

Universidade Federal Fluminense - uff
Escola de Engenharia – TCE
Departamento de Engenharia de Telecomunicações - TET
Grupo PET do Curso de Engenharia de Telecomunicações -
PET-Tele

Sistema para armazenamento, compartilhamento e versionamento de software

João Guilherme Beltrão

entregue em:
data limite: 04/04/2021

Objetivo

O objetivo a ser alcançado com esse projeto é mostrar o uso de sistemas para armazenamento, compartilhamento e versionamento de software, em especial o sistema *Git*. Assim como sua história e importância visando o uso em projetos do grupo PET. Além disso, também será estudado ambientes de hospedagem de código como o *GitHub* e o *GitLab*.

Motivações

- Aprender e aprofundar o conhecimento sobre o sistema de versionamento *Git*.
- Implementar o uso de software como o *git* nos projetos do grupo PET para facilitar o versionamento e construção de códigos em grupo.
- Implementa também o uso de ambientes de hospedagem como *GitHub* e *GitLab* para facilitar o compartilhamento e armazenamento de códigos de um projeto.

1 Versionamento

1.1 O que é versionamento?

Versionamento de software é o processo de salvar momentos de um documento para que esses possam ser acessados no futuro caso ocorram erros ou mudanças indesejáveis. Dessa forma o usuário consegue ter acesso a vários estados do projeto caso ocorram problemas com o mais recente.

Contudo, Versionar um projeto manualmente tem muitas desvantagens. Primeiramente, é muito trabalhoso para salvar todos os estados de um código, logo, algumas versões se perderam o que não é o ideal. Outro problema é para trabalhar com mais pessoas, já que cada pessoa estaria salvando seu próprio código podendo conflitar no momento que os juntassem. Para resolver esse problema foram criados os softwares de controle de versão.

1.2 Software de controle de versão

Os softwares de controle de versão (também chamados de SVC) são softwares que auxiliam o programador a fazer o versionamento de seu projeto. Neles cada pequena mudança do código será salva e sem o risco de perder o arquivo além de facilitar todo o processo de armazenamento e compartilhamento dos arquivos. Foram criados dos tipos de software de controle de versão o centralizado e o distribuído

1.2.1 SVC centralizado

Nos SVC centralizados quando o programador vai salvar um estado do código, o SVC mandará o arquivo para um repositório local conectado ao mesmo servidor do computador usado. Dessa forma um grupo de programadores consegue todos salvar os seus arquivos em um único lugar e compará-los, além de ter acesso a todos as versões anteriores desses arquivos. O único defeito desse tipo de versionamento é que para acessar esses arquivos e colocar novos, o programador precisava estar constantemente conectado ao servidor do repositório. Esse defeito foi resolvido com o outro tipo de SVC.

1.2.2 SVC distribuído

Nos SVC distribuído o programador salva o estado do código em um repositório local dentro de seu computador para depois serem colocados em um repositório remoto onde todos os códigos guardados em cada repositório local do grupo de programadores será armazenado e facilmente acessado por todos.

1.2.3 Vantagens de um SVC

- O programador tem total controle e fácil acesso a todas as versões anteriores de seu código e quais são as diferenças de cada versão.
- A equipe de programadores consegue se ramificar e facilmente juntar os códigos no final do projeto.
- Auxilia na organização do projeto como um todo.

2 *Git*

Git é um software de controle de versão distribuído criado em 2005 por Linus Torvalds(criador do sistema operacional Linux) e é atualmente o SVC mais usado no mercado.Ele ficou popular por alguns motivos:

- É Open Source diferente de seu principal concorrente da época (BitKeeper).
- Tinha uma performance muito superior aos outros SVC da época.
- É distribuído, o que é incontestavelmente melhor do que os SVC centralizados.

2.1 Repositório remoto

Antes de começar a usar o *Git* para versionar seus projetos, é preciso entender o que são os repositórios remotos. Como já foi dito, os SVC distribuídos versionam os códigos mandando-os para um repositório local primeiro e depois para um repositório remoto. O *Git* faz o papel do repositório local porém para

o remoto precisamos usar outra plataforma, os principais sendo o *GitHub* e o *GitLab*.

2.1.1 *GitHub* vs *GitLab*

Github e *Gitlab* são plataformas de hospedagem de código-fonte. Elas permitem que os desenvolvedores contribuam em projetos privados ou abertos, e ambas fazem o controle de versão dos projetos hospedados utilizando o *Git*.

A principal diferença entre elas é que o *GitLab* vem dando foco à integração com ferramentas *DevOps* e proporciona, nativamente, ferramentas de integração e entrega contínua, enquanto o *GitHub* foca mais em eficácia e em desempenho de infraestrutura sendo a melhor opção para projetos com grande números de programadores.

2.2 Como usar o *Git*?

O *Git* por ser um software Open Source é totalmente gratuito, então para baixa-lo é só entrar no site *git-scm.com* e fazer o download. Além disso para usá-lo com maior eficácia é importante criar uma conta nas plataformas de hospedagem como o *GitHub*.

2.2.1 Usando *Git* pelo terminal

Com o *Git* instalado podemos começar a usá-lo. Para isso alguns comando do *Git* são necessários.

1. HELP

Esse é o primeiro comando importante que se deve conhecer, com ele o programador é mandado para páginas de tutorial sobre cada um dos outros comandos.

```
git help "comando"
```

2. INIT

Esse comando cria um repositório local no arquivo que estiver sendo acessado no momento.

```
git init
```

3. STATUS

Esse comando mostra se ocorreu alguma alteração no repositório e o que pode se fazer com elas.

```
git status
```

4. ADD

Com esse comando o programador passa uma das modificações mostradas pelo status para a linha de commit.

```
git add "nome do arquivo"
```

5. COMMIT

Esse comando faz o *commit* das mudanças na linha de *commit*, o que significa mandá-los para o repositório local.

git commit -m "comentario sobre a mudança"

6. PUSH

Esse comando manda o conteúdo do repositório local para o repositório remoto, o que depende de qual o programador configurou para conectar ao *Git* (essa configuração é facilmente achada nos sites das plataformas como *GitHub* e *GitLab*).

git push

7. REMOTE

Esse comando configura o repositório remoto, é com ele que o programador diz onde os códigos serão armazenados no final.

git remote add origin "link do repositório"

8. BRANCH

Com esse comando o programador consegue criar uma *branch* (partes de um repositório onde podem se guardar códigos separadamente e que podem ou não serem juntos na *branch* principal) pelo *Git*.

git branch "nome da branch"

9. CHECKOUT

Com esse comando o programador consegue mudar a *branch* para onde os códigos serão mandados.

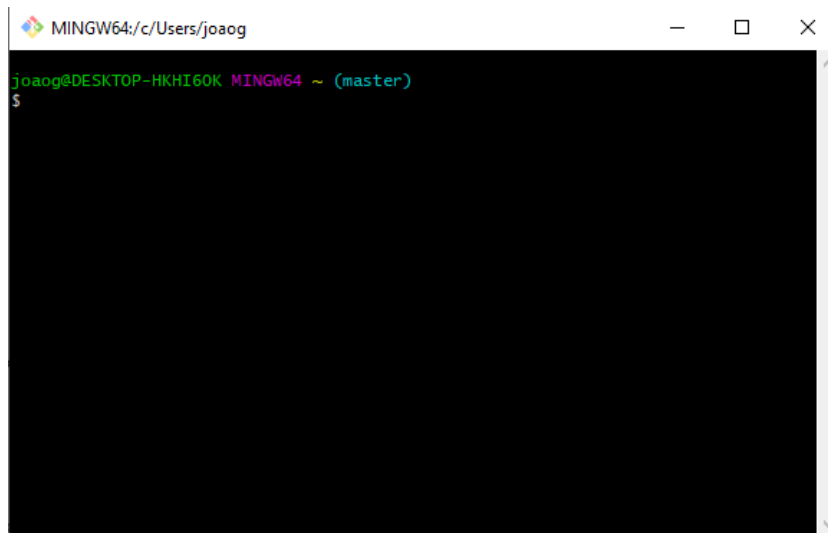
git checkout "nome da branch"

Com esses comandos em mente já temos tudo para usar o *Git*. Para começar, precisamos abrir o programa Git Bash que foi baixado junto do *Git*.



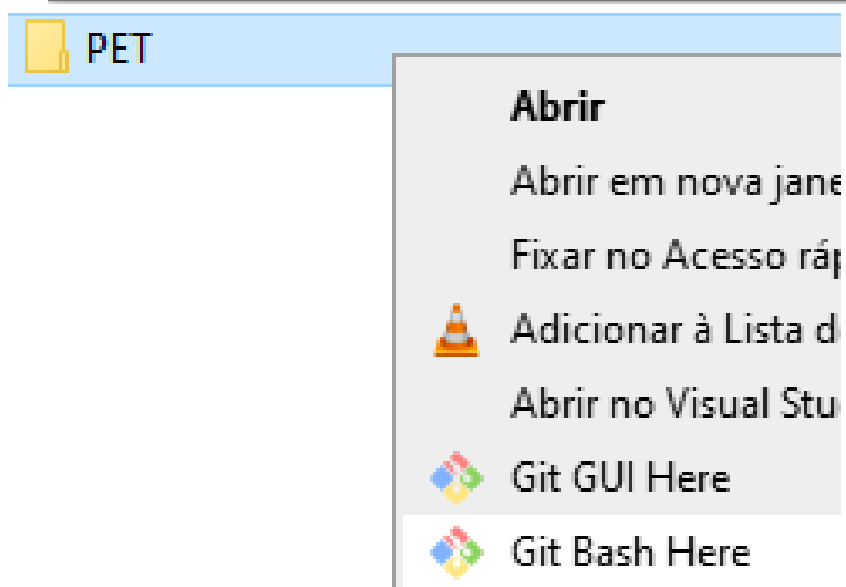
Git Bash

Aplicativo

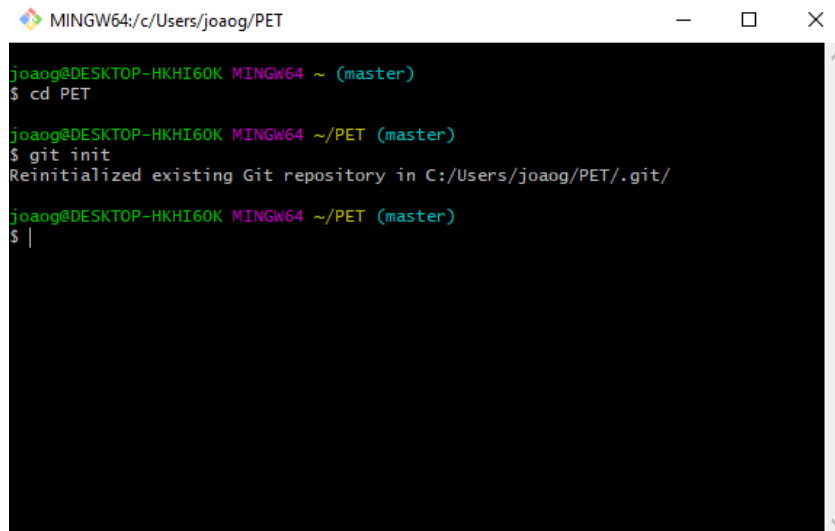


Agora precisamos entrar na pasta onde será o repositório local, para isso usamos o comando do terminal `cd "nome da pasta"` para ir entrando nas pastas até chegar no local onde será o repositório. Outra forma possível é ir ao local da pasta manualmente e clicar em abrir com *Git Bash*.

```
MINGW64/c/Users/joaog/PET
joaog@DESKTOP-HKHI60K MINGW64 ~ (master)
$ cd PET
joaog@DESKTOP-HKHI60K MINGW64 ~/PET (master)
$ |
```



Acessando o arquivo escolhido agora devemos iniciar o repositório local usando o comando *git init*.

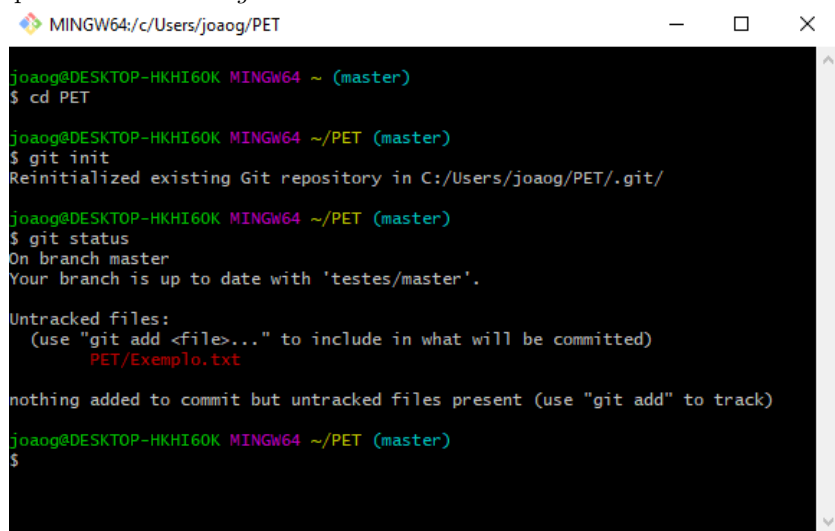


```
MINGW64/c/Users/joaog/PET
joaog@DESKTOP-HKHI60K MINGW64 ~ (master)
$ cd PET

joaog@DESKTOP-HKHI60K MINGW64 ~/PET (master)
$ git init
Reinitialized existing Git repository in C:/Users/joaog/PET/.git/

joaog@DESKTOP-HKHI60K MINGW64 ~/PET (master)
$ |
```

Agora podemos checar por mudanças na pasta que não foram enviadas para o repositório usando o *git status*.



```
MINGW64/c/Users/joaog/PET
joaog@DESKTOP-HKHI60K MINGW64 ~ (master)
$ cd PET

joaog@DESKTOP-HKHI60K MINGW64 ~/PET (master)
$ git init
Reinitialized existing Git repository in C:/Users/joaog/PET/.git/

joaog@DESKTOP-HKHI60K MINGW64 ~/PET (master)
$ git status
On branch master
Your branch is up to date with 'testes/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    PET/Exemplo.txt

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

joaog@DESKTOP-HKHI60K MINGW64 ~/PET (master)
$
```

Conseguindo ver o status da pasta temos que adicionar os arquivos modificados na linha para o *commit*. Temos duas formas de fazer o comando que é, colocando todas as mudanças da pasta de uma vez com o *git add "pasta"* ou colocando um arquivo por vez com o *git add "arquivo"*.


```
MINGW64/c/Users/joaog/PET

joaog@DESKTOP-HKHI60K MINGW64 ~/PET (master)
$ git status
On branch master
Your branch is up to date with 'testes/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  PET/Exemplo.txt

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

joaog@DESKTOP-HKHI60K MINGW64 ~/PET (master)
$ git add PET

joaog@DESKTOP-HKHI60K MINGW64 ~/PET (master)
$ git status
On branch master
Your branch is up to date with 'testes/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   PET/Exemplo.txt
```

Depois de adicioná-los já podemos passá-los para o repositório local usando o comando *git commit*. No caso desse comando precisamos mandar uma mensagem junto ao *commit*, para isso podemos usar o comando *git commit -m "mensagem a enviar"* ou colocar somente *git commit*, o que te levará a outra tela onde deverá ser posta a mensagem e depois para sair apertar "esc" e digitar ":wq".

```
MINGW64/c/Users/joaog/PET

On branch master
Your branch is up to date with 'testes/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   PET/Exemplo.txt

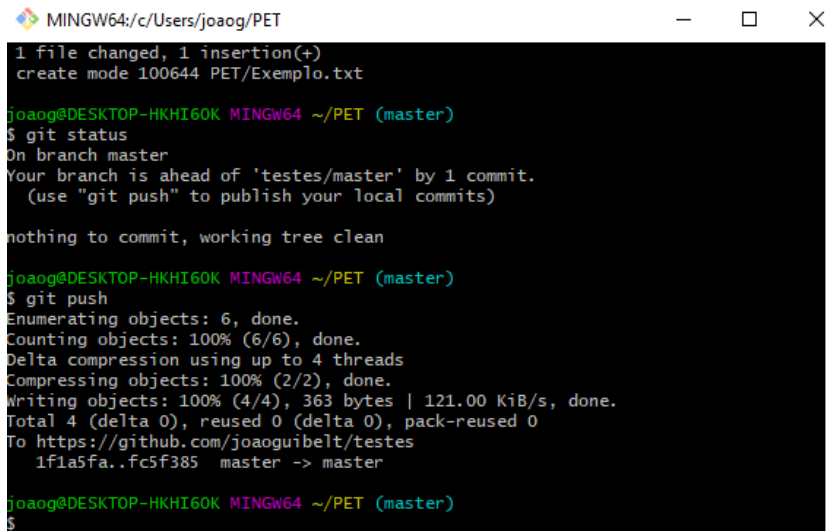
joaog@DESKTOP-HKHI60K MINGW64 ~/PET (master)
$ git commit -m Exemplo
[master fc5f385] Exemplo
1 file changed, 1 insertion(+)
create mode 100644 PET/Exemplo.txt

joaog@DESKTOP-HKHI60K MINGW64 ~/PET (master)
$ git status
On branch master
Your branch is ahead of 'testes/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean

joaog@DESKTOP-HKHI60K MINGW64 ~/PET (master)
$ |
```

O próximo passo é mandar as mudanças para o repositório remoto, para isso o seu *Git* precisa estar conectado e configurado a algum repositório nas plataformas de hospedagem(ver a subseção 2.3 e 2.4). Depois de estar conectado é só usar o comando *git push -u origin "nome da branch que receberá os códigos"* para fazer o primeiro *git push*, as próximas vezes que for usar o *git push* é só escrever o comando *git push* que ele mandará para a última *branch* conectada.

A terminal window titled 'MINGW64/c/Users/joaog/PET' with standard window controls. The terminal shows the following commands and output:

```
1 file changed, 1 insertion(+)
create mode 100644 PET/Exemplo.txt

joaog@DESKTOP-HKHI60K MINGW64 ~/PET (master)
$ git status
On branch master
Your branch is ahead of 'testes/master' by 1 commit.
(use "git push" to publish your local commits)

nothing to commit, working tree clean

joaog@DESKTOP-HKHI60K MINGW64 ~/PET (master)
$ git push
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 363 bytes | 121.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/joaoguibelto/testes
 1f1a5fa..fc5f385 master -> master

joaog@DESKTOP-HKHI60K MINGW64 ~/PET (master)
$
```

E pronto todas as mudanças feitas foram guardadas no repositório remoto e versionadas podendo ser acessadas a qualquer momento.

2.2.2 Usando o *Git* por uma interface gráfica

Além do terminal o *Git* tem compatibilidade com uma variedade de interfaces gráficas como o *sourcetree* e o *GitKraken*, e até algumas interfaces específicas para certas plataformas de hospedagem como o *GitHubDesktop*. Nelas o programador consegue de forma mais simples fazer os "commits" e visualizar mais facilmente o que o *Git* está fazendo, porém cada software desses funciona de uma forma diferente e tem suas vantagens e desvantagens que devem ser estudadas para o próprio programa.

2.3 Conectando *Git* com o *GitHub*

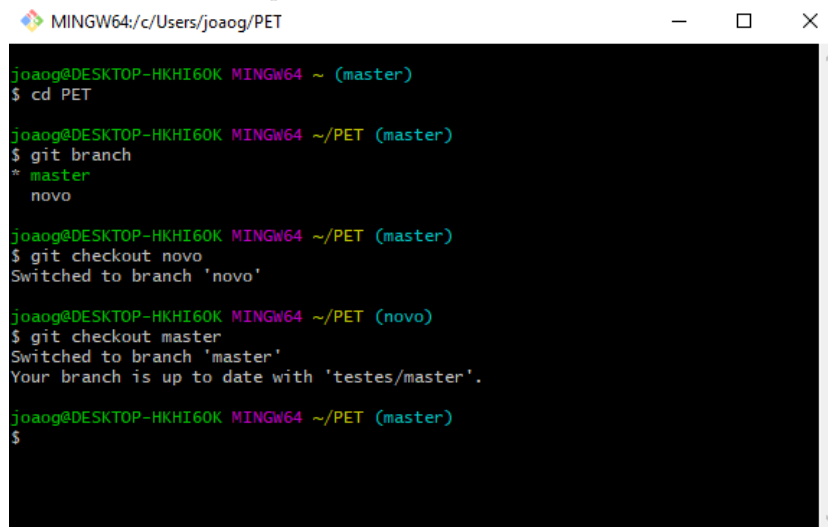
Para que o programador consiga colocar os seus códigos no *GitHub* pelo terminal ele precisa conectar o *Git* ao seu repositório, para isso a forma mais simples é usando uma chave ssh (Tutorial completo e simples sobre a configuração de uma chave ssh no site do *GitHub*).

Com a chave configurada o programador precisa usar o comando *git remote add origin "link do repositório"* para ligar o repositório ao *Git* com o nome de *origin* e assim ele terá total controle sobre o repositório.

2.4 trocando de *branch*

Outros comandos bem importantes são os que mexem as *branches*. Para criar uma nova *branch* pelo *Git* é só usar o comando *git branch "nome da branch"* e para checar as *branches* ativas usar o comando *git branch*. Quando as *branches* forem cheçadas aparecerá um asterisco do lado da que estiver sincronizada para

receber os códigos, para trocar isso deve-se usar o comando *git checkout "nome da branch a sincronizar"* que assim quando o programador fizer o *git push* os *commits* serão mandados para essa branch.



```
MINGW64/c/Users/joaog/PET

joaog@DESKTOP-HKHI60K MINGW64 ~ (master)
$ cd PET

joaog@DESKTOP-HKHI60K MINGW64 ~/PET (master)
$ git branch
* master
  novo

joaog@DESKTOP-HKHI60K MINGW64 ~/PET (master)
$ git checkout novo
Switched to branch 'novo'

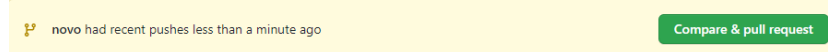
joaog@DESKTOP-HKHI60K MINGW64 ~/PET (novo)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'testes/master'.

joaog@DESKTOP-HKHI60K MINGW64 ~/PET (master)
$
```

Porém, para que uma nova *branch* seja levada para o repositório remoto é necessário que ela seja permitida a fazer isso(o que se aplica para a *branch* principal também). Para isso, na hora de usar o comando *push* usa-se *git push origin "nome da nova branch"*, assim, nas próximas vezes que for fazer o *push* nessa *branch* precisará somente usar o *git push* (igual a *branch* principal).

3 *GitHub*

Outra parte importante de se conhecer é como unir as *branches* dentro do *GitHub*. Quando se dá o *push* para uma *branch* diferente da principal precisa-se checar as mudanças para depois fazer um "*pull request*" para uni-las a principal. Isso acontece, pois, a ideia de ter *branches* diferentes é para que uma equipe de programadores consiga alterar o mesmo código sem que um interfira com o outro, facilitando muito projetos com grupos maiores. Para começar, quando as mudanças forem para o *GitHub* a seguinte mensagem no repositório aparecerá. Sendo "novo" o nome da *branch*.



Clicando no botão verde o site te levará para ver as mudanças e confirmar a união com a *branch* principal.

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

base: master ← compare: novo ✓ Able to merge. These branches can be automatically merged.

Novo

Write Preview H B I ≡ < > 🔗 ≡ ≡ ≡ ☑ @ 📎 ↶

Leave a comment

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

Create pull request

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

clicando em *"create pull request"* e depois em *"merge pull request"* na próxima página e confirmando, todas as mudanças da *branch* serão mandadas para a *branch* principal.

Continuous integration has not been set up
[GitHub Actions](#) and [several other apps](#) can be used to automatically catch bugs and enforce style.

✓ This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

Pull request successfully merged and closed
You're all set—the `novo` branch can be safely deleted.

Delete branch

Com isso feito o versionamento e armazenamento dos códigos foi feito com sucesso.

Conclusão

Concluindo, o uso de softwares de versão e plataformas de hospedagem de código-fonte como o *git* e *GitHub*, respectivamente, são essenciais para a organização e controle de projetos contemporâneos.

Referências Bibliográficas