Aula I - Git

Fontes e Links de referência

O que é controle de versão | Atlassian Git Tutorial
https://git-scm.com/
https://git-scm.com/doc
https://github.com/
https://about.gitlab.com/
https://bitbucket.org/product/
https://www.alura.com.br/artigos/o-que-e-git-github
https://www.devmedia.com.br/diferencas-entre-os-sistemas-de-controle-de-versao-e-vcs-distribuidos/33532
https://dev.to/nopenoshishi/understanding-git-through-images-4an1
GitFichas

O que é Git?
Principais ferramentas Git no mercado
Instalação Git
Usando Git Bash
Usando GitHub
Qual a estrutura e funcionamento
Fluxo Git (Git Flow)
Branches
Comandos básicos

O que é Git?

De acordo com a documentação do Git, ele é um sistema de controle de versão distribuído gratuito e de código aberto projetado para lidar com tudo, desde projetos pequenos a muito grandes com velocidade e eficiência. O Git é fácil de aprender e ocupa pouco espaço com desempenho ultrarrápido. **Ele supera as ferramentas SCM como Subversion, CVS, Perforce e ClearCase.** Suas vantagens incluem recursos como ramificação (branch) local e vários fluxos simultâneos de trabalho .
O Git é o mais utilizado entre eles atualmente por conta de permitir uma cópia do projeto, um repositório do projeto em sua máquina, para que se possa trabalhar em cima dela e então enviá-la para outro repositório, o que se denomina repositórios distribuídos. Isso permite o trabalho de modo offline, antes da comunicação com outro servidor para o envio de versões, e assim por diante. Existem várias outras diferenças entre estas alternativas.

Principais ferramentas Git no mercado

Existem várias ferramentas para se trabalhar com Git no mercado e cada empresa geralmente escolhe a que mais é adequada para os seus sistemas e projetos. O que precisamos salientar aqui é que você precisa conhecer a essência do Git, desta forma não importa a ferramenta que seu time usa, os processos importantes sempre serão os mesmos. Vamos lá para uma breve lista das mais usadas no contexto (2022 e 2023).

GitHub

GitHub, um dos mais usados no mercado de TI, de propriedade da Microsoft, é totalmente gratuito, porém existe uma fração paga e pode ser usada em ambientes corporativos. (https://github.com/)

GitKraken

GitKraken é altamente intuitivo e confiável, eficiente, visualmente agradável. Sua interface é intuitiva, já que ela permite que os usuários rapidamente executem ações básicas. Ela ainda tem um recurso de arrastar e soltar para facilitar sua vida. (https://www.gitkraken.com/)

Bitbucket

Com a melhor integração do Jira e CI/CD integrado, o Bitbucket Cloud é a ferramenta Git nativa na solução Open DevOps da Atlassian. Junte-se a milhões de desenvolvedores que optam por criar no Bitbucket. (https://bitbucket.org/product/)

GitLab

Do planejamento à produção, o GitLab reúne equipes para encurtar os tempos de ciclo, reduzir custos, fortalecer a segurança e aumentar a produtividade do desenvolvedor. (https://about.gitlab.com/)

SmartGit

SmartGit é o cliente **Git GUI** da slant.co . SmartGit suporta GitHub, Bitbucket, GitLab e Azure DevOps. (https://www.syntevo.com/smartgit/)

Instalando Git

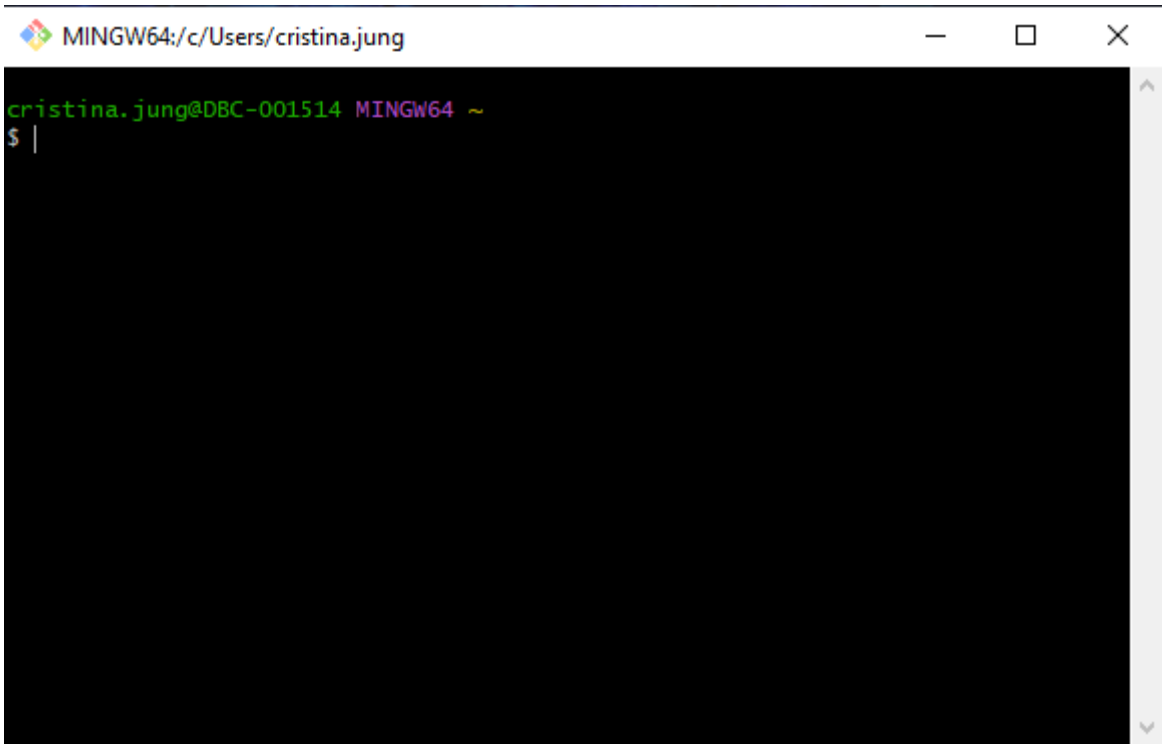
A instalação é bem simples e você poderá escolher como trabalhar com git quando estiver fazendo suas coisas individuais, porém enquanto estiver trabalhando em uma empresa, usa-se o ambiente desta empresa.

Vamos ao tutorial de como instalar:

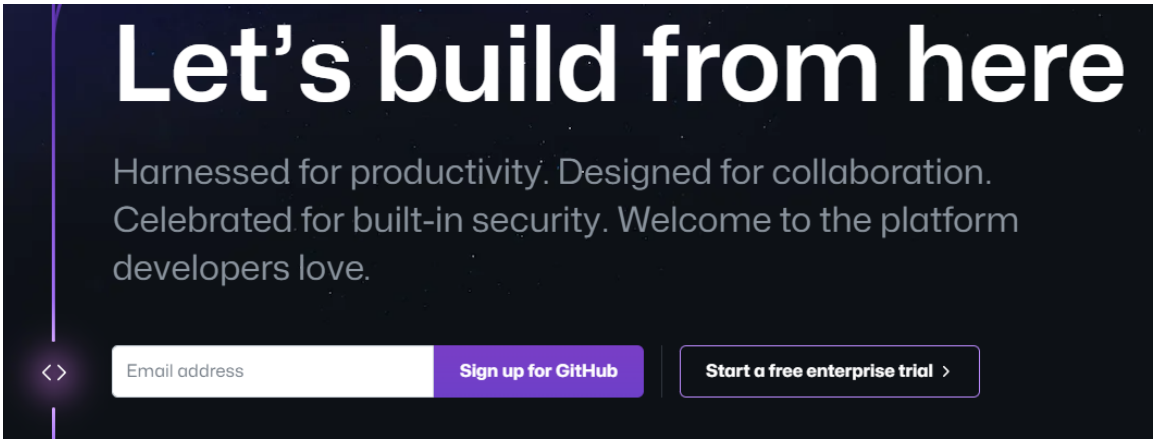
1. Acesse o endereço: (Git);
2. E faça download do git na versão do SO que você usa;

3. Depois do download, só executar o arquivo e seguir os passos da instalação;
4. **Caso você esteja utilizando Linux**, algumas distribuições já vêm com o Git instalado, então é só abrir o Terminal e digitar "git" para verificar isto. Se ele não estiver instalado, o gerenciador de pacotes da sua distribuição, por exemplo o APT para Ubuntu ou derivados de Debian, o DNF para Fedora, com certeza possuem uma versão do Git, basta executar a instalação por meio da linha de comando.

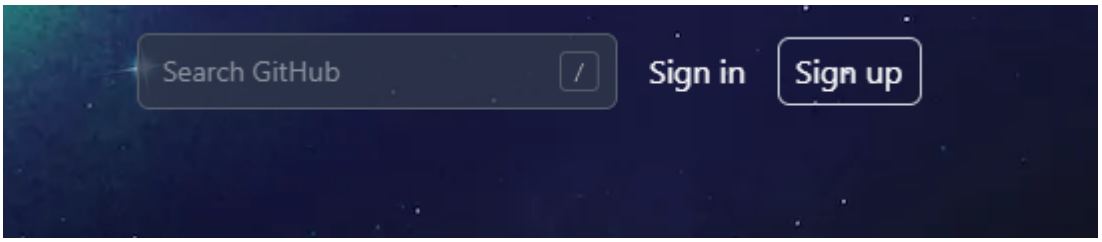
5. Para abrir o terminal, basta acessar o menu iniciar do windows ou o recurso similar no SO que você usa. A janela do terminal será aberta, como você pode ver na próxima imagem;



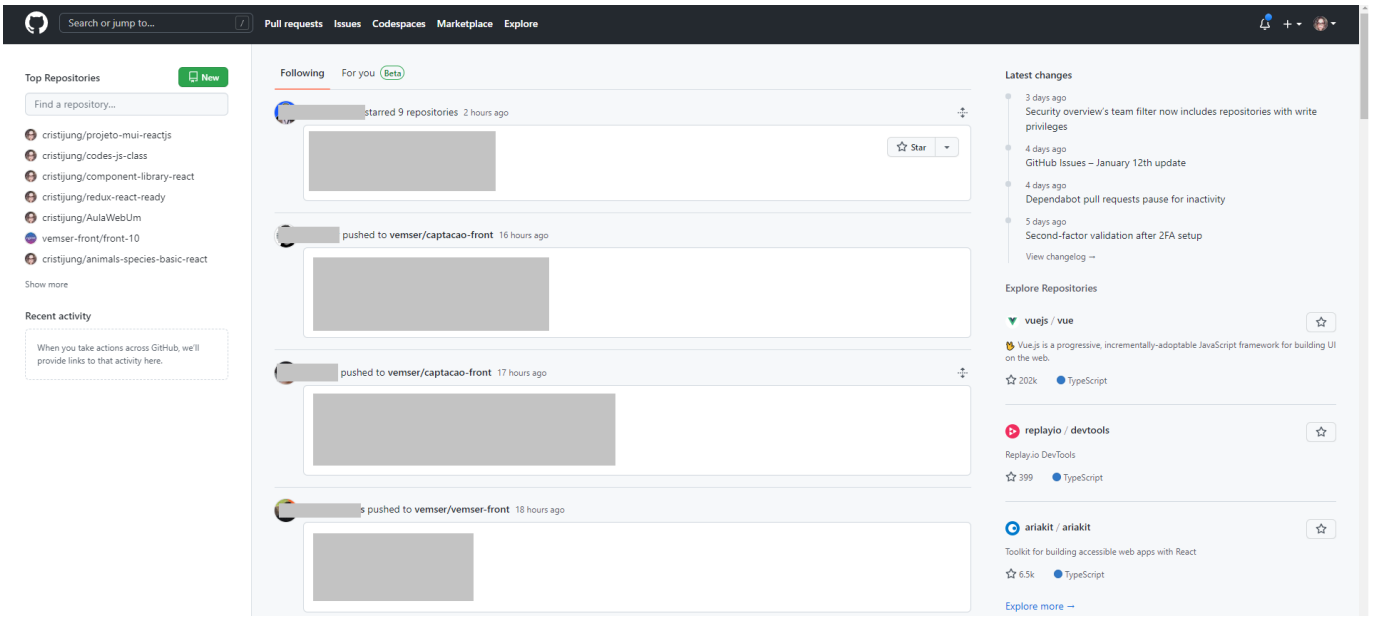
6. Agora vamos usar o Github, que será a nossa ferramenta Git no Vem Ser. Se você não tem uma conta no GitHub, é bem fácil de criar, para isso, você precisa somente de um email;
7. Acesse o endereço: (https://github.com/);



8. Clique no Sign up após de criar a conta;



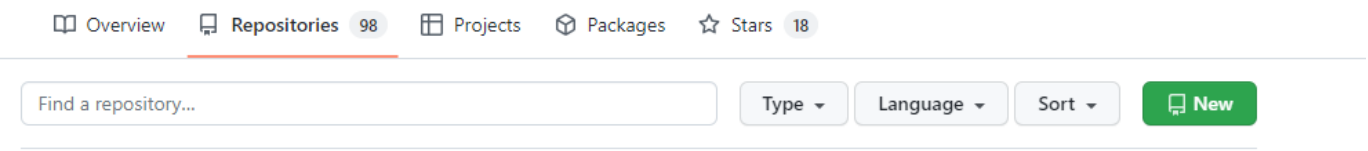
9. Quando você acessar o seu GitHUb, verá algo mais ou menos como a tela abaixo:



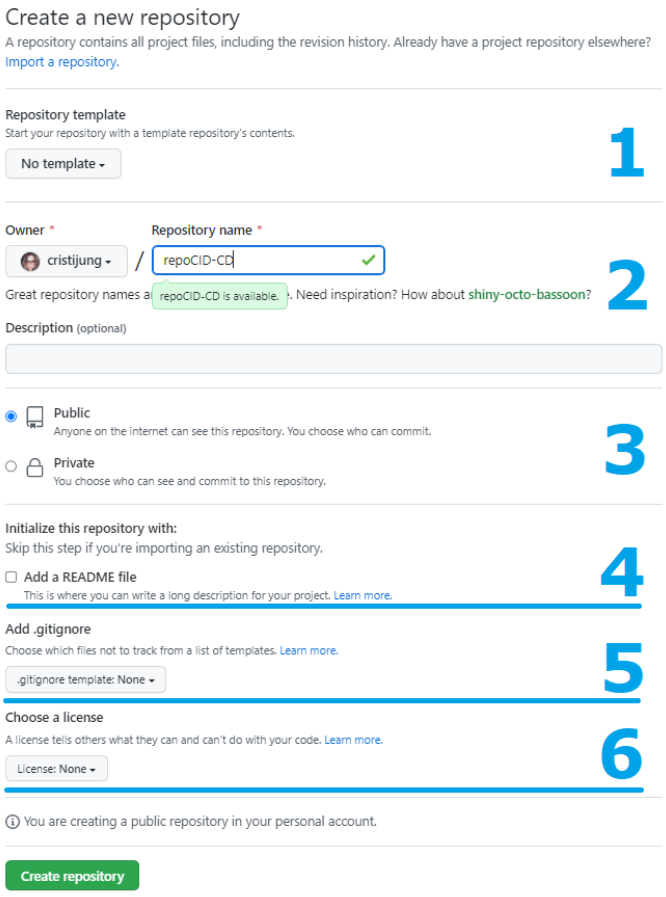
Dica para quem é iniciante no GitHub:
Usando chave SSH para não precisar configurar nome de usuário e email a cada trabalho no seu projeto. Usando o protocolo SSH, você pode se conectar a servidores e serviços remotos e se autenticar neles. Com chaves SSH, você pode se conectar ao GitHub sem fornecer seu nome de usuário e personal access token em cada visita. Você também pode usar uma chave SSH para assinar confirmações.
Para configurar a sua chave SSH, pode seguir os passos da documentação do GitHub no link abaixo:
Gerando uma nova chave SSH e adicionando-a ao agente SSH - GitHub Docs

10. Para criarmos um repositório podemos criar um novo repositório e algumas configurações específicas e relacionadas ao projeto que você ou seu time irá desenvolver.

11. Observe a imagem abaixo:



12. Basta clicar no botão verde para criar novo repositório, você verá a seguinte tela:

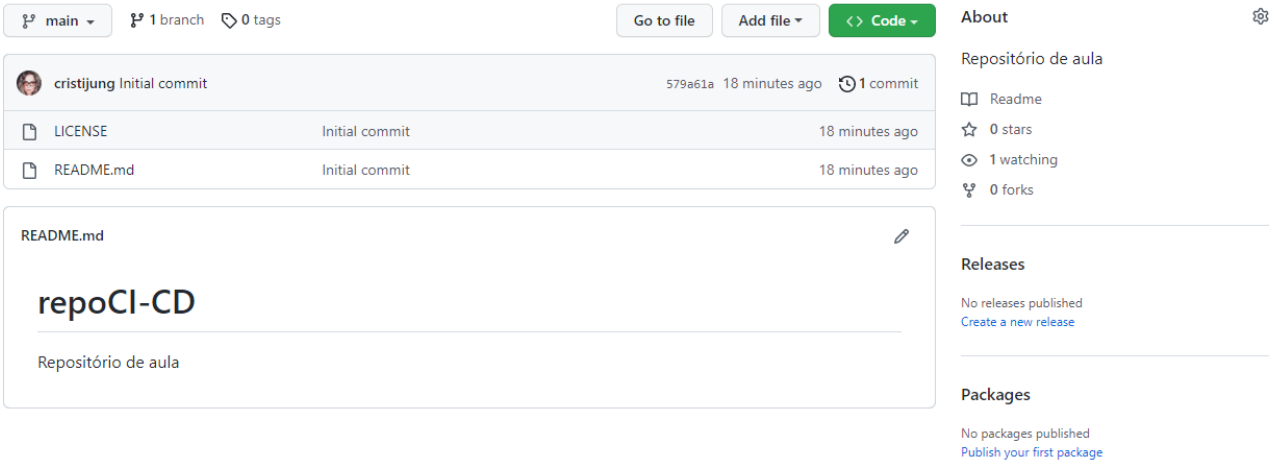


Legenda:

1. **Se o repositório se origina de um modelo ao ser criado;**
2. **Nome do repositório + a descrição deste;**
3. **Se o seu repositório vai ser público (todo mundo vê) ou se será privado. Quando um repositório for privado somente você e seus contribuidores (que você pode adicionar) terão acesso;**
4. **Marque esta opção se deseja adicionar o arquivo markdown. Um arquivo README é um arquivo de orientação de clonagem, instalação ou mesmo documentação da aplicação - Pode ser criada neste momento ou mesmo depois;**
5. **Gitignore: aqui você pode marcar as extensões dos arquivos que serão ignorados quando o 'push' for dado ou mesmo no deploy da aplicação;**
6. **Nesta opção escolhemos a licença que desejamos atribuir ao projeto que estamos criando - geralmente a licença escolhida é MIT, mas não é uma opção obrigatória;**

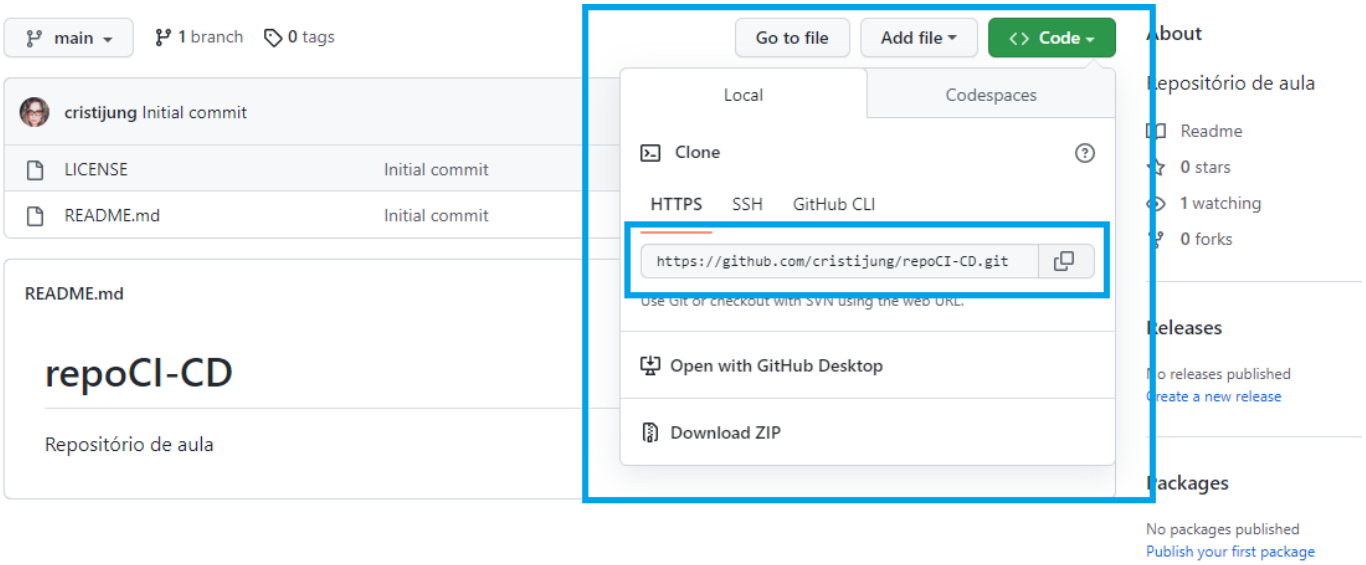
13. A próxima imagem é a tela que você verá com o repositório criado. Esta visualização é quando você já configurou a criação do README para o repositório. Se optar por não criá-lo neste momento, ele virá com o passo a passo de como

você clona o repositório e demais operações, como vistas no link → [Clonar um repositório - GitHub Docs]

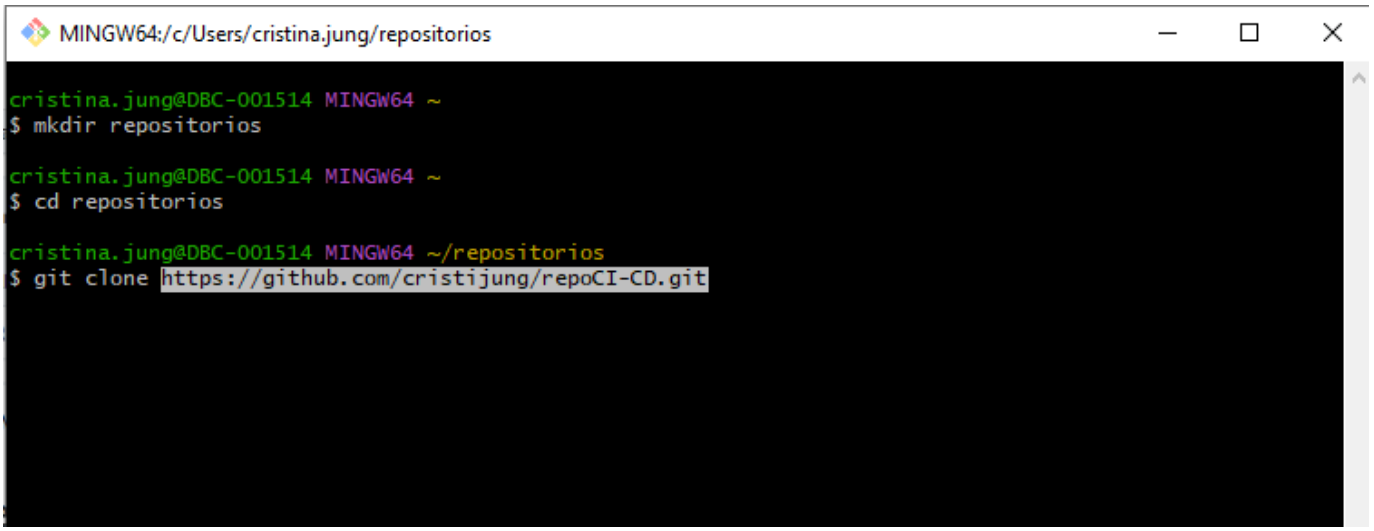


14. Agora que já temos um repositório poderemos trabalhar com ele e as operações abertas são:
    a. Clonar repositório - fazer uma cópia do repositório;
    b. Ter acesso ao histórico de atividade do repositório, como commits, PRs e merges executadas pelos outros desenvolvedores;
    c. Fazer alterações no código;
    d. Commitar no código;
    e. Estar sempre atualizado com as edições de outros usuários deste repositório;
    f. Participar de Code Review do seu time;

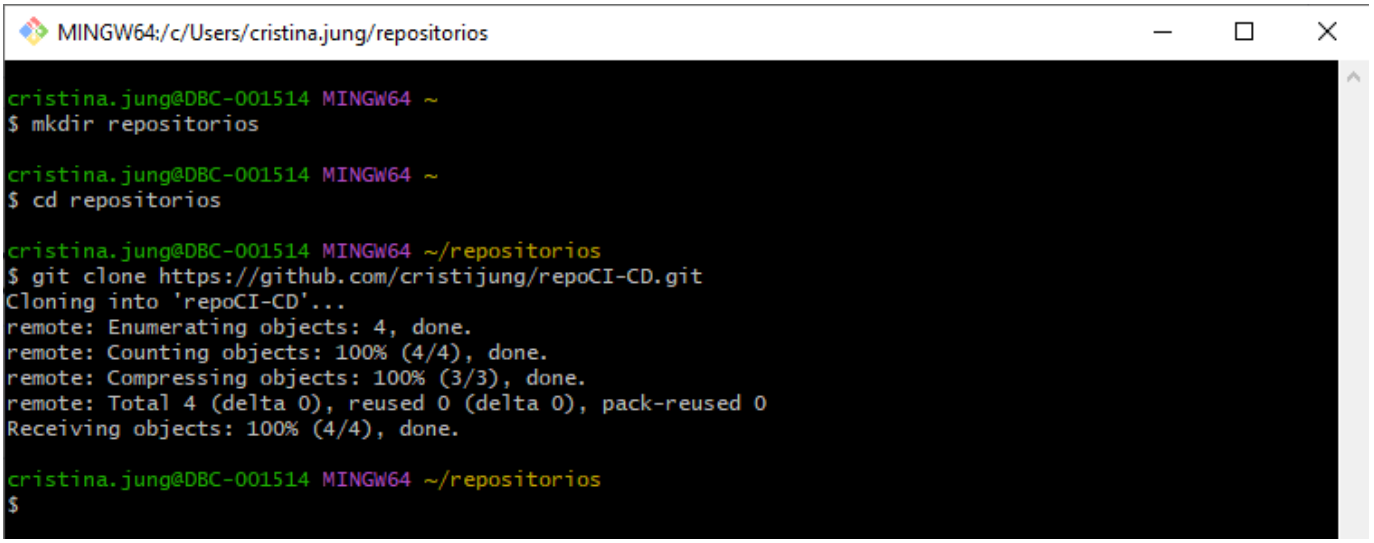
15. Para clonar um repositório no GitHub:



16. Podemos acessar o terminal do Git Bash ou outro terminal que você esteja acostumado a usar e vamos começar a usar os comando básicos do Git:



17. Depois de clonado, será possível acessar o diretório criado pela clonagem. Será possível listar também;



Diferença entre Clonar & Baixar um repositório Git	
Clonar	**Download**
Ao clonar um repositório **é feito uma cópia completa de todo o histórico de commits e todas as ramificações do repositório.** Isso significa que você terá acesso a todas as versões anteriores dos arquivos, bem como a **capacidade de criar novos commits e interagir diretamente com o repositório - ser um contribuidor.** O comando para clonar um repositório Git é geralmente git clone <URL do repositório>. **Ao clonar um repositório, você obtém uma cópia completa e funcional do projeto, incluindo todas as informações do Git, como histórico de commits, ramificações e tags.**	Ao baixar o repositório, **teremos acesso a apenas uma versão "snapshot" dos arquivos no momento em que o arquivo compactado foi gerado. Não há informações de controle de versão ou histórico de commits disponíveis.** Geralmente, os repositórios compactados são fornecidos no formato de um arquivo ZIP ou tarball. **Ao baixar um repositório compactado, você não obtém as informações do Git, como histórico de commits ou ramificações.** Portanto, você não terá a capacidade de interagir com o repositório como faria ao cloná-lo.

Estrutura e funcionamento do Git

A principal diferença entre o Git e qualquer outro VCS (Version Control System), e a maneira como o Git trata seus dados. Conceitualmente, a maioria dos outros sistemas armazenam informação como uma lista de mudanças nos arquivos. Estes sistemas (CVS, Subversion, Perforce, Bazaar, e assim por diante) tratam a informação como um conjunto de arquivos e as mudanças feitas em cada arquivo ao longo do tempo.

O fluxo de trabalho básico Git é algo assim:

- Você modifica arquivos no seu diretório de trabalho.
- Você prepara os arquivos, adicionando imagens deles à sua área de preparo.
- Você faz commit, o que leva os arquivos como eles estão na área de preparo e armazena essa imagem de forma permanente para o diretório do Git.

Os Três Estados (imagem acima)

Este é o principal fato a se lembrar sobre Git: o Git tem três estados principais que seus arquivos podem estar:

- committed;
- modificado (modified);
- e preparado (staged).

Committed significa que os dados estão armazenados de forma segura em seu banco de dados local. **Modificado** significa que você alterou o arquivo, mas ainda não fez o commit no seu banco de dados. **Preparado** significa que você marcou a versão atual de um arquivo modificado para fazer parte de seu **próximo commit**.

Isso nos leva a três seções principais de um projeto Git:

- **O diretório Git:** é onde o Git armazena os metadados e o banco de dados de objetos de seu projeto. Esta é a parte mais importante do Git, e é o que é copiado quando você clona um repositório de outro computador.
- **O diretório de trabalho:** é uma simples cópia de uma versão do projeto. Esses arquivos são pegos do banco de dados compactado no diretório Git e colocados no disco para você usar ou modificar.
- **Área de preparo:** é um arquivo, geralmente contido em seu diretório Git, que armazena informações sobre o que vai entrar em seu próximo commit. É por vezes referido como o **"índice"**, mas também é comum referir-se a ele como área de preparo **(staging area).**

Fluxo completo Git

Branches

Fluxo de trabalho de uma branch

O que é uma Branch? Uma branch no contexto Git é quando criamos uma ramificação do sistema, é como se fizéssemos uma cópia da imagem do sistema e puxamos para o lado. Nesta branch podemos editar e codar conforme a funcionalidade que estamos desenvolvendo e isso tudo, sem interferir na principal, cujo nome é main.

A ideia central por trás do fluxo de trabalho de uma branch que contém recursos é que todo desenvolvimento destes recursos deve ocorrer em uma branch separada em vez de na branch principal.

Atualmente não utilizamos mais o termo master como repositório principal, e sim main, sendo esse é o novo padrão do GitHub desde o fim de 2020. Essa nomenclatura é mais inclusiva e facilita o entendimento entre pessoas desenvolvedoras.

Como funciona?

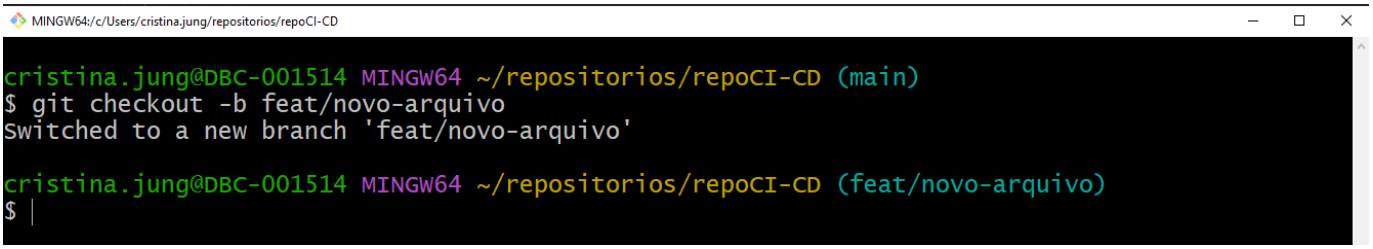
O fluxo de trabalho da branch de recursos pressupõe um repositório central, e a ramificação principal (main) representa o histórico oficial do projeto. Em vez de fazer o commit direto na ramificação principal local, os desenvolvedores criam uma nova ramificação sempre que começam a trabalhar em um novo recurso. As branches dos recursos devem ter nomes descritivos e semânticos, como **itens-de-menu-animados** ou **problema-#-1061**. A ideia é dar um objetivo claro e bastante focado a cada branch. O Git não faz distinção técnica entre a branch main e as de recursos, então os

desenvolvedores podem editar, preparar e fazer o commit de alterações em uma branch normal de desenvolvimento.

Para criar uma nova branch, podemos usar o comando:

git checkout -b <nome-da-branch>

Este comando cria a branch e já troca para trabalhar nela:



Para ver quais as branches que existem no repositório e para ver em qual estou

git branch

```
cristina.jung@DBC-001514 MINGW64 ~/aula-git/aula01-git (feat/novo-arquivo)
$ git branch
* feat/novo-arquivo
  main
```

Para trocar e 'navegar' entre branches:

git checkout <nome-da-branch>

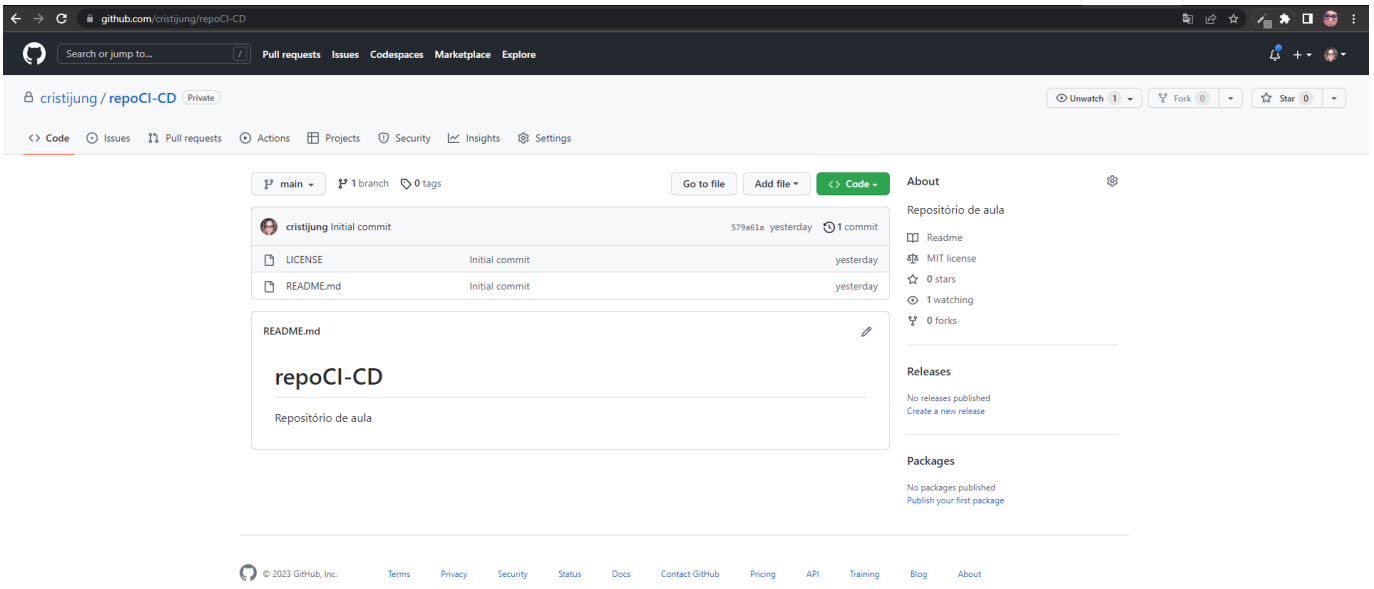
```
cristina.jung@DBC-001514 MINGW64 ~/repositorios/repoCI-CD (feat/novo-arquivo)
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.

cristina.jung@DBC-001514 MINGW64 ~/repositorios/repoCI-CD (main)
$
```

Fique sempre atento para estar trabalhando e codando na branch correta! Você poderá criar quantas branches precisar, mas não pode esquecer que a organização do código será sempre do desenvolvedor 'proprietário', então seja responsável pelas suas tarefas.

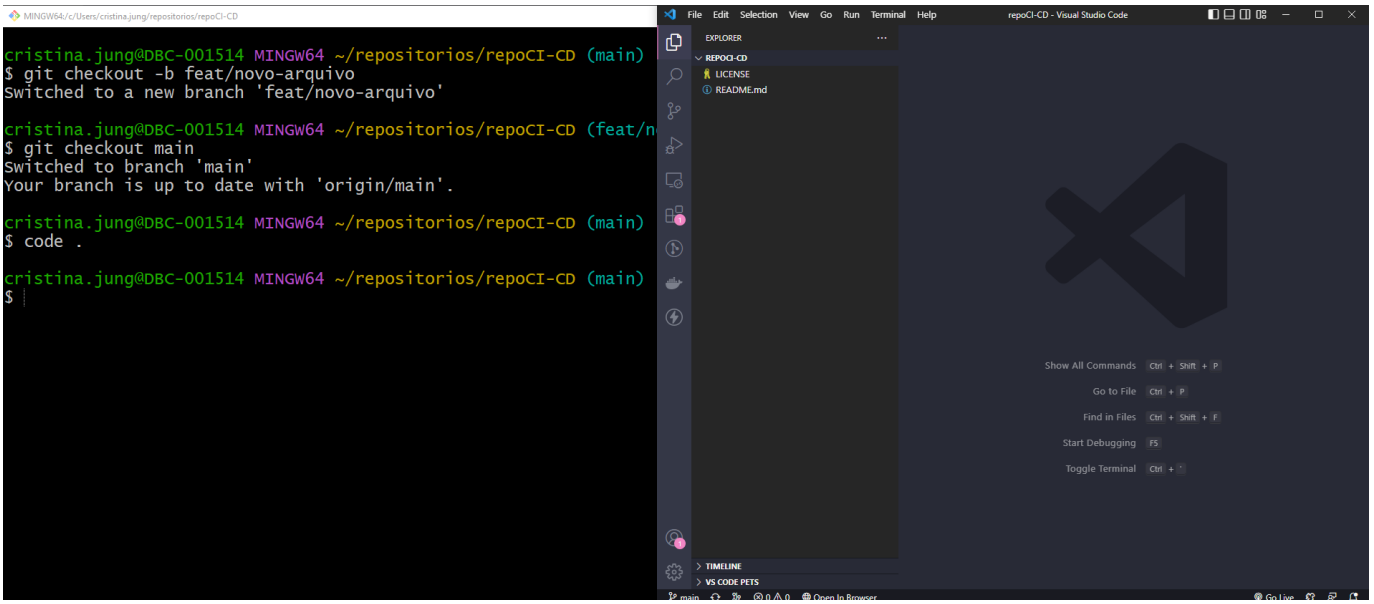
Uma dica: enquanto sua branch ou arquivo que você tenha criado no repositório, o git não enviará para a área remota → ele não sobre branches ou arquivos vazios.

Vamos acessar nosso repositório GitHub pelo navegador;



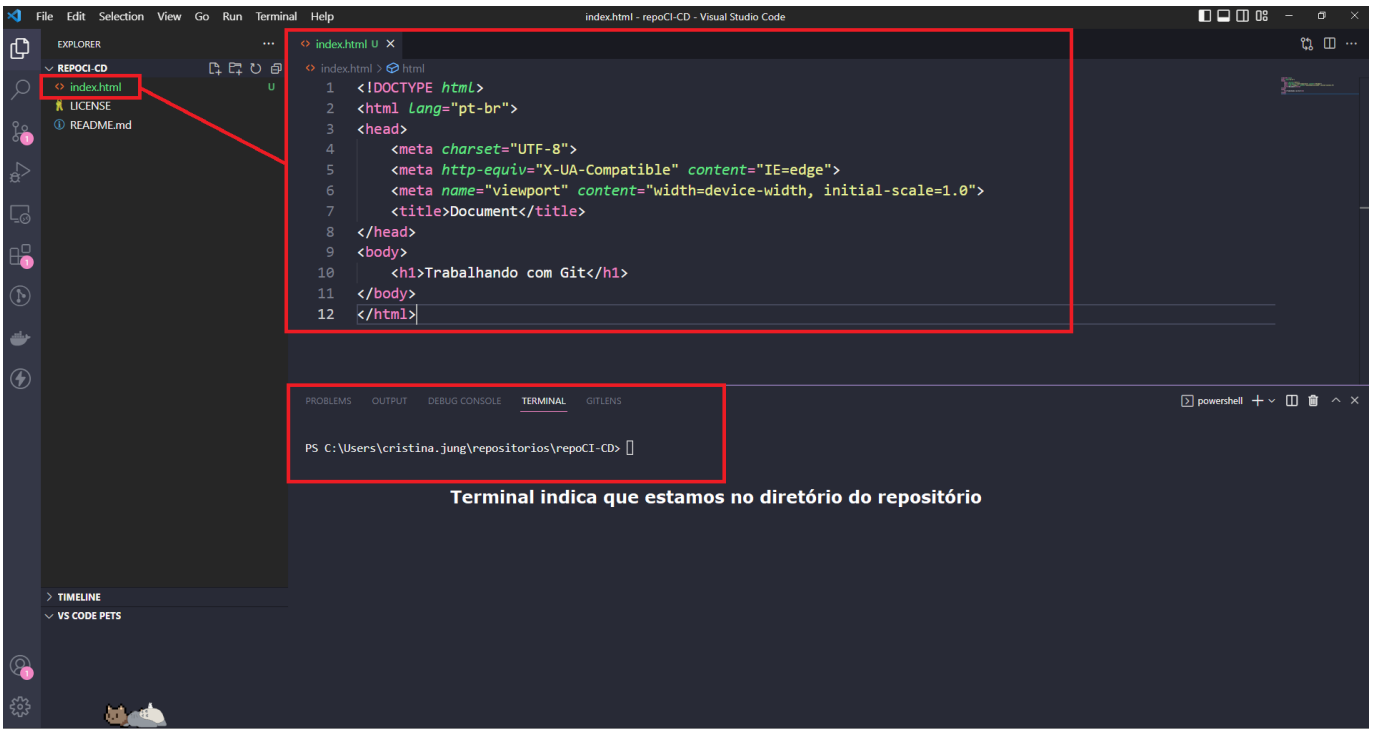
Para criar um arquivo ou mesmo fazer alguma edição diretamente de forma remota, podemos abrir uma IDE neste repositório. Se pressionarmos a tecla `.` o GitHub irá abrir o VSCode. Desta forma podemos fazer alterações diretamente no repositório remoto.

Nestes próximos exemplos, vamos usar como IDE o próprio VSCode, então, desta forma, acessamos a pasta onde está o repositório e digitamos: **code .**
Se preferir, poderá usar a IDE que mais lhe agrada.



No VsCode:

Abrimos o terminal e vamos começar a conhecer os principais comandos do Git. Uma observação aqui: você não precisa usar o terminal integrado da IDE, poderá continuar usando o Git Bash tranquilamente.



Terminal indica que estamos no diretório do repositório

No exemplo acima, foi criado somente um arquivo 'index.html' dentro do repositório para que possamos trabalhar com estas informações.

Primeiros comandos a serem executados neste tutorial:

git fetch: comando que busca branches e/ou tags (coletivamente, "refs") de um ou mais outros repositórios, juntamente com os objetos necessários para completar seus históricos. Muito usado quando estamos trabalhando em um repositório que é constantemente atualizado. Uma opção que podemos usar nestes casos é a adição de mais de um comando ao mesmo tempo, como: **git fetch && git pull →** neste caso estaremos dando duas orientações, uma para buscar qualquer branch, arquivos e histórico da branch e o pull (veremos daqui a pouco), define que estamos 'baixando' para o diretório local, desta forma, você garante que todas as alterações que foram feitas por todos os contribuidores do repositório.

```
cristina.jung@DBC-001514 MINGW64 ~/repositorios/repoCI-CD (main)
$ git fetch && git pull
Already up to date.
```

git status: o comando git status nos dá todas as informações necessárias sobre a branch atual.

git checkout: como já vimos, é um comando que podemos navegar entre branches, pois é importante que toda a vez que vamos trabalhar numa branch, então precisamos acessar esta ramificação.

git branch: este comando exibe as branches existentes no repositório e destaca a branch em que nos encontramos. Na imagem abaixo, foi dado git branch para que possamos ver em qual branch nos encontramos e em seguida git checkout + o nome da branch. Observe que nos encontramos na branch: **feat/novo-arquivo**

```
cristina.jung@DBC-001514 MINGW64 ~/repositorios/repoCI-CD (main)
$ git branch
  feat/novo-arquivo
* main

cristina.jung@DBC-001514 MINGW64 ~/repositorios/repoCI-CD (main)
$ git checkout feat/novo-arquivo
Switched to branch 'feat/novo-arquivo'

cristina.jung@DBC-001514 MINGW64 ~/repositorios/repoCI-CD (feat/novo-arquivo)
$
```

Um ponto a observar: nome das branches!

As branches dentro de um contexto de projeto devem possuir um padrão de nomes, não saímos por aí colocando qualquer nome que aparecer na nossa cabeça! Precisamos obedecer ao padrão de projeto que a empresa e/ou o time que estamos trabalhando usam.

Geralmente existe um consenso geral, porém será sempre o padrão da empresa ou time que irá definir.
Nos tópicos abaixo, uma listagem base de como devemos definir nomes para as branches.

Branches

Nomes de branches são compostos de 3 partes:

1. Tipo ou categoria da branch. Os tipos podem ser os seguintes:
    a. **docs:** apenas mudanças de documentação;
    b. **feat:** uma nova funcionalidade;
    c. **fix:** a correção de um bug;
    d. **perf:** mudança de código focada em melhorar performance;

e. **refactor:** mudança de código que não adiciona uma funcionalidade e também não corrigi um bug;
f. **style:** mudanças no código que não afetam seu significado (espaço em branco, formatação, ponto e vírgula, etc);
g. **test:** adicionar ou corrigir testes.

2. O que o branch faz em si.
3. Código da task no Jira ou Trello→ **Ex.: Ts-123**. (exemplo de mais usados)

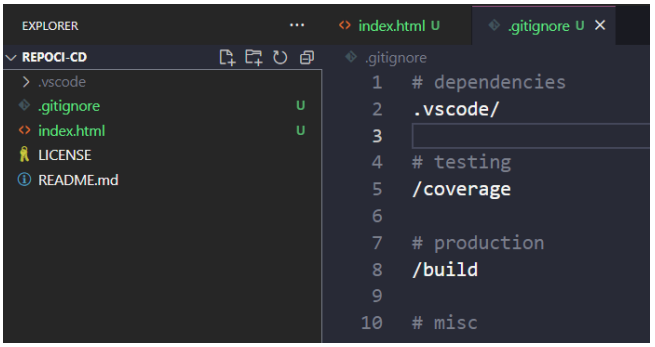
Exemplos de alguns nomes de branches que podem existir em nossa aplicação:

- **feat-cadastro-usuarios-Ts-123 → feat/cadastro-usuarios-Ts-123**
- **refactor-edicao-colaboradores-Ts-355**
- **fix-busca-checklists-Ts-232**

É preciso cuidar também dos caracteres especiais, no link abaixo, você encontrará as orientações nas documentações específicas:
Lidando com caracteres especiais nos nomes do branch e tags - GitHub AE Docs

O gitignore

Um arquivo gitignore específica arquivos intencionalmente não rastreados que o Git deve ignorar. No gitignore, cada linha especifica um padrão a ser ignorado. Ao decidir se deve ignorar um caminho, o Git normalmente verifica gitignore os padrões de várias fontes, com a seguinte ordem de precedência, do mais alto ao mais baixo declarado.
Vamos fazer o nosso gitignore agora e você poderá sempre consultar a documentação neste Ignorar arquivos - GitHub Docs
Só criarmos dentro da nossa pasta o arquivo .gitignore e neste caso colocaremos o .vscode/ para que o git ignore este.



Acessando os três estados do Git

Neste exemplo que estamos trabalhando, já criamos um simples arquivo HTML, porém ele se encontra somente no diretório local. Para que possamos verificar o que está acontecendo com eles podemos usar o comando **git status**. Este comando além de mostrar em qual branch estamos, irá exibir os arquivos modificados (em vermelho) **no diretório local. → 1º estado**

```
cristina.jung@DBC-001514 MINGW64 ~/repositorios/repoCI-CD (feat/novo-arquivo)
$ git status
On branch feat/novo-arquivo
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        index.html

nothing added to commit but untracked files present (use "git add" to track)

cristina.jung@DBC-001514 MINGW64 ~/repositorios/repoCI-CD (feat/novo-arquivo)
$
```

Para que possamos preparar estas alterações para que possam ser rastreadas, vamos usar o comando:

git add: ao criarmos, modificarmos ou excluirmos um arquivo, essas alterações acontecerão em nosso espaço de trabalho local e não serão incluídas no próximo commit (a menos que alteremos estas configurações). Precisamos usar o comando **git add** para incluir as alterações de um ou vários arquivos em nosso próximo commit. Para adicionar um único arquivo, usamos a sintaxe: **git add <nome-do-arquivo>** e, se desejamos adicionar todas as alterações, usamos: **git add .**

Na imagem abaixo, você verá a sequência de comandos: **git add .** e git status para que possa verificar a adição dos arquivos (em verde) **→ neste momento eles estarão na área de preparação, prontos para serem comitados. → 2º estado**

```
cristina.jung@DBC-001514 MINGW64 ~/repositorios/repoCI-CD (feat/novo-arquivo)
$ git add .

cristina.jung@DBC-001514 MINGW64 ~/repositorios/repoCI-CD (feat/novo-arquivo)
$ git status
On branch feat/novo-arquivo
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   .gitignore
        new file:   index.html


cristina.jung@DBC-001514 MINGW64 ~/repositorios/repoCI-CD (feat/novo-arquivo)
$
```

Agora vamos commitar estes arquivos, no momento em que usamos o **git commit** estamos efetivamente dizendo ao git que os arquivos adicionados para o monitoramento serão de fato acrescentados com suas modificações ao diretório git local e posteriormente deverão ser enviados para o diretório git remoto.

git commit: talvez esse seja o comando mais usado do Git. Quando chegamos a determinado ponto em desenvolvimento, queremos salvar nossas alterações (talvez após uma tarefa ou resolução de problema específica). Git commit é como definir um ponto de verificação no processo de desenvolvimento. Também precisamos escrever uma mensagem breve para explicar o que desenvolvemos ou alteramos no código-fonte e git commit salva suas alterações no espaço de trabalho local. Sintaxe: **git commit -m 'mensagem do commit'**

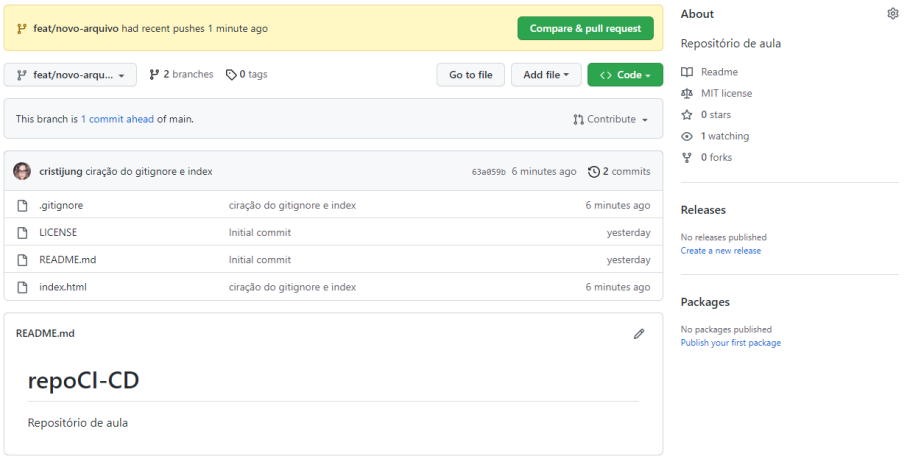
```
cristina.jung@DBC-001514 MINGW64 ~/repositorios/repoCI-CD (feat/novo-arquivo)
$ git commit -m 'ciração do gitignore e index'
[feat/novo-arquivo 63a059b] ciração do gitignore e index
 2 files changed, 23 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 index.html
```

Observação: mais adiante falaremos sobre commits semânticos e sua importância.

git push: após fazer o commit das alterações, a próxima coisa a fazer é enviar estas alterações ao servidor remoto. **Git push faz o upload dos seus commits no repositório remoto.** Normalmente, basta digitarmos o comando git push, porém como estamos em uma branch nova, criada somente no diretório local, precisaremos usar outro parâmetro junto: **git push --set-upstream origin feat/novo-arquivo**

```
cristina.jung@DBC-001514 MINGW64 ~/repositorios/repoCI-CD (feat/novo-arquivo)
$ git push --set-upstream origin feat/novo-arquivo
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 641 bytes | 641.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'feat/novo-arquivo' on GitHub by visiting:
remote:      https://github.com/cristijung/repoCI-CD/pull/new/feat/novo-arquivo
remote:
To https://github.com/cristijung/repoCI-CD.git
 * [new branch]      feat/novo-arquivo -> feat/novo-arquivo
branch 'feat/novo-arquivo' set up to track 'origin/feat/novo-arquivo'.
```

Neste momento podemos verificar no nosso repositório GitHub que as alterações que fizemos no nosso diretório local já se encontram no remoto, incluindo a branch. Poderemos seguir trabalhando nesta branch por enquanto e mais adiante, veremos como mesclar estas alterações na branch main.



git pull: o comando git pull é usado para obter as atualizações de um repositório remoto. Esse comando é uma combinação de git fetch e git merge, o que significa que, quando usamos git pull, ele recebe as atualizações do repositório remoto (git fetch) e aplica imediatamente as alterações mais recentes em seu espaço de trabalho local (git merge).
