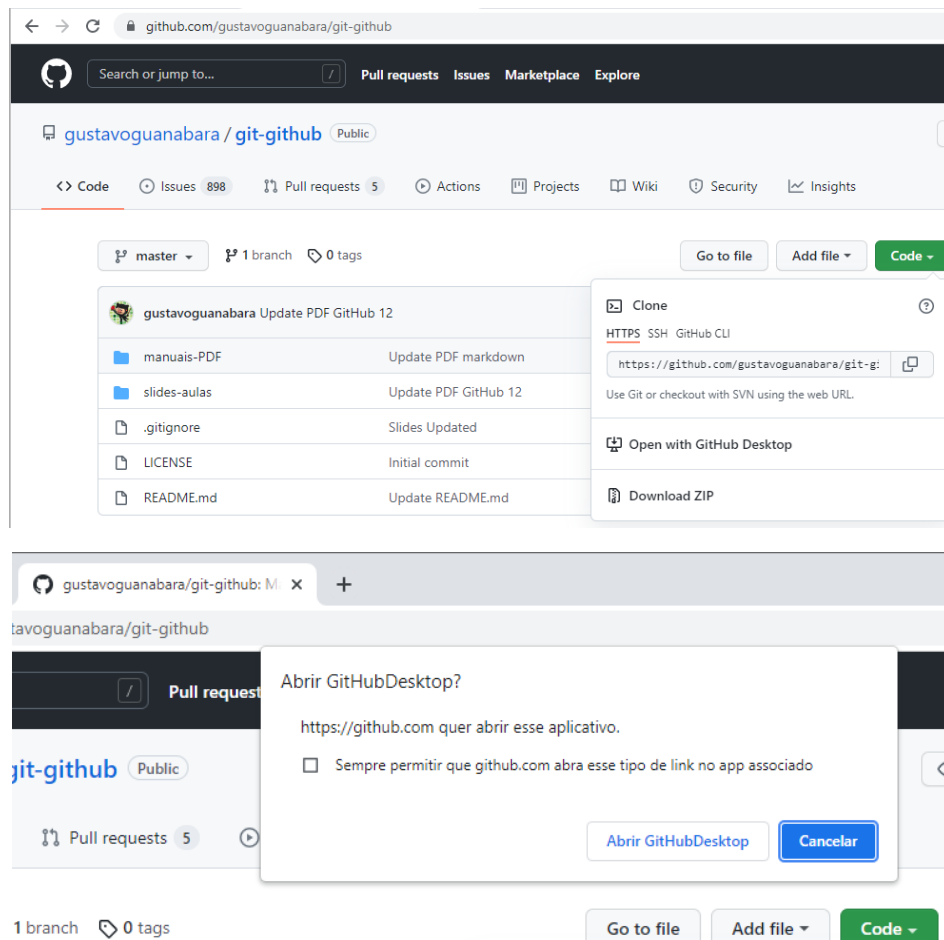
Clonando repositórios

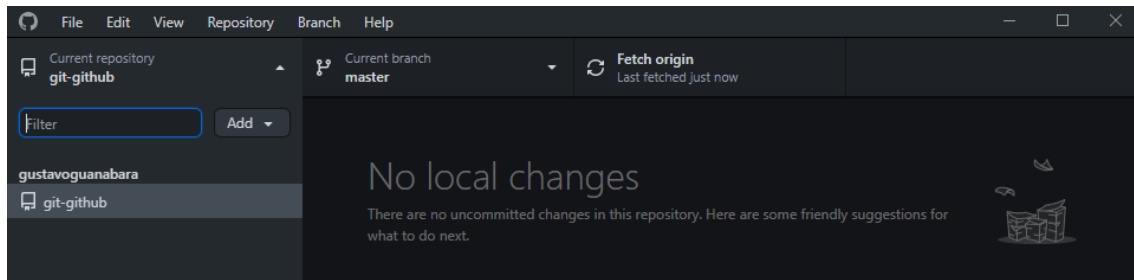
Antes de partirmos para o assunto principal do tópico, é considerada uma **boa prática**, sempre que formos começar um novo dia de trabalho em algum projeto, selecionarmos a opção “**Fetch Origin**” no Github desktop, essa função compara os códigos versionados com os presentes em nossas máquinas para **verificar se tem algum pull, push ou commit para fazer**.

Clonar um repositório público é a ação de pegar todos os arquivos referentes a esse repositório e fazer com que o git faça o versionamento do mesmo em nossa máquina para uso próprio (conceito de **software livre**). A princípio isso pode parecer uma coisa errada, mas até os mais **famosos softwares disponibilizam seus códigos fontes no próprio Github, como o VScode, Linux e entre outros**.

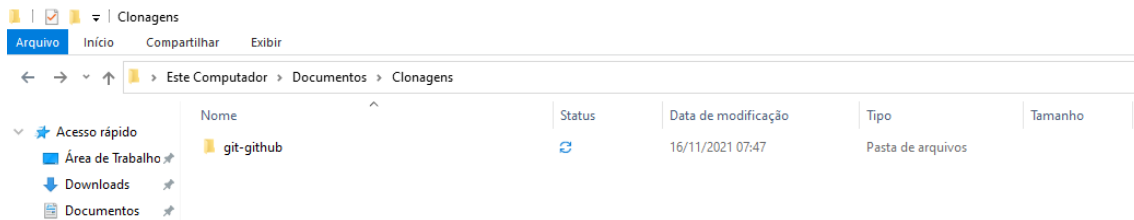
Passo a passo para clonar um repositório:

1. Escolha um repositório que seja público;
2. Selecione a opção “code”;
3. Selecione a opção “open in desktop”;
4. Autorize o navegador;
5. Depois de abrir no Github desktop altere as configurações se necessário e selecione clonar;
6. Por fim, quando clonado, o repositório fica salvo com o indicativo de quem ele é;

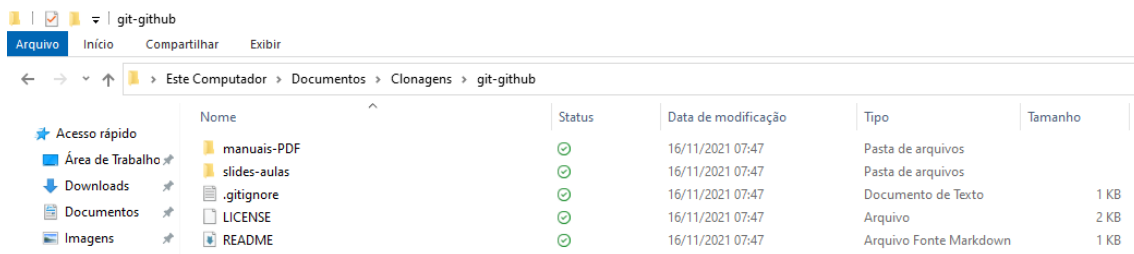




Repositório salvo no Github desktop com o nome do proprietário



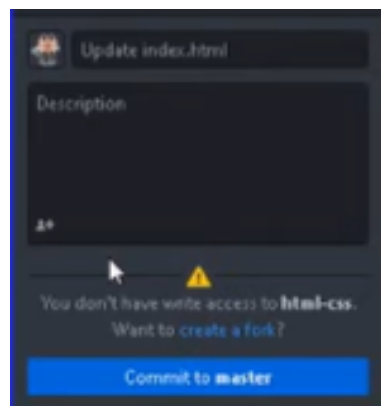
Pasta do repositório no local que marcamos ao clonar



Todos os arquivos do repositório

Por fim, do mesmo modo que podemos abrir nossos repositórios na web, no explorador de arquivos e no VSCode podemos fazer a mesma coisa com os clonados.

No entanto, quando alteramos os códigos de um repositório clonado **não é possível, obviamente, fazermos um commit/push dos branches executados (ramificações/alterações) que vimos anteriormente**, pois os códigos não pertencem ao nosso repositório, ao tentarmos fazer um commit por exemplo aparece um alerta, veja:



Para fazermos isso, devemos criar um outro projeto próprio copiando o código. Mas, no primeiro caso, podemos **sugerir ou requisitar alterações** para o dono do repositório, por meio de **issues, forks e branchs**.

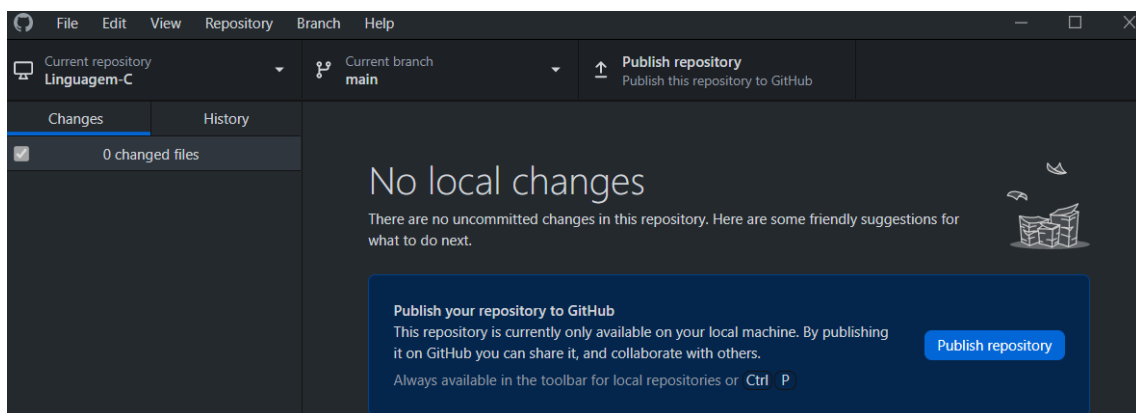
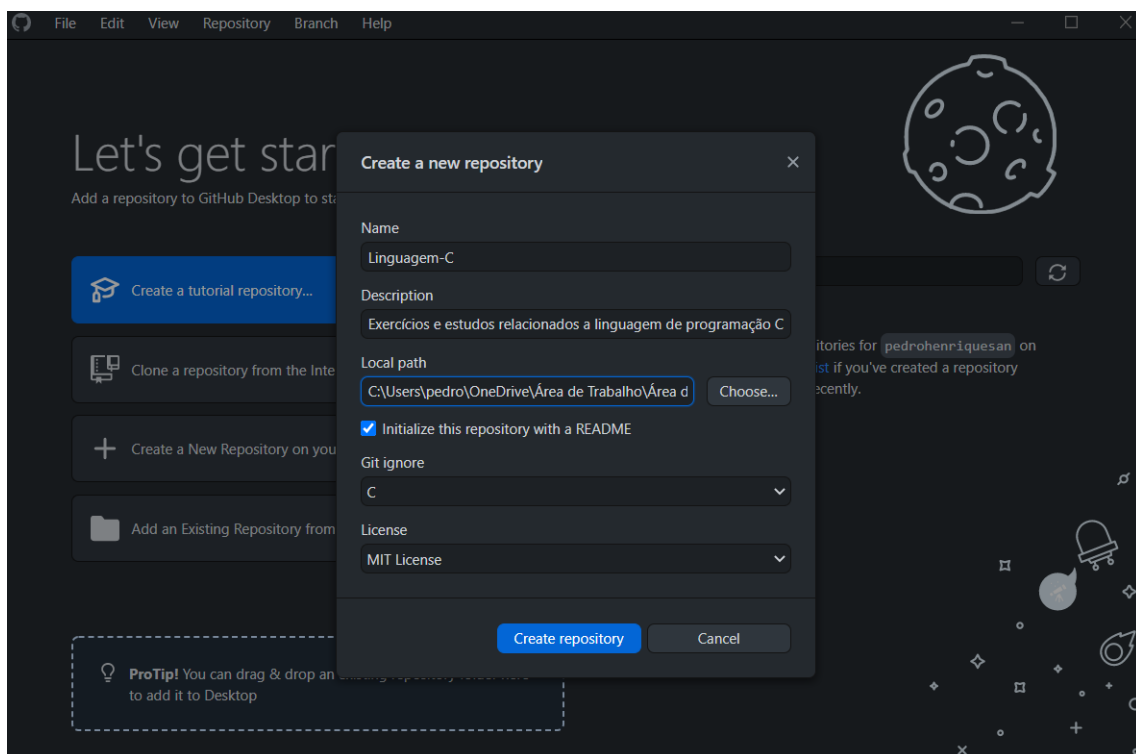
Para darmos fim a este tópico, existe um modo de saber se seu próprio repositório ou algum clonado está ou não versionado, veja:

- Abra a pasta do repositório;
- Vá na aba Exibir e selecione itens ocultos;

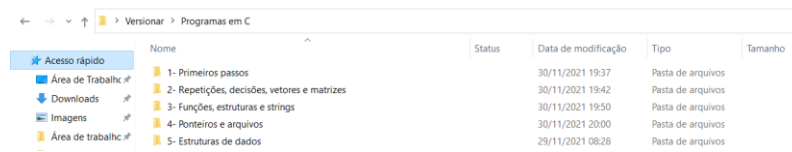
Se a pasta `.git` existir significa que o repositório está versionado e o conteúdo dessa pasta são as versões anteriores juntos com seus arquivos.

Versionando projetos antigos

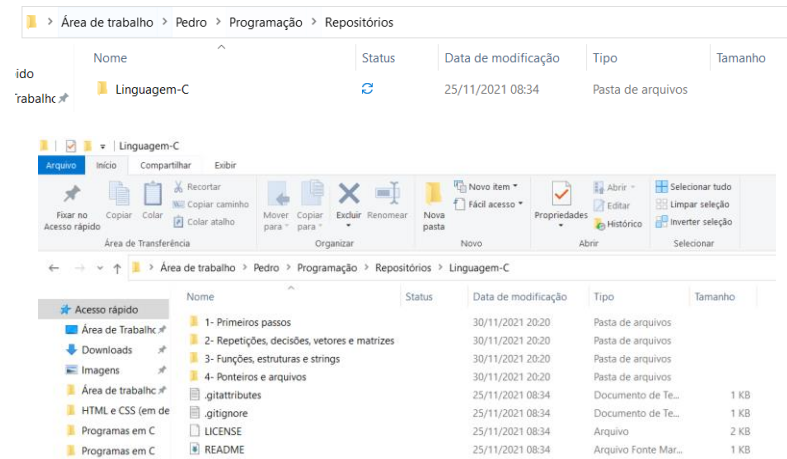
- Cria-se o repositório



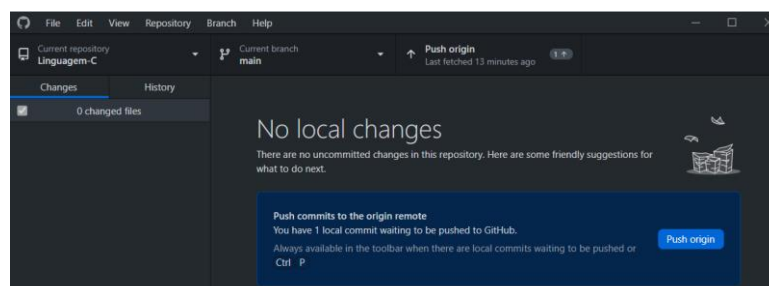
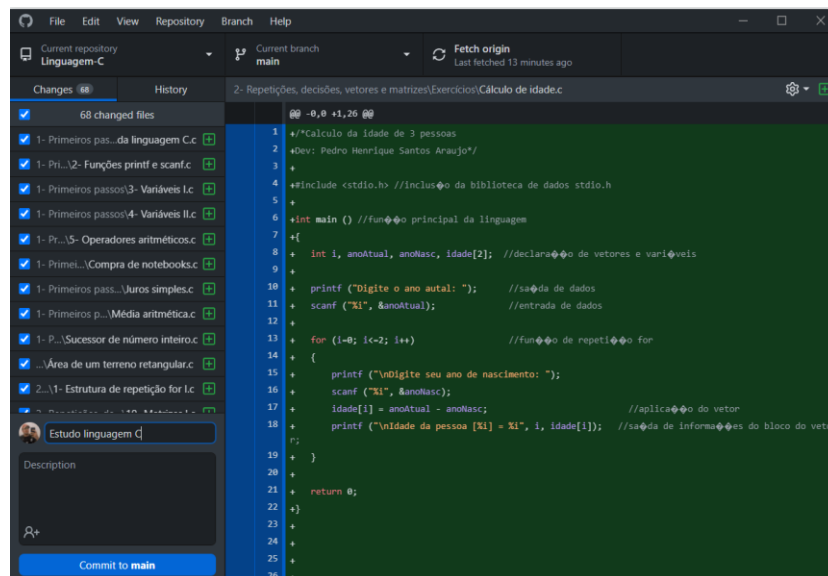
- Organize seus projetos da maneira que deseja



- Transfira esses arquivos para a pasta do repositório criado



- Pelo Github desktop faça o commit e depois o push



Issues e pull-requests

Introdução

Agora vamos iniciar o estudo sobre duas ferramentas do github muito usadas e frutíferas em nossas vidas profissionais, são elas: **as issues e os pull-requests**. Para entendermos essas duas ferramentas, imaginemos que entramos no repositório da Microsoft referente ao projeto do Visual Studio Code, com ele **aberto achamos um erro no código, com esse conflito temos dois caminhos para tomar, fazer uma issue ou um pull-request**.

Em resumo, as **issues** são problemas que você se deparou e **não sabe como resolver**, enquanto os **pull-requests** são problemas que você se deparou que você **sabe resolver**.

Issues

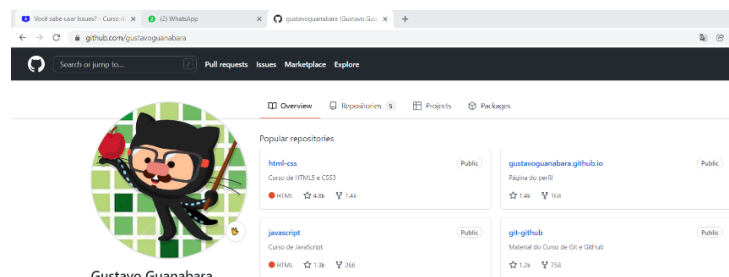
As issues podem ser interpretadas como problemas, questões ou levantamentos de algo, como um problema de funcionamento que foi citado na introdução. As issues quase sempre são usadas por iniciantes que não sabem resolver todo problema que encontra na carreira, com isso quando você se depara com um problema você compartilha com a comunidade do Github através de uma Issue.

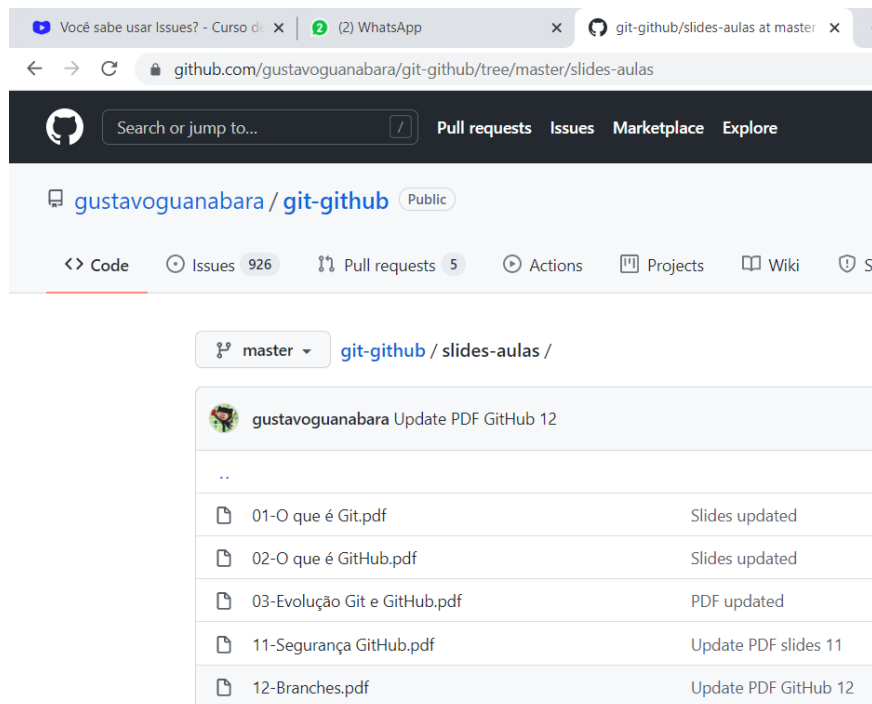
Veja um passo a passo:

- Acessando o github da Microsoft;
- Entrando no repositório do VSCode e pesquisando por issues (problemas identificados pela comunidade), issues open ainda não foram resolvidas, enquanto issues closed foram resolvidas;
- Em resumo, as issues são problemas encontrados em algum projeto, o qual ainda não possui uma solução, ou seja, quando você se depara com um problema, o primeiro passo é **verificar se já existe um issue já aberta** e que possa te ajudar, ao invés de já ir criando outra issue igual, **esse hábito evita issues duplicadas** (que não são bem vistas na comunidade)
- **Nunca cria uma issue antes de verificar se não tem uma igual ou análoga já aberta ou fechada.**

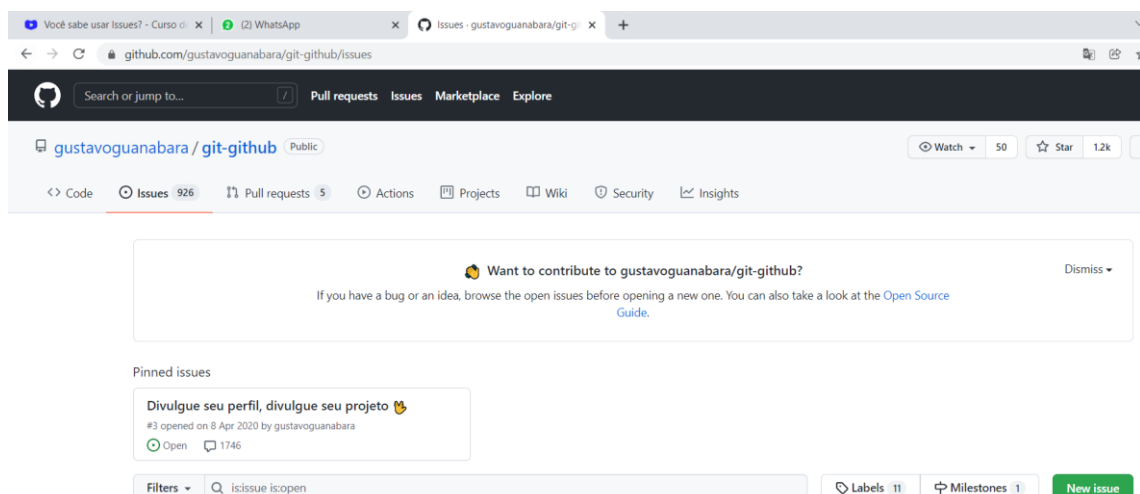
Exemplo de criação de Issue

Abrindo o perfil x do github e acessando o repositório y, o qual contém o projeto com o erro encontrado:

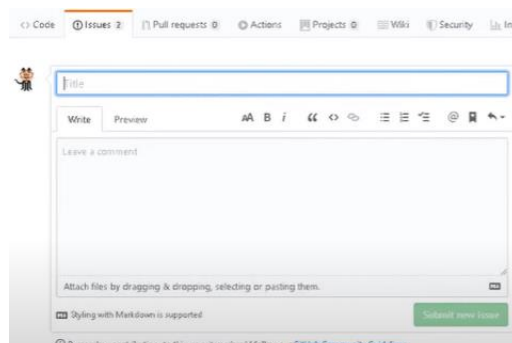




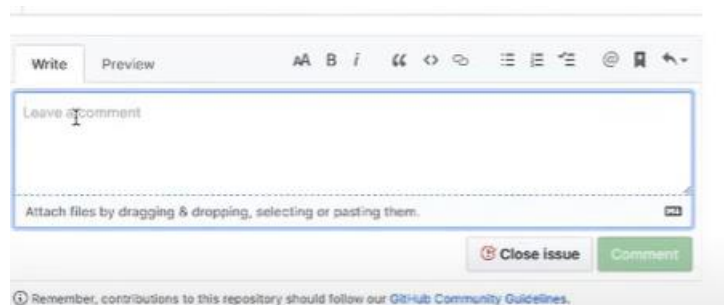
O problema em questão é: Não consigo encontrar o slide da aula 4 no atual repositório. Esse seria seu problema, sua issue, para compartilhar sua dúvida com a comunidade usa-se as issues:



Depois de verificar se não há uma issue aberta/fechada com a mesma dúvida que a sua, clica-se em “new issue”:

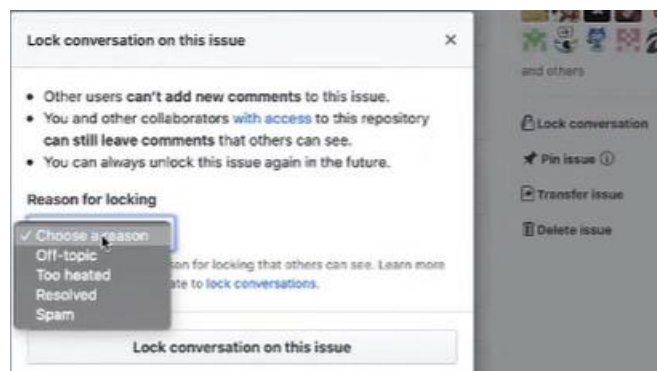


Com isso, você preenche os campos requeridos com todos os detalhes do erro encontrado ou dúvida pessoal referente ao projeto em questão:



Por fim, com a issue publicada, **qualquer pessoa** além de poder **publicar issues podem ver a sua e ajudar com a dúvida um dos outros**. Ademais, é importante dizer que **nunca se deve falar de um assunto diferente ao da issue**.

Agora, imagine que você seja o dono do projeto que foram feitas diversas issues, quando você tiver uma resposta para a mesma, basta responde-la e depois fecha-la, com isso, para bloquear os comentários da mesma depois de resolvida basta seguir o passo a passo abaixo:



Pull-requests

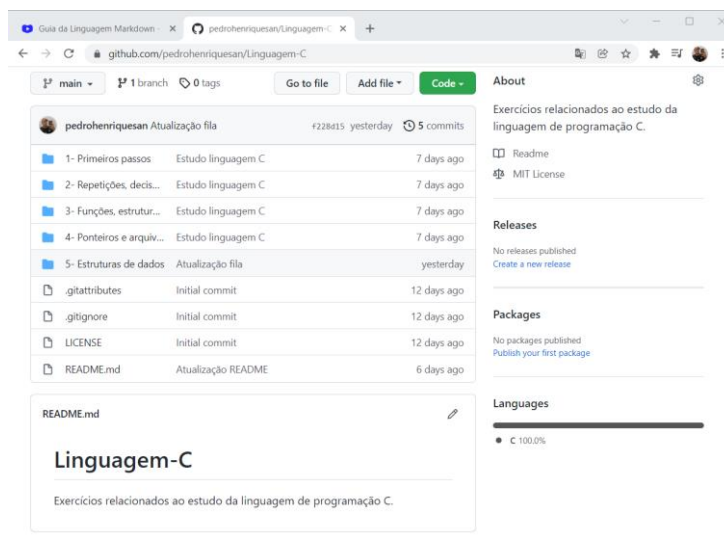
Sobre o segundo caminho a se tomar quando se depara com um problema, imagine que você já tem uma boa experiência e acha que pode resolver o problema encontrado ao invés de criar um Issue. Neste cenário você vai fazer um pull-request, para isso, você deve **clonar o repositório para sua máquina, analisar o erro, criar um fork, depois um branch (ramificação) do projeto, resolver o problema e finalmente fazer um pull-request**.

Com isso, se o proprietário do projeto ver sua solução como viável, ele irá te declarar como **colaborador do projeto**, essa é uma das grandes vantagens da **característica de rede social do github**, os **pull-requests adicionam muito na carreira** quando feitos de forma séria e responsável.

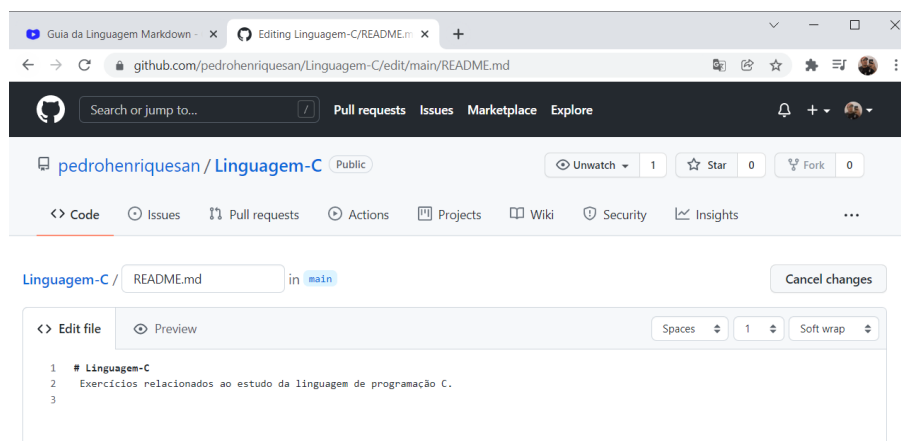
Linguagem markdown

Assim como a HTML, a linguagem **markdown não é uma linguagem de programação, e sim uma linguagem de marcação**, criada por John Gruber e Aaron Swartz com o intuito de facilitar a criação de conteúdo formatado e facilitar a transcrição para HTML. Além disso, eles afirmavam que o código em HTML era poluído (baixa readability) e criaram a md para ser mais limpa e fácil de ser lida.

Markdown basicamente é uma **linguagem de marcação que pode ser considerada como um HTML simplificado, que usa símbolos simples digitados pelo teclado que geram resultados visuais**, muitas tecnologias são **compatíveis** com md, inclusive o **Github**, como exemplo temos o arquivo README.md que é escrito em markdown (md).

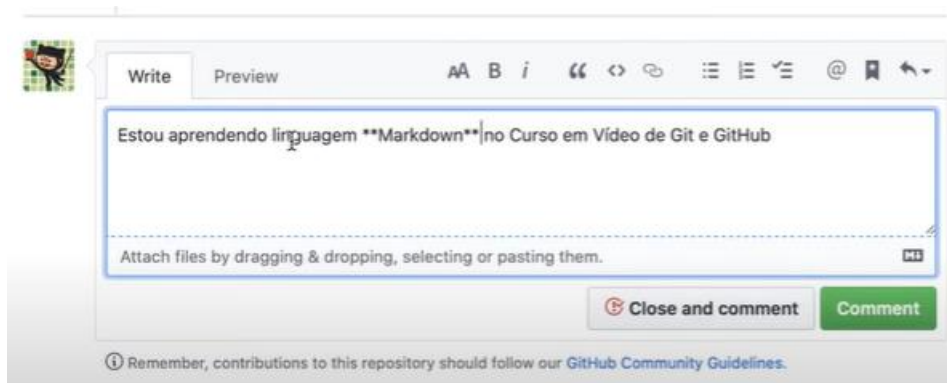


Veja um repositório em aberto, na parte inferior temos o arquivo README.md, cliando no lápis na parte superior para editar o arquivo abre-se o seguinte código que markdown:



Nesse editor de código já nos deparamos com um código incrivelmente mais fácil de ser lido em comparação com HTML e, já podemos aplicar as marcas da linguagem markdown para alterarmos nosso **README.md**. Além desse arquivo podemos usar a **linguagem nas Issues e nos pull-requests**.

Para nos inserirmos nas marcas da linguagem markdown vamos usar as Issues como exemplo de uso.



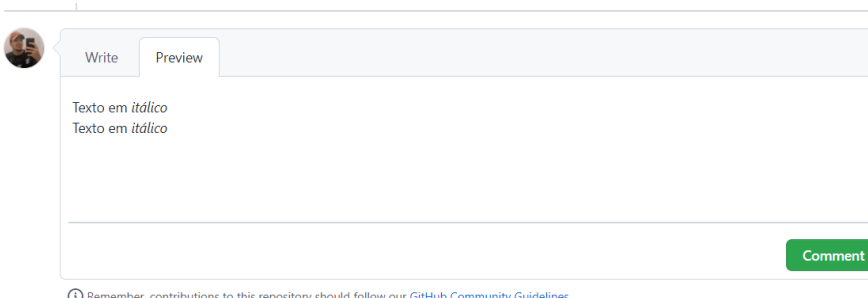
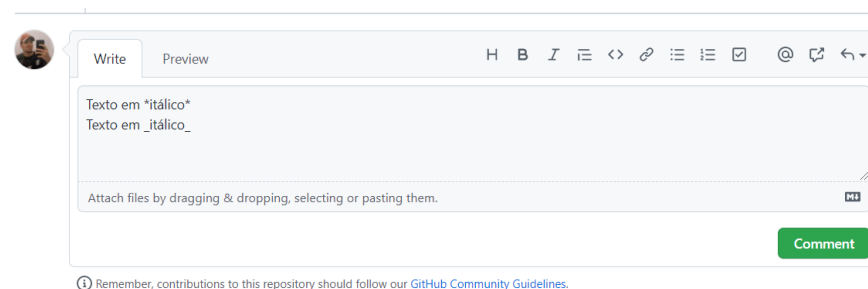
Aqui (Write) é digitado o código.



Aqui (Preview) é visto como está ficando os resultados visuais.

Vamos as marcas da linguagem:

- **Itálico:** a marca para colocar algum texto em itálico é (*texto* ou _texto_) sem espaços. Veja:



- **Negrito:** a marca para colocar algum texto em negrito é (****texto**** ou **__texto__**) sem espaços. Veja:

The first screenshot shows the 'Write' tab of the GitHub comment editor. The text area contains two lines: 'Texto em **negrito**' and 'Texto em __negrito__'. The second screenshot shows the 'Preview' tab, where the text is rendered as 'Texto em **negrito**' and 'Texto em **negrito**'. Both screenshots include a 'Comment' button and a reminder to follow the GitHub Community Guidelines.

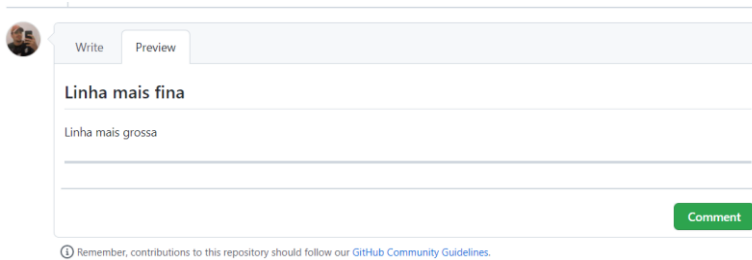
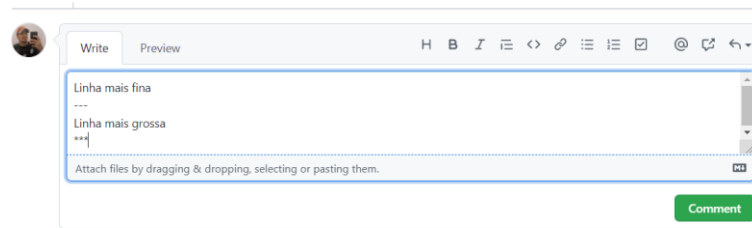
- **Strikes:** a marca para riscarmos palavras ao meio é (**~~texto~~**) sem espaços. Veja:

The first screenshot shows the 'Write' tab of the GitHub comment editor. The text area contains the line 'Texto ~~riscado~~'. The second screenshot shows the 'Preview' tab, where the text is rendered as 'Texto ~~riscado~~'. Both screenshots include a 'Comment' button and a reminder to follow the GitHub Community Guidelines.

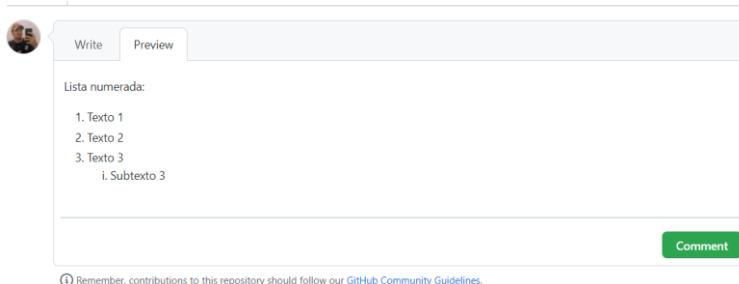
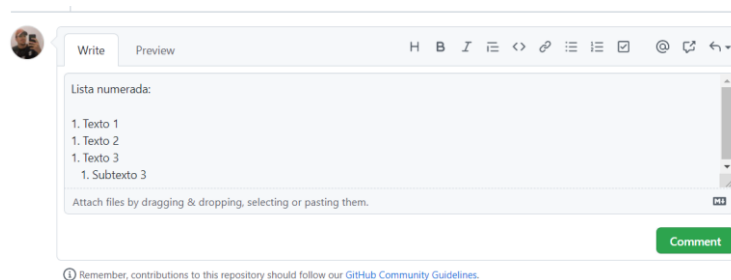
- **Títulos:** a marca para adicionarmos títulos é (**# título**) com espaço, lembrando que o número de hashtags indica o nível do título. Veja:

The first screenshot shows the 'Write' tab of the GitHub comment editor. The text area contains three lines: '# Título de nível 1', '## Título de nível 2', and '### Título de nível 3'. The second screenshot shows the 'Preview' tab, where the text is rendered as a heading structure: 'Título de nível 1' (h1), 'Título de nível 2' (h2), and 'Título de nível 3' (h3). Both screenshots include a 'Comment' button and a reminder to follow the GitHub Community Guidelines.

- É possível misturar marcas, exemplo: `___*texto em negrito e itálico*___`
- **Linha horizontal**: a marca para colocar uma linha horizontal é `(---` para linhas mais finas e `***` para linhas mais grossas). Veja:

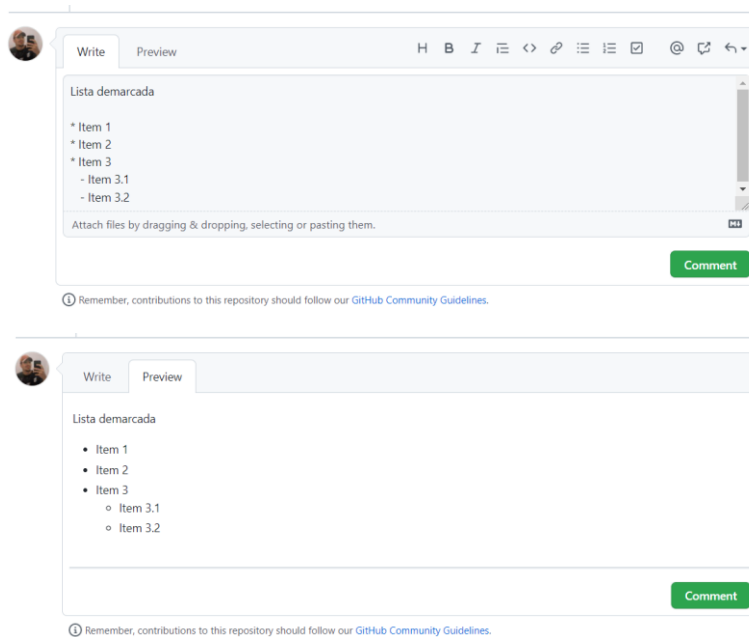


- **Lista numerada**: as marcas para uma lista numerada são as seguintes:



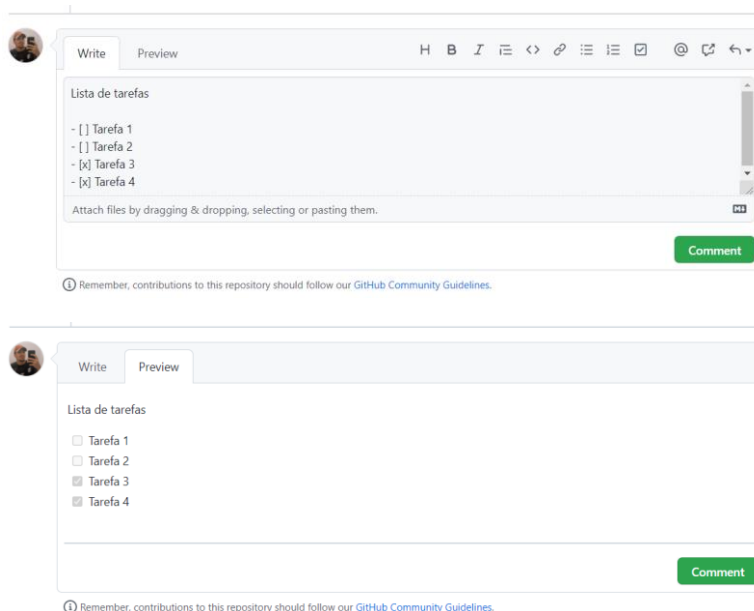
Os números de cada linha não importam, se quiser adicionar um subitem coloca-se três espaços antes do número (nesse caso, comece com 1 para não confundir com a lista principal), se atentando sempre com os espaços.

- **Lista com marcadores:** as marcas para uma lista com marcadores são as seguintes:



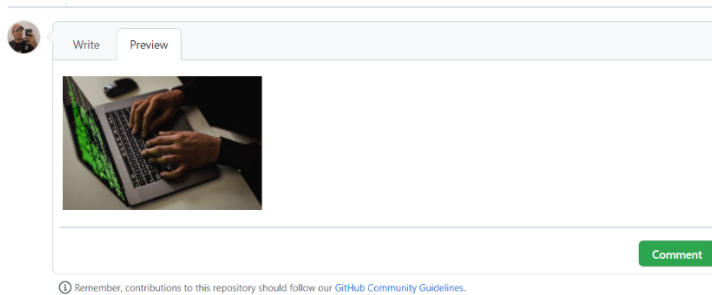
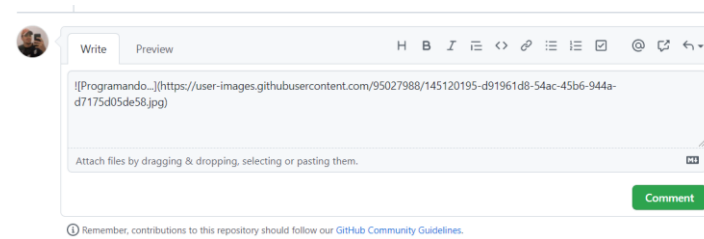
Se quiser adicionar um subitem coloca-se três espaços antes do número (pode-se usar * ou -), se atentando sempre com os espaços.

- **Lista de tarefas:** as marcas para uma lista de tarefas são as seguintes:

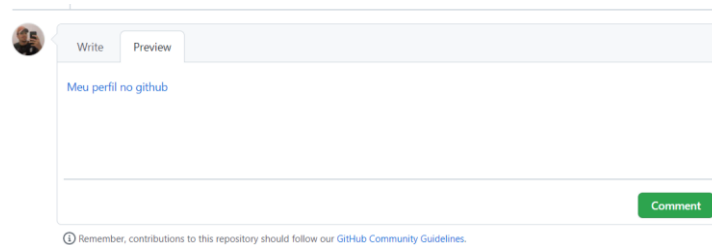
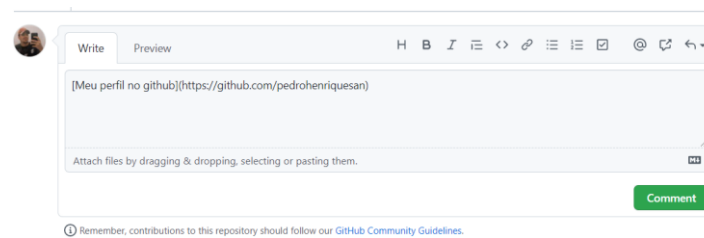


se atentando sempre com os espaços e, quando deseja marcar uma tarefa como concluída basta colocar um x dentro dos colchetes.

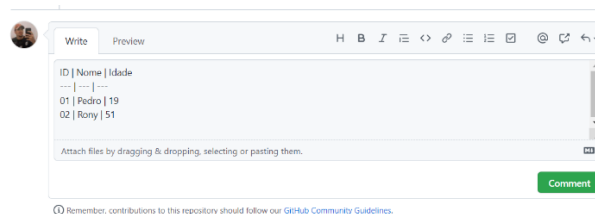
- **Imagens:** usa-se ![o que é a imagem](origem da imagem) ou basta arrastar para área inferior. Veja:



- **Links:** usa-se [texto aparente](url). Veja:

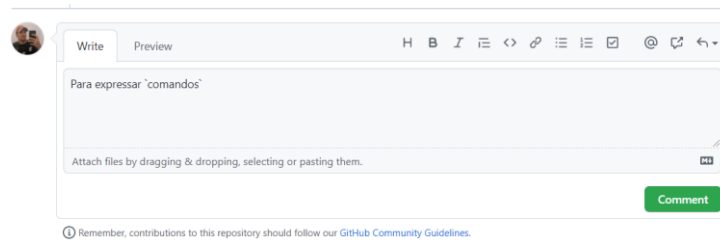


- **Tabelas:** são feitas da seguinte maneira:



Se atentando aos espaços necessários.

- **Comandos:** para expressar comandos basta indicar entre crases. Veja:



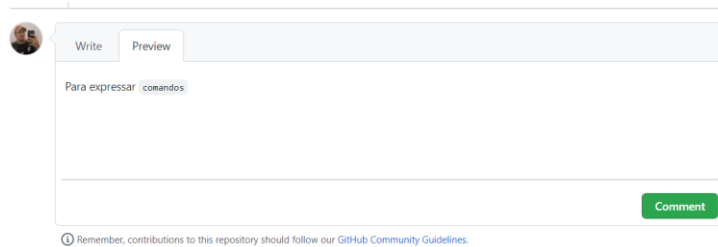
Write Preview H B I

Para expressar "comandos"

Attach files by dragging & dropping, selecting or pasting them.

Comment

Remember, contributions to this repository should follow our [GitHub Community Guidelines](#).



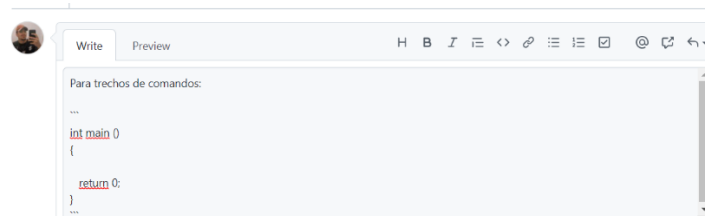
Write Preview

Para expressar `comandos`

Comment

Remember, contributions to this repository should follow our [GitHub Community Guidelines](#).

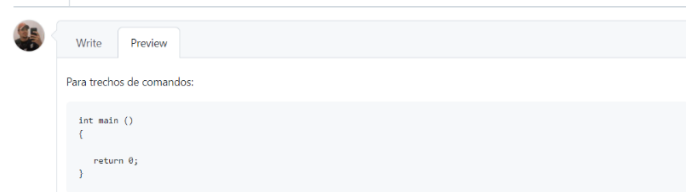
- **Trechos de código:** para expressar parte de códigos. Veja:



Write Preview H B I

Para trechos de comandos:

```
...  
int main ()  
{  
    return 0;  
}  
...
```

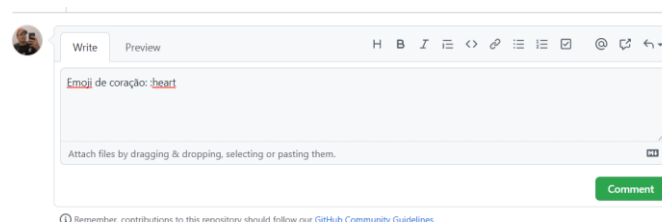


Write Preview

Para trechos de comandos:

```
int main ()  
{  
    return 0;  
}
```

- **Emojis:** usa-se o :, abrindo assim uma lista de emojis ou digita-se o nome do mesmo. Veja:



Write Preview H B I

Emoji de coração: `:heart`

Attach files by dragging & dropping, selecting or pasting them.

Comment

Remember, contributions to this repository should follow our [GitHub Community Guidelines](#).



Write Preview

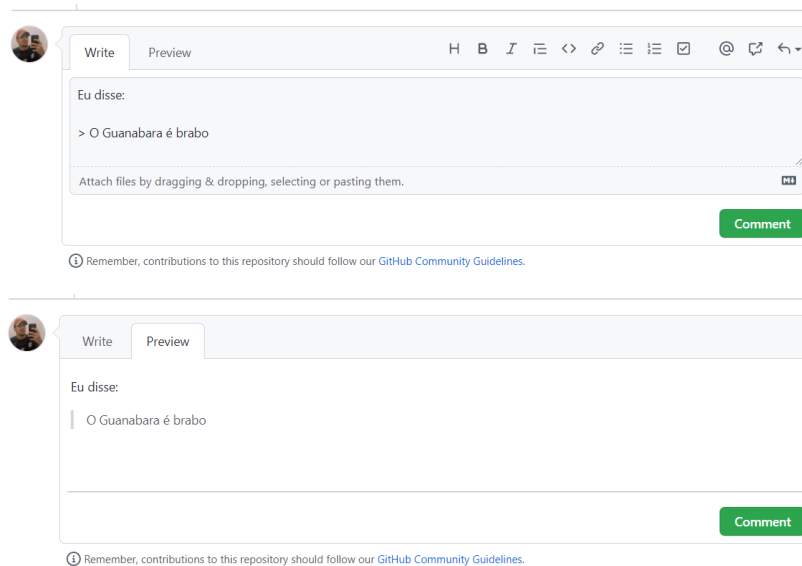
Emoji de coração: ❤️

Comment

Remember, contributions to this repository should follow our [GitHub Community Guidelines](#).

Acesse (github.com/ikatyang) que contém a maioria dos nomes de emojis no repositório emojis.

- **Citações:** Usa-se o símbolo de maior > antes das frases. Veja:

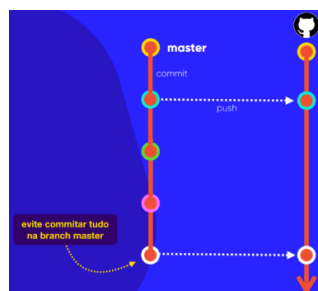


- Para inserirmos algum **símbolo que pertença a sintaxe** na linguagem markdown basta usa-se a **barra invertida** antes do símbolo.
- Para mencionarmos **pessoas** usa-se @ e para mencionarmos **issues ou pull-requests** usamos #.

Branches

Para começarmos com o assunto, imagine que ao criar um projeto novo estamos criando uma árvore, a qual possui seu tronco principal e suas ramificações, com esse pensamento, podemos associa-lo as branches, a qual significa literalmente galhos ou ramificações, bom, **quando se cria um projeto, nosso tronco principal é chamado de branch master que sempre é a versão final**, até que as versões futuras (branches secundárias) do seu software vão se tornar ao decorrer do tempo a branch master.

Sempre que iniciamos um projeto novo, ou seja, um repositório novo, é **obrigatório a existência da branch master**, essa branch vai recebendo commits e a cada commit temos um novo versionamento, depois do **versionamento local (master)** no git fazemos um push para o **versionamento remoto (origin)**, no entanto, **não é recomendado fazer todos os commits na nossa branch master**, isso, porque e cometemos um erro grave, estamos tornando nosso tronco principal falho, o que não é interessante.

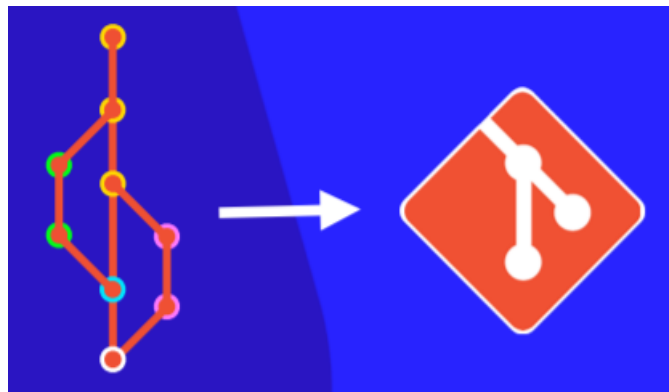


Para resolver esse problema e evitar erros muito graves na versão final do projeto (branch master) existem as branches secundárias (ramificações do projeto).

Imagine que o seu **projeto foi organizado em 3 equipes (3 branches)**, uma master, uma para o conteúdo e outra para o design, cada alteração referente a cada assunto será alterado na sua respectiva branch, **mantendo assim a branch master intacta**. Desse modo, **para juntar todas as alterações de cada branch** secundárias na branch master e obter uma nova versão **fazemos um merge**.

O merge nada mais é do que juntar um branch com outra branch, para fazê-lo é necessário conferir cada commit para não ocorrer nenhum erro. **Esse sistema do Github é muito comparado com universos paralelos, onde universos secundários podem ser alterados sem danificar o universo principal (branch master)**. Além disso, em projetos grandes e até mesmo em pequenos, **o uso das branches são de suma importância**.

Por fim, antes de partirmos para a prática, veja como o logo do Git é muito parecido com uma simulação de como funcionam as branches e o merge:



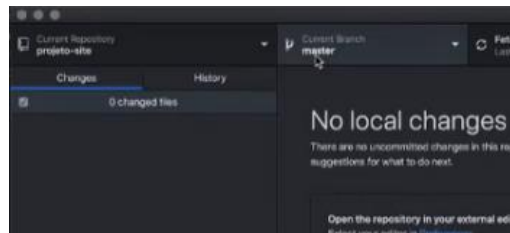
- A linha central é branch master e cada bolinha amarela é um commit.
- A linha a esquerda é a branch de conteúdo e cada bolinha verde é um commit na própria branch.
- Linha direita é a branca de design e cada bolinha rosa é um commit na própria branch.
- As linhas diagonais superiores são a criação dessas branches.
- As linhas diagonais inferiores representam o merge feito.

Branches na prática:

Para entendermos de fato o funcionamento das branches siga esse passo a passo, como se tivéssemos criando um novo projeto:

- Abra o github desktop para a criação de um novo repositório do mesmo modo que já foi apresentado nesse material anteriormente.
- Abra o repositório criado com o VScode.
 - **Nesse momento, se qualquer commit for feito, os mesmos estão alterando nossa branch master, o que não é indicado.**

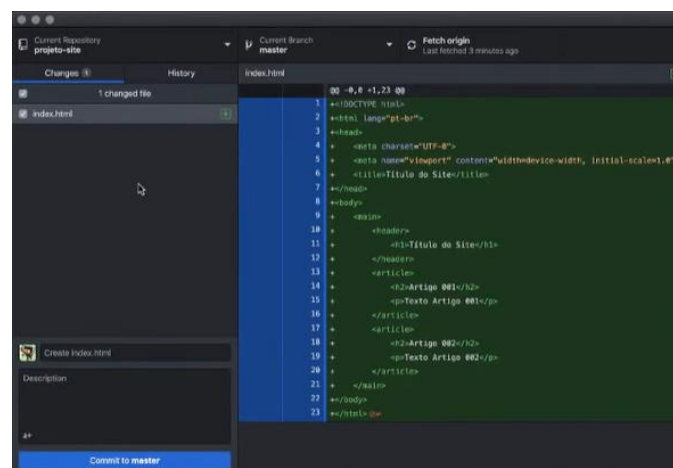
- No Github desktop é indicado em qual branch estamos fazendo alterações, se atente a isso.



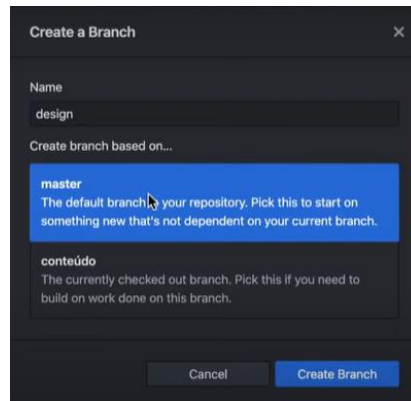
- Com o repositório aberto com o VScode cria-se a estrutura básica do seu site, ou seja, a estrutura básica do HTML junto com alguns elementos principais.

```
index.html x
1 <!DOCTYPE html>
2 <html lang="pt-br">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Título do Site</title>
7 </head>
8 <body>
9   <main>
10     <header>
11       <h1>Título do Site</h1>
12     </header>
13     <article>
14       <h2>Artigo 001</h2>
15       <p>Texto Artigo 001</p>
16     </article>
17     <article>
18       <h2>Artigo 002</h2>
19       <p>Texto Artigo 002</p>
20     </article>
21   </main>
22 </body>
23 </html>
```

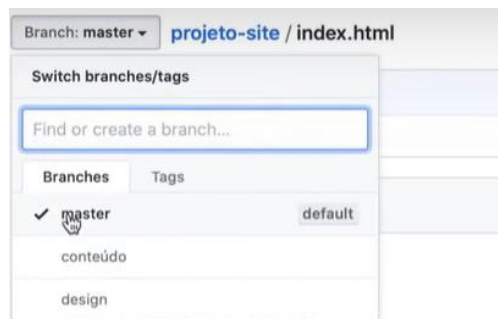
- Faça o commit e o push dessa primeira versão da branch master e observe que no Github desktop é indicado em qual branch está sendo feita essa alteração (commit to master), se atente a isso.



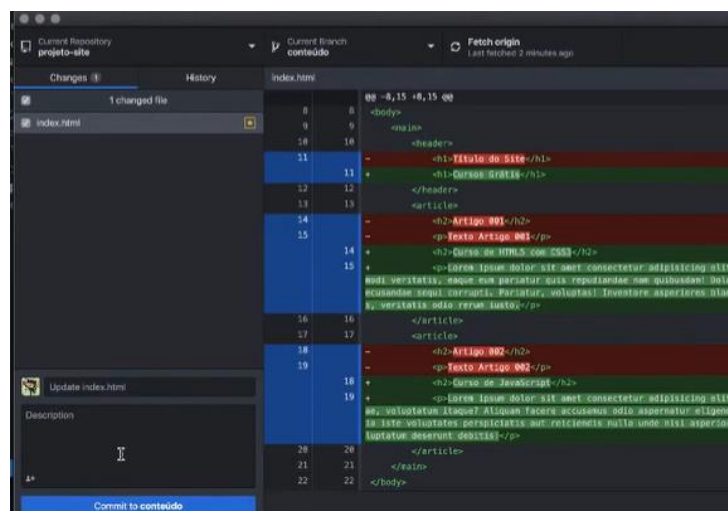
- Bom, partindo para as branches citadas acima, vamos criar primeiro a **branch de conteúdo**, ambas são criadas da mesma forma.
- No menu superior do Github desktop acesse branch → new branch → dê um nome para a mesma → selecione a qual branch pertence essa ramificação → create branch.



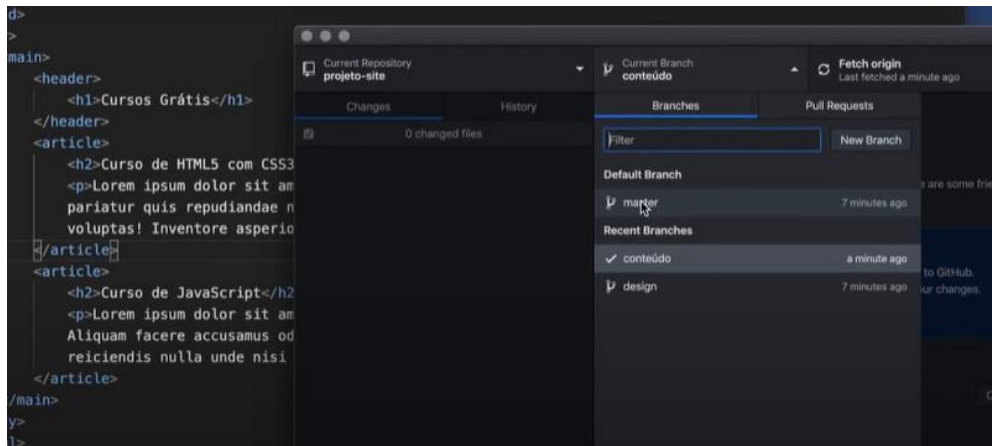
- Depois de criar a branch a publique no repositório, nesse momento o Github desktop **vai oferecer fazer um pull request (não o faça)**, voltando ao Github é possível selecionar a branch que deseja trabalhar.



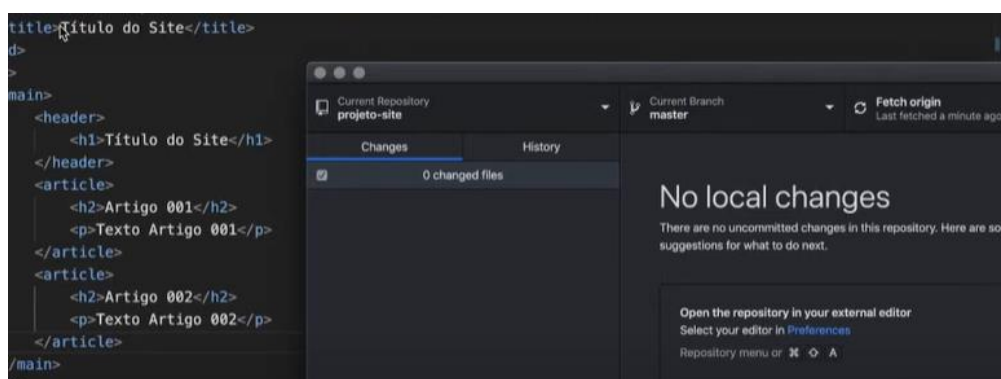
- A partir das duas branches criadas, se for necessário alterar o conteúdo faça-o na branch conteúdo e depois execute o merge, do mesmo modo para o design.
- **No github desktop selecione uma das branches criadas e abra a mesma com o github desktop**, talvez você se confunda no VScode, pois o mesmo não indica qual a branch aberta, mas no **Github desktop é possível ver nas áreas de commits (commit to conteúdo)**. Veja:



- Os commits e pushes são feitos da mesma forma em qualquer branch.
- **Para vermos esses “universos paralelos” de fato funcionando, basta mudar de branch no github desktop e observar a mágica no VScode**. Veja:



Branch conteúdo.



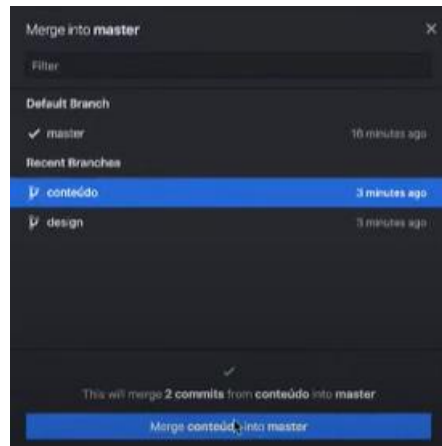
Branch master sem nenhuma alteração, uma vez que, ainda não foi feito um merge.

- Compare as duas imagens acima e veja que o **conteúdo adiciona** na branch conteúdo **não foi transferida para a branch master**, porque isso **só é possível depois de se realizar um merge** para a branch master.
- Ao final, alterando a branch design do mesmo modo como alteramos a branch conteúdo, vamos ter 3 “versões” diferentes que serão necessárias se unirem e, é aqui que entra o merge.
 - **Branch master (tronco principal).**
 - **Branch conteúdo (galho).**
 - **Branch design (galho).**

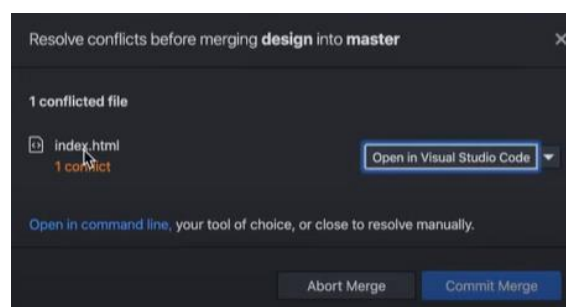
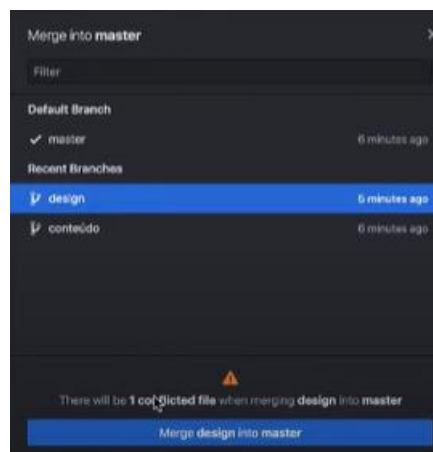
Um tópico muito importante de se comentar é que, caso aconteça de **dois programadores** atuantes em duas branches diferentes alterem a mesma linha de código com informações diferentes, isso acarretará um **conflito na hora de executar um merge**.

Merge na prática:

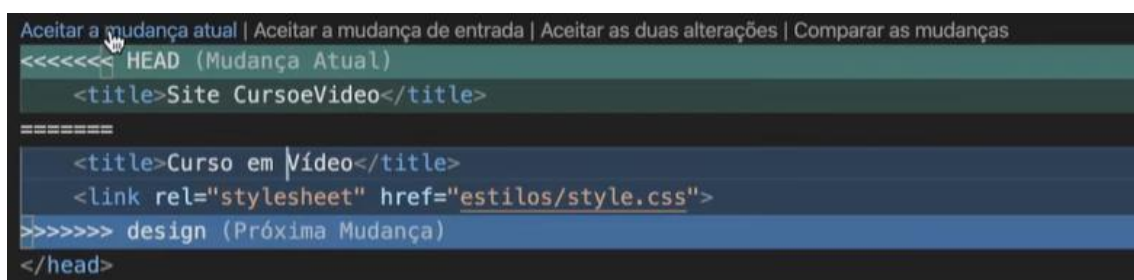
- Para fazermos um merge, a branch de origem deve estar selecionada no Github desktop, no nosso caso a branch master.
- É necessário se atentar aos commits feitos e os resultados obtidos.
- No menu superior do Github desktop selecione branch → merge into current branch.
- Selecione então a branch que deseja juntar a branch master.



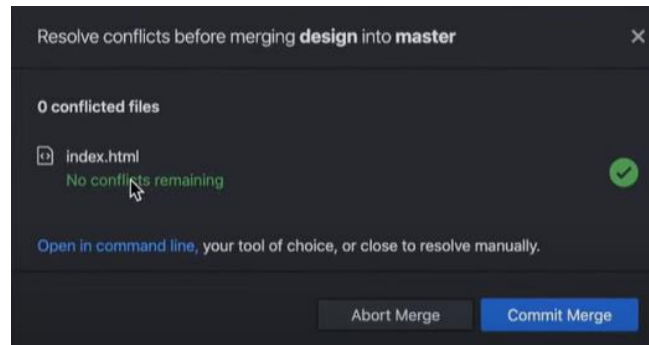
- Depois de fazer o merge, faça o push do mesmo modo que fazemos com os commits.
- Ademais, como foi dito acima, ao tentar fazer um merge com a branch de design vai existir um **conflito em virtude dos títulos diferentes** nas duas branches secundárias.



Não é possível fazer o merge sem resolver tal conflito.

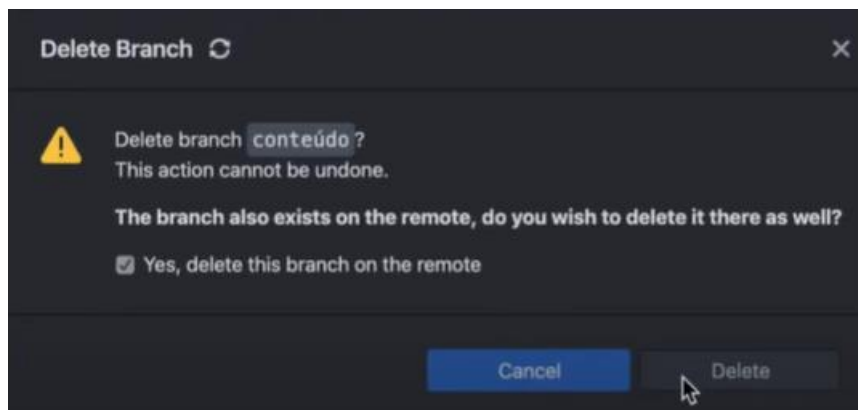


Selecione então como deseja prosseguir, qual alteração deseja manter.



Depois basta fazer o commit do merge.

Para finalizar o assunto de branches, considerando que você finalizou um projeto e, o mesmo possui muitas branches que já foram utilizadas e seus respectivos merges já foram feitos, é possível apaga-las. Veja:



Basta selecionar no menu superior do Github desktop branch → delete.

Além disso é possível deletar a branch apenas localmente.

