

Versionamento de códigos



O Git é um sistema de controle de versão distribuído que permite gerenciar e rastrear alterações em arquivos de código-fonte durante o desenvolvimento de software.

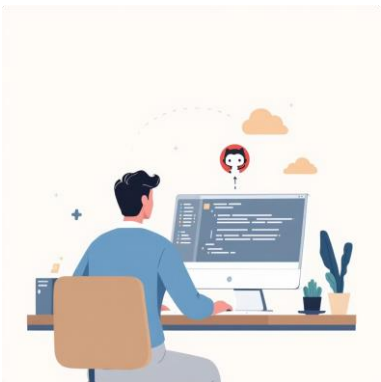
O GitHub é uma plataforma baseada na web que utiliza o Git para hospedagem de código e colaboração



por Romisson Oliveira

Repositório Local vs. Remoto

NOTA - Em nossos estudos vamos utilizar o sistema operacional Windows como base. Instale o Git Bash em sua máquina e faça o cadastro no GitHub. Este manual não irá mostrar como fazer. Serve apenas para um resumo de estudos dos temas abordados.



Um repositório é um **espaço de armazenamento** que contém todos os arquivos, histórico de alterações e versões de um projeto. É basicamente a mesma noção que temos de um servidor disponível para guardarmos determinada informação.

De forma simplificada, o **repositório local** fica armazenado na máquina do desenvolvedor, enquanto o **repositório remoto** é hospedado em um servidor na nuvem, **como o GitHub**. Essa distinção é essencial para entender a dinâmica do versionamento e colaboração em projetos de software.

O repositório remoto, como o GitHub, facilita a colaboração entre membros da equipe, permitindo que várias pessoas contribuam para um projeto simultaneamente. Ao hospedar o repositório em um servidor remoto, os desenvolvedores têm um ponto centralizado para compartilhar e acessar as versões mais recentes do código. Isso torna mais fácil coordenar o trabalho e manter a integridade do projeto em um ambiente de colaboração.

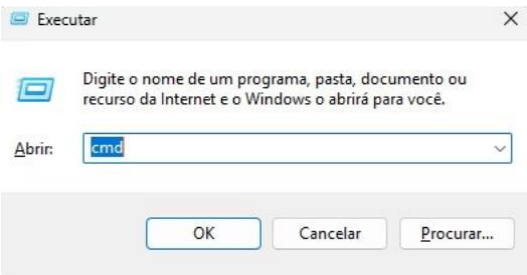
Git Bash

O **Git Bash** é um terminal que permite interagir com o Git em sistemas operacionais como o Windows. Ele oferece um ambiente que simula um terminal Unix, facilitando o uso de comandos para gerenciar repositórios de código.

Ele é uma ferramenta essencial para desenvolvedores, facilitando o versionamento e a colaboração em projetos.

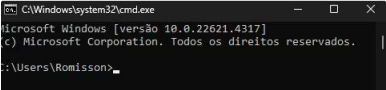
Verificando a versão do Git Bash

Abra o menu executar através do atalho Win+R. E digite o comando 'cmd':

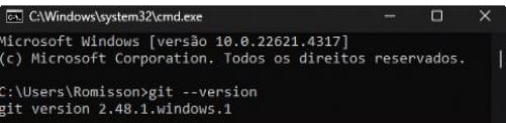


Clique em 'OK'.

Uma tela de prompt de comando será aberta:



Digite a seguinte comando: **git --version** e aperte a tecla 'Enter'.



Assim, será exibida a **versão atual do Git** instalado em sua máquina. Mantenha sempre seu software atualizado.

Abrindo Git Bash

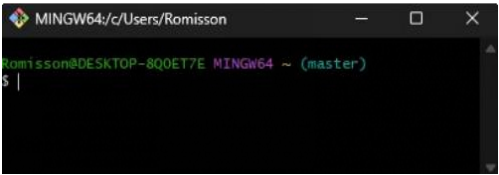
Você poderá abrir o Git Bash direto na pasta que deseja criar um versionamento. E também poderá navegar entre as páginas no próprio prompt do Git.

Isso quer dizer que o **Git Bash é baseado em Unix** significa que ele adota características e funcionalidades de sistemas operacionais da família Unix. Isso inclui comandos, ferramentas e comportamentos típicos encontrados em ambientes como Linux e macOS, como no nosso exemplo a navegação entre diretórios. Aqui estão alguns aspectos importantes:

- Interface e Comandos:** O Git Bash oferece um ambiente de terminal que reconhece comandos comuns de sistemas Unix, como **ls**, **cd**, e **touch**. Esses comandos são familiares para desenvolvedores que usam Linux ou macOS.
- Estrutura de Arquivos:** O comportamento relacionado a diretórios e arquivos segue o padrão Unix, como caminhos de arquivos usando barras (**/**) em vez de barras invertidas (****), típicas do Windows.
- Ferramentas Integradas:** Ele inclui ferramentas como Bash, SSH e outros utilitários usados em ambientes Unix, tornando a interação com Git semelhante à de sistemas baseados em Linux.
- Simulação no Windows:** Embora o Windows tenha seu próprio estilo de terminal (CMD ou PowerShell), o Git Bash cria um ambiente que imita o Unix, facilitando o trabalho de desenvolvedores que preferem esses comandos ou que já estão acostumados a usar Git em outros sistemas.

Visão geral

Ao abrir o software do Git Bash, acessando menu iniciar e digitando Git Bash, execute como administrador. A seguinte tela será exibida:



O título do Git Bash indica o seguinte:

- MINGW64:** Refere-se ao ambiente MinGW-w64, que é uma extensão do MinGW (Minimalist GNU for Windows). Ele fornece ferramentas para compilar e executar programas no Windows, simulando um ambiente Unix.
- /c/Users/Nome:** Representa o caminho do diretório atual no sistema de arquivos. No caso, **/c/** indica a unidade C: do Windows, e **Users/Nome** é o diretório onde você está navegando.

Na parte interna do software temos uma linha preenchida pelas informações do sistema:

O texto **Nome@DESKTOP-8QOET7E MINGW64 ~ (master)** tem os componentes principais:

- Nome:** Refere-se ao nome de usuário configurado no Git Bash. Esse nome pode ser definido usando o comando **git config --global user.name "Seu Nome"**. Ele é usado para identificar quem está realizando os commits no repositório. Veremos o que isso significa mais adiante.
- DESKTOP-8QOET7E:** É o nome do computador ou dispositivo em que o Git Bash está sendo executado. Esse nome é atribuído pelo sistema operacional e pode ser alterado nas configurações do Windows.
- MINGW64:** Já foi explicado.
- (master):** Refere-se a a branch que está aberta no momento. Logo mais veremos a explicação detalhada sobre isso.

E na linha onde possui o símbolo **\$**: serve para inserir comandos git.

Comando cd (change directory)

Ele é usado para navegar entre diretórios no sistema de arquivos. Por exemplo:

- cd nome-do-diretório:** Acessa um diretório específico.
- cd ..:** Retorna ao diretório pai.
- cd /:** Vai para o diretório raiz.
- cd -:** Volta ao último diretório acessado.
- cd ../../:** Volta **dois níveis** na hierarquia de diretórios.

É uma ferramenta essencial para trabalhar com arquivos e pastas no terminal.

Comando ls (list)

O comando **ls** é usado para listar os arquivos e diretórios no terminal e é amplamente utilizado em sistemas baseados em Unix, como Linux e Git Bash. Ao inserir este comando no git, ele mostrará todos os arquivos do diretório atual. Pode usar o comando **dir** também, porém **ls** é muito mais organizado e a exibição melhor de visualizar.

Aqui estão algumas funcionalidades principais:

- ls:** Lista os arquivos e diretórios no diretório atual.
- ls -a:** Mostra todos os arquivos, incluindo os ocultos (aqueles que começam com um ponto **.**).
- ls -l:** Exibe uma lista detalhada, incluindo permissões, proprietário, tamanho e data de modificação.
- ls -R:** Lista arquivos e diretórios de forma recursiva, incluindo subdiretórios.
- ls -h:** Mostra os tamanhos dos arquivos em um formato mais legível (como KB, MB).

É uma ferramenta essencial para navegar e gerenciar arquivos no terminal.

Resumindo: Basta navegar até a pasta que deseja criar um repositório local. E após está com este diretório aberto, digite o comando **git init** no Git Bash. Assim será criado um versionamento local com Git na pasta que está no diretório.

Inicializando um repositório Git de forma simples

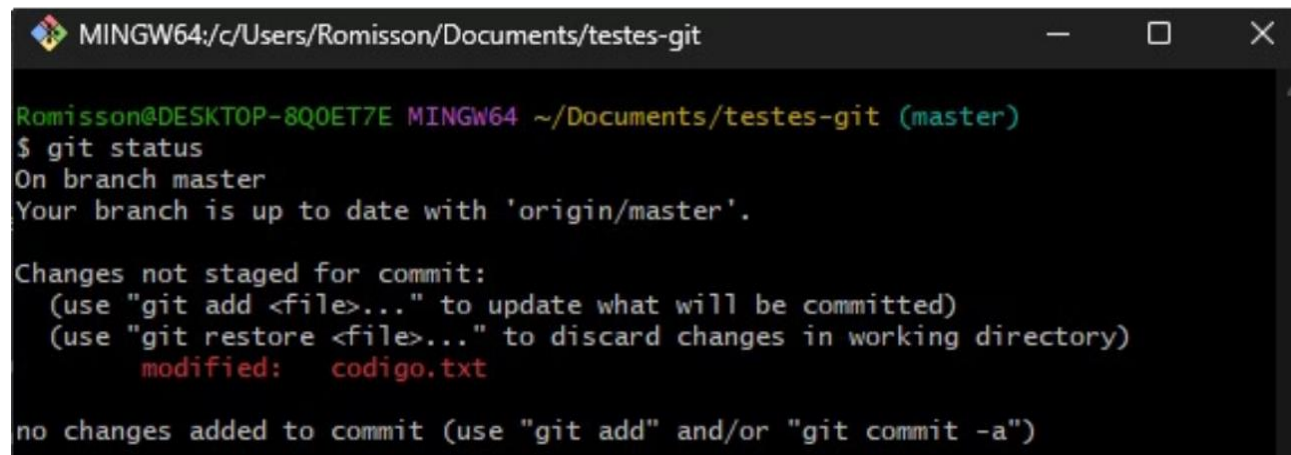
Abra a pasta do projeto. Em algum lugar na pasta clique com botão direito do mouse, "Mostrar mais opções" e clique em "Open Git Bash Here".

Dentro do cmd do git digite o seguinte comando: `git init` . Um repositório local foi criado na pasta.

Para visualizar se um git foi criado na sua pasta, clique no menu "Visualizar", "Mostrar", "Itens escondidos". Assim, você verá uma pasta ".git" indicando que um versionamento criada na sua pasta "master" (principal).

\$ git status

Comando git para visualizar o status atual do versionamento. Assim saberá se há alteração em algum arquivo da pasta que foi versionada. Geralmente aparece de vermelho os arquivos que não foi adicionados ao versionamento.

A screenshot of a Windows terminal window titled 'MINGW64:/c/Users/Romisson/Documents/testes-git'. The terminal shows the output of the 'git status' command. The output indicates that the user is on the 'master' branch and that the branch is up to date with 'origin/master'. It then lists 'Changes not staged for commit', showing that the file 'codigo.txt' has been modified but not yet staged. The terminal text is as follows:

```
Romisson@DESKTOP-8Q0ET7E MINGW64 ~/Documents/testes-git (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   codigo.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Nas informações deste comando, você verá em vermelho os arquivos que foram modificados porém não foram adicionados ao controle de versão local.

\$ git add

Com este comando você poderá adicionar o arquivo em questão para que ele faça parte do seu versionamento do git.

- `$ git add .` : Com este comando você poderá adicionar todos os arquivos que você tem. Não precisa necessariamente indicar qual arquivo deseja. Com este comando fará com que todos os que não foram adicionados sejam incluídos no seu projeto.
- `$ git add <nome-do-arquivo>` : No exemplo anterior este bloco seria escrito assim: `$ git add codigo.txt`. Isso faria ele adicionar o item "codigo.txt" ao controle de versão.

Depois de adicionado, gere um status para saber se foi executado corretamente. Se na exibição de `git status` indicar que *"No commits yet"* significa que nenhum commit foi criado. Ou se você acessar aquele arquivo no `git status` e que ele estiver na cor verde, significa que ele foi adicionado porém ainda não foi feito commit com ele. Para isso, você precisa gerar um commit para que ele incluía no seu projeto.

\$ git commit

O comando **git commit** é usado para registrar mudanças no repositório Git. Ele cria um **ponto de salvamento** no histórico do projeto, permitindo que você volte a versões anteriores se necessário. É importante que antes de gerar um commit, você adicione as alterações ao repositório local.

- `git commit -m "<mensagem-do-commit>"` -> É o comando para gerar um commit para que seja adicionado ao seu projeto. O comando `-m`, serve para configurar uma mensagem ao seu commit. Use isso para como boa prática, até para conseguir compreender o que foi feito no momento da criação do seu commit.
- `git commit -a -m "<mensagem-do-commit>"` -> Desta forma fazemos uma adição dos arquivos sem precisar fazer `"git add ."`. Assim, você adiciona todos as mudanças e faz o commit com o nome da versão que você especificar.



Modificar arquivos

Faça alterações nos seus arquivos de projeto



git add

Adicione os arquivos à área de staging



git commit

Crie um ponto de salvamento com uma mensagem descritiva

Configuração de Usuários e Conexão com GitHub

Usuários Locais x Usuários Globais

A diferença principal é que o **usuário global define configurações padrão para todo o sistema**, enquanto o **usuário local permite personalização por projeto**, sem impactar outros repositórios.

Usuário global

Configurar:

- `git config --global user.name "SeuNome"` : Serve para cadastrar o nome do usuário global.
- `git config --global user.email "seu@email.com"` -> Serve para cadastrar o email do usuário global

Usuário local

Configurando:

- `git config user.name "SeuNome"` : Serve para cadastrar o nome do usuário global.
- `git config user.email "seu@email.com"` -> Serve para cadastrar o email do usuário global

`$ git config --global --list` : Serve para listar os usuários cadastrados.

\$ git push

O comando **git push** é usado para **enviar alterações do repositório local para um repositório remoto**, como o GitHub. Ele permite que outros colaboradores acessem as atualizações feitas no projeto.

Ao inicializar a primeira vez, o git bash vai pedir para configurar o repositório remoto (GitHub):

```
$ git push fatal: No configured push destination. Either specify the URL from the command-line or configure a remote repository using git remote add and then push using the remote name git push
```

Para isso, vá em GitHub e abra sua conta lá. Se você não tiver um repositório, crie um clicando em **+** (no menu superior) e **create new repository**. Lá você irá configurar seu novo repositório com um título e um url será gerado. Escolha se seu repositório será **Público**, onde outras pessoas poderão ver, modificar, baixar seu projeto. Ou se será **Particular**. Importante marcar a opção de criar um **Readme**, pois este é um arquivo onde você era resumir o que é o seu projeto.

Após criar se repositório remoto no GitHub, basta copiar o URL e ir em git bash digitar: `git remote add origin <nome> <URL>`, ou apenas `git remote add origin <URL>`.

Segue URL do meu repositório: [Romisson/testes-git](https://github.com/Romisson/testes-git)

Agora que foi definido o repositório remoto do projeto, use o comando `git push`.

Óbvio que ele não irá executar corretamente. Afinal, não definimos uma **branch** para onde nosso projeto será carregado. Já iremos fazer isso agora, mas antes, veja a mensagem que ele exibiu ao digitar o comando `git push`:

```
$ git push fatal: The current branch master has no upstream branch. To push the current branch and set the remote as upstream, use
```

```
git push --set-upstream origin master
```

```
To have this happen automatically for branches without a tracking upstream, see 'push.autoSetupRemote' in 'git help config'.
```

Agora, se você quer que a branch onde será enviada seja a **branch master**, basta digitar o código que o erro acima sugeriu: `git push --set-upstream origin master`.

Agora todos os commit criados localmente será carregado ao repositório remoto. Já configurado o repositório remoto, os próximos "push" que precisar apenas digite `git push`.

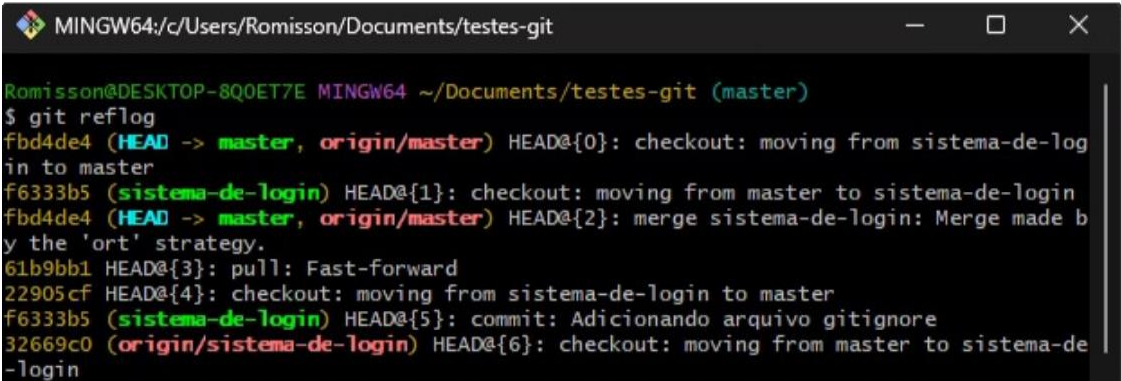
Histórico e Navegação entre Versões

Para verificar o histórico de atualização e ver as versões criadas, use o seguinte comando:

```
$ git reflog
```

Este comando é uma ferramenta poderosa no Git que permite rastrear todas as alterações feitas no ponteiro **HEAD** (ponteiro especial que indica o commit atual em que está trabalhando) do repositório local. Ele registra ações como commits, mudanças de branch, merges, resets e rebases, mesmo que essas ações não estejam mais visíveis no histórico normal de commits (**git log**). Isso é especialmente útil para recuperar commits "perdidos" ou desfazer alterações recentes.

Por exemplo, se você acidentalmente redefinir o repositório para um estado anterior, o git reflog pode ajudar a identificar o commit que você deseja restaurar. Cada entrada no reflog é marcada com um índice, como HEAD@{0}, que representa a ação mais recente, e assim por diante.



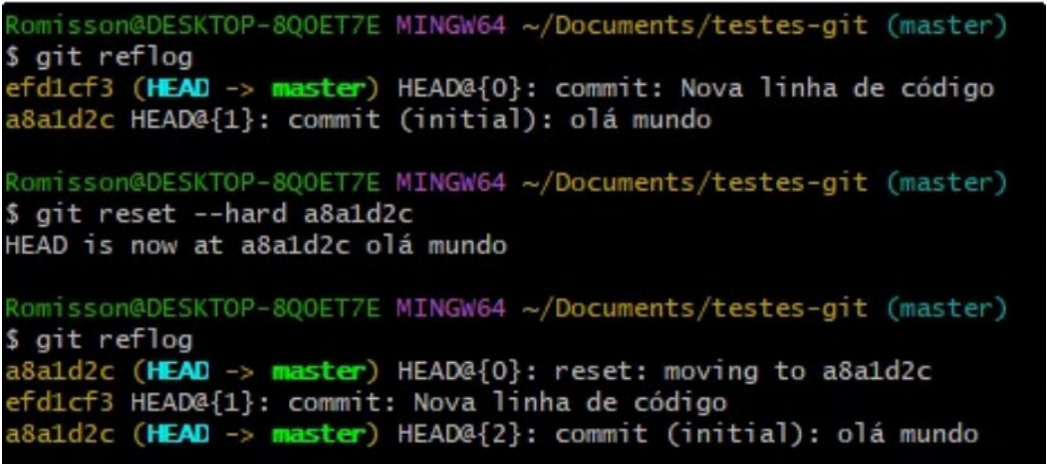
Desta forma, irá ser disponibilizado um ID da sua versão. E conterà algumas informações inclusive o nome da commit que você adicionou no momento da criação.

Através deste id, você poderá navegar entre versões com o comando: "git reset --hard <id>"

No caso acima, se quer que volte para a versão do HEAD@{4}, que possui o id: 22905cf, insira o seguinte comando git: "git reset --hard 22905cf". Assim, você irá para versão onde foi movido da branch "sistema-de-login" para branch "master".

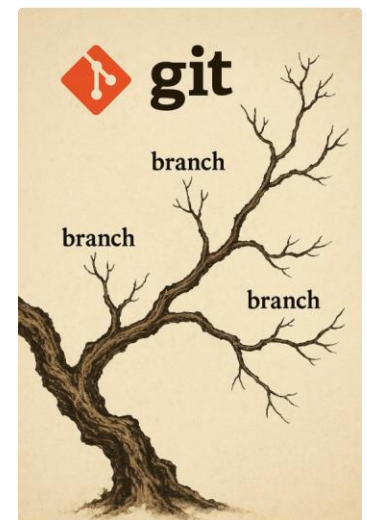
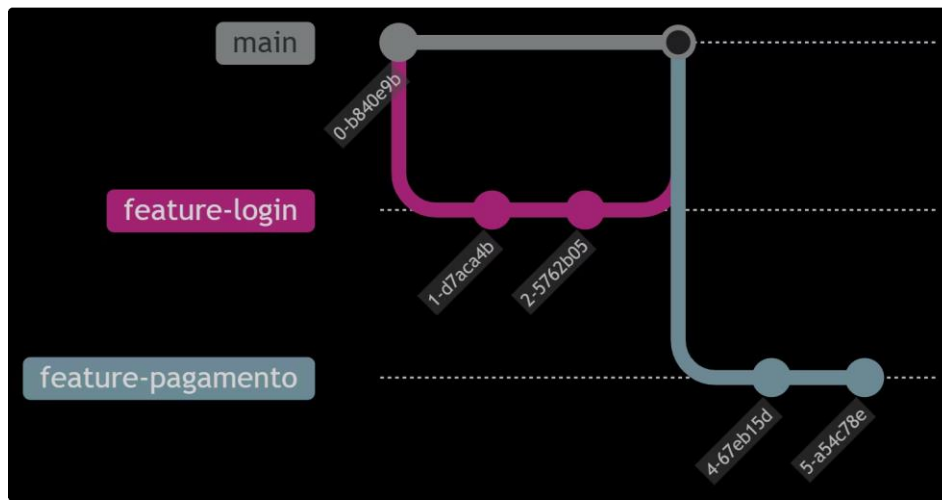
Exemplo:

- Criei uma pasta chamada **testes-git**
- Abri o Git Bash na pasta. Ou poderia apenas abrir o Git Bash e navegar até a pasta através do comando "**cd**". Assim, poderá, por exemplo, usar o comando "**cd Documents/testes-git**".
- Após isso, usei o comando "**git init**"
- Dentro da pasta, criei o arquivo txt: **codigo.txt**
- Dentro deste arquivo, inseri a linha "ola mundo".
- Depois adicionei ao controle de versão com "**git add codigo.txt**"
- Depois feito um commit com o nome de "olá mundo"**git commit -m "olá mundo"**
- Inseri uma modificação ao codigo.txt, inserindo uma nova linha de código.
- No git bash, digitei: **git status**. Após ver a modificação no git bash, adicionei e inseri o commit logo no início: **git commit -a -m "Nova linha de código"**
- Agora, para visualizar os commit feitos, inseri o código: **git reflog**.



- Depois de visualizar o id de cada versão que eu desejo retornar, usei o comando "**git reset --hard**":
- **git reset --hard a8a1d2c**
- Consegui voltar até a versão "olá mundo".

Trabalhando com Branches



O que são branches e como criar uma?

Branch (galhos) são caminhos diferentes para o versionamento do seu código. São ramificações que permitem trabalhar em diferentes versões do código sem afetar a linha principal de desenvolvimento. Elas são essenciais para colaboração e organização em projetos.

Você pode renomear suas branches porém sempre terá uma branch que você irá considerar um branch master: **principal**.

O recurso de criar branches (ramificações) do seu projeto será necessário quando você estiver trabalhando em equipe. Se será apenas você ao desenvolver seu projeto, não complique sua vida criando branches.

Suponha que você tem uma branch principal chamada "master". Nela, você irá inserir apenas códigos do projeto que funcionam. Enquanto as equipes de desenvolvimento necessitam desenvolver novas funcionalidades, todo o projeto não pode parar. Assim, você cria uma nova branch (**staging**) e nela a equipe cria e testa novas funcionalidades.

Imagine que em um determinado momento uma branch foi idealizada e aceita para fazer parte da branch principal. Agora vamos utilizar um recurso chamado "Merge". Vamos "mergiar" duas branches a uma principal (mesclar).

Visualizando as branches disponíveis:

```
$ git branch
```

Este comando fará você ver quais existem.

Para criar uma nova branch: use o comando "`git branch <nome-da-branch>`".

Em nosso exemplo: `git branch staging`. ***staging** - é o nome popular usado para se referir a um branch que ainda não esta pronta para ser usada na branch principal.

Após criado uma branch, use o comando "`git branch`" para visualizar novamente. Agora que possuímos duas branches no projeto, observe que a branch atual aparecerá com um asteriscos (*) e terá a **coloração de verde**, indicando que é a branch selecionada.

```
$ git checkout <nome-da-branch>
```

Nosso exemplo: `git checkout staging`.

Se for feito corretamente aparecerá a mensagem: "**Switched to branch 'staging'**" E inclusive no diretório, você poderá ver entre parênteses a branch selecionada, conforme explicado no início deste estudo.

Agora que você tem uma branch igual a principal, poderá fazer alterações no seu projeto e mesmo que adicione a versão e não será armazenado a branch principal.

Merge e Pull Request

Merge

É o processo de unir as alterações de uma branch secundária à branch principal. Em outras palavras, significa trazer as alterações de outras branches para a principal.

Para isso, siga essa sequência de passos:

- Vá até a branch que receberá as atualizações (**master**, no nosso caso);
- Use o comando `git merge <nome-da-branch-que-tem-alteração>` (aquele que tem as alterações): exemplo: `git merge staging`
- Feito a mescla, basta dar um `git push` para que a master no repositório remoto receba a alteração.

Atenção - Ao entrar na branch principal, antes de gerar um merge, você precisa garantir que as **alterações serão as mais atualizadas possíveis**. Para isso, use o comando: `git pull ->` Este comando fará com que todas as atualizações sejam carregadas. Depois de gerar um pull, faça o merge normalmente.

- `git pull` da branch principal;
- gerar uma nova branch a partir da branch principal (se já criada, basta navegar até ela);
- trabalhar e adicionar novas funcionalidades da nova branch que criou;
- finalizar o trabalho na branch temporária, criando o versionamento;
- `git pull`, para garantir que seja o mais atualizado possível;
- `git checkout` para branch principal, ou seja, navegar para a branch que receberá atualização;
- `git merge` (unir) o código da branch temporária com a branch principal (depois de testar, é claro);
- `git push` da branch principal.

Exemplo: Imagine que você recebeu a função de criar um sistema de login no projeto. Com a branch principal do projeto aberta no git bash, crie uma branch para desenvolver o projeto de login. Para fazer tudo isso de forma direta em uma linha de código, use o comando: `git checkout -b <nome-da-nova-branch> <nome-da-branch-de-origem>`

```
$ git checkout -b sistema-de-login master
```

Mensagem padrão do comando: *Switched to a new branch 'sistema-de-login'*. Significa que você foi direcionado para nova branch e já criou ela. **Sem este comando**, você teria que:

- Criar uma branch: `git branch sistema-de-login`;
- Navegar até a nova branch: `git checkout sistema-de-login`.

Podemos visualizar a branch, com: `git branch`

Agora que já está na nova branch, crie a funcionalidade de login. Feito as alterações e os testes e definido que é a hora de inserir na branch principal, adicione e crie um commit.

```
git commit -a -m "Criado sistema de login"
```

Agora que a nova versão já foi adicionada e comitada, vá até a branch que receberá esta funcionalidade, master no nosso caso: `git checkout master`

Agora, gere um `git pull` atualizando sua branch.

Insira o comando: `git merge sistema-de-login`

Por fim, envie a branch master para o repositório remoto com `git push`.

No caso acima, foi feito a alteração na branch principal sem mesmo nenhuma verificação.

Agora em **sistemas reais**, será necessário fazer um recurso chamado '**pull request**'.

Pull Request - Merge democrático

Aprovar ou reprovar a modificação em um branch. Um pull request é uma solicitação para integrar alterações feitas em uma branch específica a outra branch, geralmente a principal, em um repositório remoto. Ele é amplamente utilizado em plataformas como GitHub para facilitar a colaboração em equipe.

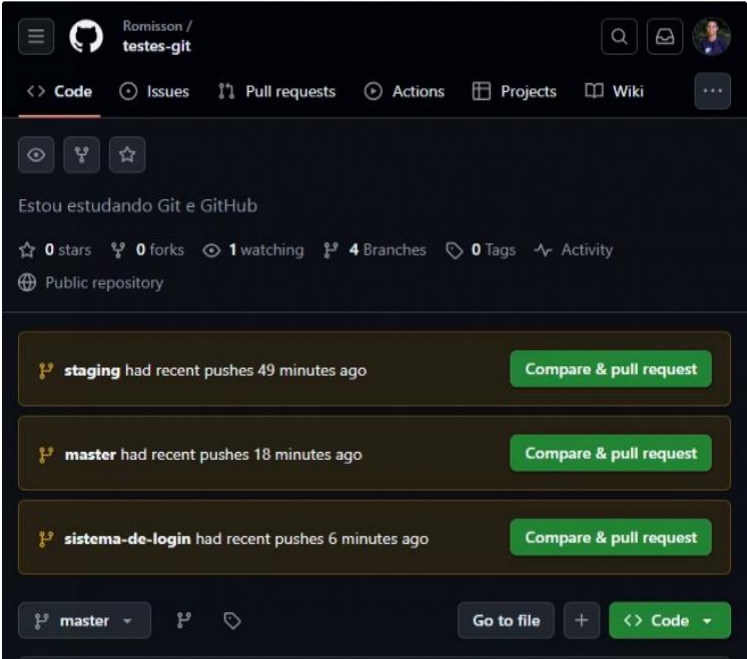
Imagine, que você faz parte de uma equipe de desenvolvedores. A partir daí você ficará responsável por gerar uma nova funcionalidade no sistema. Sendo que existe uma hierarquia na empresa e para que sua modificação faça parte da branch principal do projeto, precisará passar pela aprovação ou reprovação de um desenvolvedor competente. Assim, após terminar sua parte, você fará uma "requisição" para que possa fazer parte da ramificação principal. Você utilizará um '**pull request**' para isso.

Vamos testar, vá até a branch que você criou para sistema de login, faça uma alteração no código, adicione a nova versão do código e gere um commit.

Vamos enviar esta branch para o repositório remoto com `git push`. Se for a primeira vez, ele precisará um caminho `git push --set-upstream sistema-de-login`. Se não funcionar, basta digitar `git push`. Assim ele pedirá para configurar e exibirá o código acima, selecione, copie (**ctrl+ins**) e cole na linha de comando.

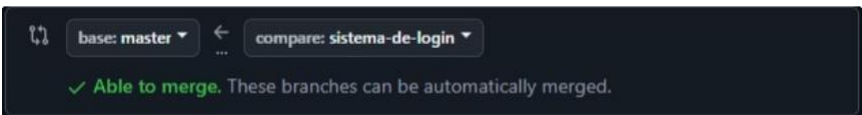
A partir de agora, este tutorial serve apenas para GitHub - em outras plataformas de versionamento remoto, basta procurar por tutoriais:

Vá na branch principal no repositório do GitHub. E você verá a página indicando que existem '**Compare & pull request**'.



Clique em "**Compare & pull request**" do sistema-de-login.

Agora precisamos especificar que queremos fazer um merge com a base da master comparando com o sistema-de-login, que é onde fizemos as alterações que desejamos mergiar para dentro da base:



Depois preencha as informações para explicar o que foi feito para que outro consulte e valide suas alterações. Depois -> '**create pull request**'.

Como nossa equipe não possui mais pessoas para aprovar/reprovar nosso código, então você verá logo de cara um '**Merge pull request**'. Porém se houver uma pessoa para isso, você verá na aba lateral em '**reviewers**'. Ali estará um, dois ou mais responsáveis para mergiar a branch à principal.

Se caso seu commit for **reprovado** pelo '**reviewer**', você verá uma marca vermelha e o comentário explicando o porquê de seu código ter sido reprovado e geralmente acompanha um pedido de alteração. Assim, basta você fazer as alterações e gerar um novo commit, não precisa gerar um pull request novo, apenas um novo commit será suficiente para que a parte competente veja sua alteração. Basta comentar neste commit marcando as pessoas que são responsáveis para que vejam seu novo commit.

Git Ignore

O `.gitignore` é um arquivo de configuração do Git que especifica quais arquivos ou pastas não devem ser rastreados pelo versionamento. Ele é útil para evitar que arquivos temporários, dependências ou informações sensíveis (como senhas e chaves de API) sejam commitados acidentalmente no repositório.

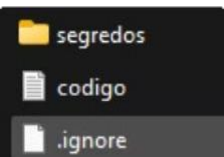
Em outras palavras, arquivos que você não deseja que seja adicionados ao versionamento deverá ser especificados com este recurso.

Exemplo: Imagine que eu criei uma pasta chamada `segredos` e tenho um arquivo chamado `senhas.txt` e dentro dela tenha códigos sensíveis que não desejo que outros usuários vejam. Então esta pasta `segredos` não deve de jeito nenhum subir com o controle de versão. Ela, portanto, deve ser ignorada (*ignore*, em inglês). Como faremos para que isso não seja adicionado no nosso código? (Dentro da branch `sistema-de-login`), dou um `git pull`, e vejo com `git status`. Veremos que a pasta `segredos/` está *untracked* (ainda não foi adicionado ao controle de versão) e que precisa ser adicionado à branch. E não podemos simplesmente usar comando `git add segredos/` e adicionar ao versionamento, pois queremos que tal pasta seja ignorada. E se usarmos este comando de adicionar ou até mesmo `git add .`, iremos adicionar a pasta ao controle de versão e isso que não queremos.

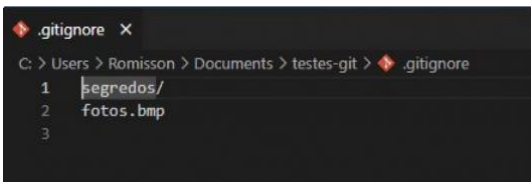
Para ignorar esta pasta, **APENAS** no terminal Git Bash, você conseguirá fazer isso: `touch .gitignore`

Assim, na pasta onde está repositório remoto raiz, ele criará um arquivo `.ignore`.

Clique no arquivo `.ignore` e ele abrirá um bloco de nota.



E dentro da nota, você irá adicionar os itens que deseja que seja ignorado: `segredos/` `fotos.bmp`



Atenção: Ao utilizar o comando de criação do `.gitignore`, eu usei `touch .ignore`, sem querer: e assim foi criado um arquivo `ignore` erroneamente. Para corrigir via comando git, usei o seguinte código: `ren .ignore .gitignore` S O comando `ren` no Windows é usado para **renomear arquivos ou pastas** diretamente no prompt de comando. Ele é uma abreviação de "rename" e permite alterar nomes de forma rápida e eficiente.

Podemos também fazer com que todos os arquivos de um determinado tipo seja ignorado. Imagine que você tenha imagens `.png` e que não quer que nenhuma imagem seja carregado para o repositório remoto, dentro do arquivo `.ignore`, digite: `*.png` S Assim, todos os arquivos deste formato serão ignorados.

Se os arquivos já foram rastreados antes de serem inseridos no `.gitignore`, basta:

- `git rm --cached segredos/ -r` # Remove a pasta do tracking
- `git rm --cached *.bmp` # Remove os arquivos .bmp
- `git add .gitignore` # Adiciona o .gitignore
- `git commit -m "Corrige .gitignore"`

Voltando, agora que seu `.gitignore` foi criados e dentro dele foi inserido o que você deseja que seja ignorado, você precisa adicioná-lo ao controle de versão, use o `git add .`, gere um commit e envie ao repositório remoto.

Resumo - Git e GitHub

1	<code>git init</code> Inicializa um novo repositório	2	<code>git add .</code> Adiciona os arquivos atuais ao próximo commit	3	<code>git status</code> Verificar o status atual do repositório git
4	<code>git commit</code> 4.1. <code>git commit -m "mensagem-do-commit"</code> S Cria um novo commit com um mensagem 4.2. <code>git commit -a -m "mensagem-do-commit"</code> S Adiciona o arquivo e faz o commit já com a mensagem.	5	<code>git push</code> Envia as atualizações para a nuvem na branch atualmente ativa.	G	<code>git branch</code> Permite listar e ver qual branch está ativa atualmente
7	<code>git branch nome-da-nova-branch</code> Permite criar uma nova branch	8	<code>git checkout nome-da-branch</code> Permite mudar para nova branch	9	<code>git checkout -b "nome da branch de origem" "nome da nova branch"</code> Permite mudar e criar um nova branch com base na outra.
10	<code>git merge "branch a ser mergiada"</code> Permite fazer a mescla da branch ativa atualmente com outra branch citada no comando.	11	<code>git pull</code> Atualiza a branch atualmente ativa	12	<code>clear</code> Permite limpar a tela do git bash para melhor visualização.
13	<code>cd</code> Serve para navegar, encontrar e abrir novas pastas	14	<code>exit</code> Sair do Git Bash	15	<code>git config -global --list</code> Permite visualizar todos os usuários cadastrados no repositório local.
1G	<code>git config --global user.name "SeuNome"</code> Serve para configurar um usuário global	17	<code>git config --global user.email "seu@email.com"</code> Serve para configurar um email global.	18	<code>git remote add origin <url></code> Serve para configurar o repositório remoto (GitHub), adicione a URL (HTTPS ou SSH)
19	<code>git push</code> No primeiro push: <code>git push --set-upstream origin master</code> # Ou (versões mais novas): <code>\$ git push -u origin master</code> Push subsequentes: <code>\$ git push</code>	20	<code>git reflog</code> Mostra TODAS as ações no repositórios (incluindo as "perdidas")	21	<code>git log</code> Mostra o histórico de commit



Considerações Finais: A Importância do Git e GitHub no Desenvolvimento de Sistemas

O **Git** e o **GitHub** revolucionaram a forma como desenvolvemos software, tornando-se ferramentas **indispensáveis** para qualquer profissional da área de tecnologia. Sua importância vai muito além do simples "controle de versão" 4 eles são a base para **colaboração eficiente, rastreabilidade de código e gestão de projetos modernos**.

Dominar **Git** e **GitHub** não é opcional 4 é essencial para:

- Desenvolvedores individuais (backup e histórico do código).
- Equipes (trabalho colaborativo sem caos).
- Empresas (gestão eficiente de projetos complexos).

Quem ignora essas ferramentas fica para trás no mercado de tecnologia.