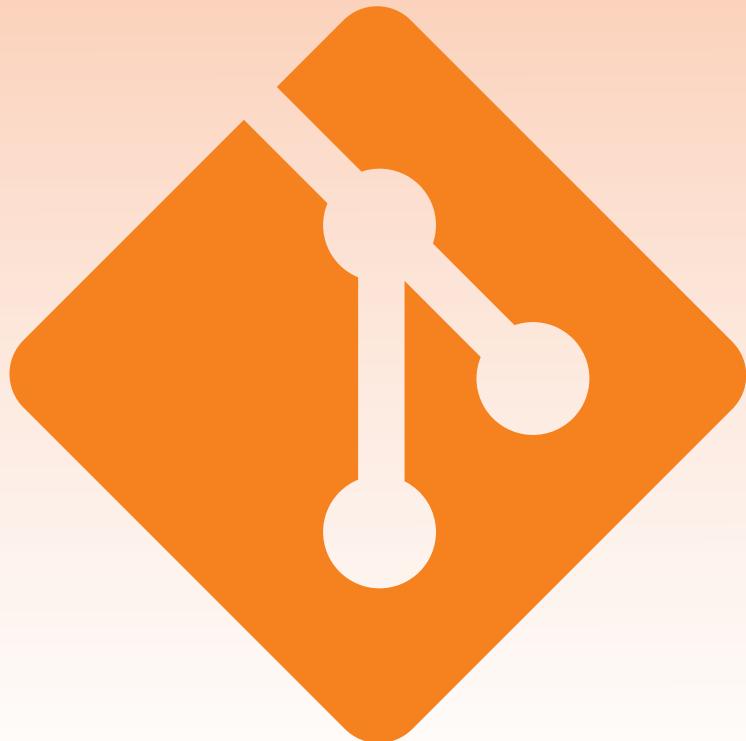




git init

uma introdução ao controle
de versão com Git



2023

Programa de Educação Tutorial - Engenharia Elétrica
Universidade Federal de Minas Gerais

Prefácio

O Git trata-se de uma ferramenta fundamental no processo de desenvolvimento de software. Portanto, esta apostila traz, por fim primordial, introduzir de maneira básica os conceitos relacionados a essa ferramenta de controle de versão. Assim, essa apostila pode ser usada por auxílio aos cursos básicos de desenvolvimento de software.

É com muito orgulho que podemos compartilhar nossos conhecimentos obtidos anteriormente e durante a redação da presente apostila.

Autores

- Davi Ferreira Santiago
- Felipe Meireles Leonel
- João Vitor Saade Simão



PETEE UFMG



/peteeUFMG



www.petee.cpdee.ufmg.br



@petee.ufmg

Grupo PETEE

O que é PET?

Os grupos PETs são organizados a partir de formações em nível de graduação nas Instituições de Ensino Superior do país orientados pelo princípio da indissociabilidade entre **ensino, pesquisa e extensão** e da educação tutorial.

Por esses três pilares, entende-se por:

- Ensino: As atividades extra-curriculares que compõem o Programa têm como objetivo garantir a formação global do aluno, procurando atender plenamente as necessidades do próprio curso de graduação e/ou ampliar e aprofundar os objetivos e os conteúdos programáticos que integram sua grade curricular.
- Pesquisa: As atividades de pesquisa desenvolvidas pelos petianos têm como objetivo garantir a formação não só teórica, mas também prática, do aluno, de modo a oferecer a oportunidade de aprender novos conteúdos e já se familiarizar com o ambiente de pesquisa científica.
- Extensão: Vivenciar o processo ensino-aprendizagem além dos limites da sala de aula, com a possibilidade de articular a universidade às diversas organizações da sociedade, numa enriquecedora troca de conhecimentos e experiências.

PETEE UFMG

O Programa de Educação Tutorial da Engenharia Elétrica (PETEE) da Universidade Federal de Minas Gerais (UFMG) é um grupo composto por graduandos do curso de Engenharia Elétrica da UFMG e por um docente tutor.

Atualmente, o PETEE realiza atividades como oficinas de robôs seguidores de linha, minicursos de Matlab, minicursos de LaTeX, Competição de Robôs Autônomos (CoRA), escrita de artigos científicos, iniciações científicas, etc.

Assim como outras atividades, o grupo acredita que os minicursos representam a união dos três

pilares: O pilar de ensino, porque ampliam e desenvolvem os conhecimentos dos petianos; O pilar da pesquisa, pois os petianos aprendem novos conteúdos e têm de pesquisar para isso; O pilar da extensão, porque o produto final do minicurso é levar à comunidade os conhecimentos adquiridos em forma de educação tutorial.

O Grupo

Alexandre Augusto Leal Martins
Arnaldo Kokke de Brito
Caio Teraoka de Menezes Câmara
Davi Ferreira Santiago
Felipe Meireles Leonel
Fernanda Camilo dos Santos Xavier
Gabriel Costa Matsuzawa

Gustavo Alves Dourado
Izaias Barboza Neto
João Vitor Saade Simão
Lucas José de Souza Oliveira
Luciana Pedrosa Salles
Yuan Dias Fernandes Pena Pereira

Agradecimentos

Agradecemos ao Ministério da Educação (MEC), através do Programa de Educação Tutorial (PET), à Pró-Reitoria de Graduação da Universidade Federal de Minas Gerais (UFMG), à Fundação Christiano Ottoni (FCO) e à Escola de Engenharia da UFMG pelo apoio financeiro e fomento desse projeto desenvolvido pelo grupo PET Engenharia Elétrica da UFMG (PETEE - UFMG).

Agradecemos especialmente ao Prof. Dr. Douglas Guimarães Macharet do Laboratório de Visão Computacional e Robótica (VeRLab) e do Departamento de Ciência da Computação da UFMG pelas sugestões e recomendações durante a confecção dessa apostila.

Contato

Site:
<http://www.petee.cpdee.ufmg.br/>

Facebook:
<https://www.facebook.com/peteeUFMG/>

Instagram:
<https://www.instagram.com/petee.ufmg/>

E-mail:
petee.ufmg@gmail.com

Localização:
Universidade Federal de Minas Gerais, Escola de Engenharia, Bloco 3, Sala 1050.

```
santiagofdavi@sANTIAGO-LAPTOP:~$ sudo apt install git  
[sudo] password for santiagofdavi:  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
Suggested packages:  
  git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk gitweb git-cvs git-mediawiki git-svn  
The following packages will be upgraded:  
  git  
1 upgraded, 0 newly installed, 0 to remove and 158 not upgraded.  
Need to get 4557 kB of archives.  
After this operation, 41.0 kB of additional disk space will be used.  
Ign:1 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 git amd64 1:2.25.1-1ubuntu3.5  
Err:1 http://security.ubuntu.com/ubuntu focal-updates/main amd64 git amd64 1:2.25.1-1ubuntu3.5  
  404  Not Found [IP: 185.125.190.39 80]  
E: Failed to fetch http://security.ubuntu.com/ubuntu/pool/main/g/git/git_2.25.1-1ubuntu3.5_amd64.deb  404  Not Found [IP:  
  : 185.125.190.39 80]  
E: Unable to fetch some archives, maybe run apt-get update or try with --fix-missing?
```

Sumário

I

Introdução

1	Apresentação	10
1.1	O que é Git?	10
1.1.1	Tipos de sistemas de controle de versões	10
1.1.2	Como o Git pode ser classificado	11
1.2	História	11
2	Instalação e Configurações Iniciais	13
2.1	Instalação	13
2.1.1	Conferindo instalações anteriores	13
2.1.2	Windows	13
2.1.3	MacOS	15
2.1.4	Linux	16
2.2	Configurando um usuário	16

II

Conceitos Básicos

3	Comandos Básicos	19
3.1	Inicialização e configuração	19
3.1.1	git init	19
3.1.2	git config	19
3.2	Enviar arquivos	19
3.2.1	git add	19

3.2.2	git commit	20
3.2.3	git fetch	21
3.2.4	git pull	21
3.2.5	git push	22
3.2.6	Arquivos .gitignore	22
3.2.7	git tag	23
3.3	Verificar informações	25
3.3.1	git log	25
3.3.2	git diff	25
3.4	Receber arquivos	26
3.4.1	git clone	26
3.5	Alterar branches	26
3.5.1	git branch	26
3.5.2	git checkout	27
3.5.3	git merge	28
3.5.4	git rebase	29
3.5.5	git stash	30
4	GitHub	32
4.1	Função	32
4.2	Cadastro	32
4.2.1	Chave SSH	33
4.3	Utilização	34
4.3.1	Criação de repositórios	35
4.3.2	Gerenciamento de repositórios	36
4.3.3	Perfil	38

III

Exercícios

5	Configurando o Git Exercises	41
5.1	Configurações Iniciais	41
5.2	Comandos do Git Exercises	42
5.2.1	Executando os exercícios	42
5.2.2	Corrigindo uma atividade	42
5.2.3	Executando exercícios em sequência	42
6	Exercícios	43
7	Dicas	47
8	Gabaritos	49

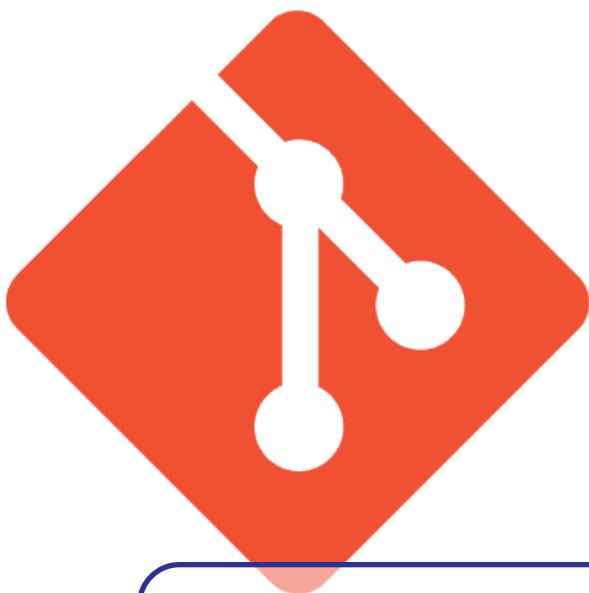
9	GitKraken	53
9.1	Função	53
9.2	Instalação	53
9.3	Integração com o GitHub	53
9.4	Utilização	54
9.4.1	Abrir um repositório	54
9.4.2	Clonar repositório	55
9.4.3	Criar repositório	55

Bibliografia	58
---------------------------	-----------



Introdução

1	Apresentação	10
1.1	O que é Git?	
1.2	História	
2	Instalação e Configurações Iniciais ...	13
2.1	Instalação	
2.2	Configurando um usuário	



git

1. Apresentação

1.1 O que é Git?

O Git é um sistema de controle de versões, ou seja, tem por finalidade o gerenciamento de versões de um determinado arquivo ao longo do tempo. Para isso, ele salva um estado do arquivo que está sendo alterado a fim de ser possível voltar a esse estado novamente caso as alterações feitas não sejam satisfatórias. Usar esse tipo de software é muito vantajoso, pois evita que se armazene cópias dos vários estados de um arquivo no sistema e, assim, diminui-se muito o risco de se perder uma versão que se deseje utilizar no futuro.

1.1.1 Tipos de sistemas de controle de versões

Existem dois tipos de sistemas de versionamento, os centralizados e os distribuídos. Os centralizados funcionam a partir de um servidor central que recebe as novas versões e recupera versões anteriores para os diversos usuários que estão trabalhando no arquivo. Já os distribuídos, cada usuário tem um servidor local para salvar os estados do arquivo editado e após esse usuário chegar em uma versão estável ele envia suas alterações para um servidor central onde está o arquivo principal. Esse tipo de sistema de versionamento é o ideal para projetos que estão sendo editados por um número grande de pessoas, porque a organização do projeto e o tratamento de erros fica bem mais fácil de ser realizado.

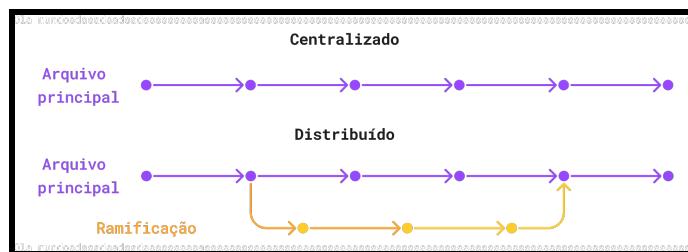


Figura 1.1.1: Tipos de sistemas de versionamento

1.1.2 Como o Git pode ser classificado

O Git é considerado um software que se utiliza de um sistema de controle de versões distribuído pelo fato de organizar os projetos em forma de árvores. Assim, o projeto fica organizado com um “tronco” tendo o projeto principal e “galhos” com as versões em desenvolvimento.

1.2 História

O sistema de controle de versões Git foi desenvolvido por Linus Torvalds e a comunidade Linux com o objetivo de criar um sistema concorrente ao Bitkeeper e que ainda suprisse os pontos deficitários dos concorrentes. Essa necessidade de criar um novo sistema de versionamento de software veio a partir de 2005 quando a equipe do BitKeeper deixou sua ferramenta paga para a comunidade Linux por conta de uma alegação de violação de licença. A partir disso, Linus e sua comunidade se viram obrigados a criar seu próprio sistema para gerenciar as versões do kernel do Linux.



Figura 1.2.1: Linus Torvalds, o criador do kernel do Linux e do Git.

Ao desenvolver o Git, Linus tinha em mente que seu sistema não poderia ser igual ao BitKeeper, mas sim uma versão melhorada. Para esse fim, ele decidiu aplicar alguns critérios para o desenvolvimento do Git:

- Fazer algo que não se pareça de forma alguma com CVS(Sistema de controle de versão considerado por Linus Torvalds como ineficiente)
- Trabalhar com um fluxo distribuído.
- Ter um robusto sistema contra corrompimento de arquivos.
- Ser altamente eficiente.

Assim, em 3 de abril de 2005 o desenvolvimento do sistema foi iniciado e já em 29 de abril do mesmo ano ele já era referência por conta de sua velocidade em relação aos concorrentes. Linus esteve a frente do projeto até o momento em que viu o Git em uma versão utilizável, após isso a manutenção ficou sob a gestão de um dos principais contribuidores nesse desenvolvimento, Junio Hamano. Por fim, o desenvolvimento inicial do Git culminou com o lançamento de sua versão 1.0 em 21 de dezembro de 2005.

Atualmente o Git é um dos sistemas de versionamento mais utilizados no mercado, sendo usado por grandes empresas como Twitter e Yahoo. Assim, sendo indiscutível que o conhecimento básico

desse sistema é de suma importância para qualquer um que queira trabalhar com tecnologia nos dias atuais.

```
santiagofdavi@sANTIAGO-LAPTOP:~$ sudo apt install git  
[sudo] password for santiagofdavi:  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
Suggested packages:  
  git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk gitweb git-cvs git-mediawiki git-svn  
The following packages will be upgraded:  
  git  
1 upgraded, 0 newly installed, 0 to remove and 158 not upgraded.  
Need to get 4557 kB of archives.  
After this operation, 41.0 kB of additional disk space will be used.  
Ign:1 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 git amd64 1:2.25.1-1ubuntu3.5  
Err:1 http://security.ubuntu.com/ubuntu focal-updates/main amd64 git amd64 1:2.25.1-1ubuntu3.5  
  404  Not Found [IP: 185.125.190.39 80]  
E: Failed to fetch http://security.ubuntu.com/ubuntu/pool/main/g/git/git_2.25.1-1ubuntu3.5_amd64.deb  404  Not Found [IP:  
  : 185.125.190.39 80]  
E: Unable to fetch some archives, maybe run apt-get update or try with --fix-missing?
```

2. Instalação e Configurações Iniciais

2.1 Instalação

Existem várias maneiras de se realizar a instalação do Git em diferentes sistemas operacionais. Nesta seção são apresentadas algumas sugestões dos autores de como fazê-la.

2.1.1 Conferindo instalações anteriores

Por diversos motivos o Git já pode ter sido instalado por acaso em sua máquina (como a instalação do XCode no MacOS, por exemplo). Para verificar, abra o seu terminal ou prompt de comando e execute o seguinte comando:

```
$ git --version
```

Caso seja retornado algum número de versão, o Git já se encontra instalado em sua máquina:

```
git version 2.25.1;
```

Caso o retorno seja negativo, prossiga com a instalação correspondente ao seu sistema operacional.

2.1.2 Windows

Inicie o browser de sua preferência e acesse o seguinte link: <https://gitforwindows.org/>. Esse link direciona para a página oficial de downloads do instalador do Git para Windows.



Figura 2.1.1: Site *Git for Windows*.

Faça o download do instalador. Uma vez finalizado, basta clicar duas vezes com o mouse sobre o arquivo para abrir o instalador; iniciando-se, assim, a instalação.



Figura 2.1.2: *Instalador quando aberto*.

Para concluir a instalação, basta seguir selecionando as configurações pessoais desejadas (na dúvida, não altere as configurações sugeridas pelo instalador) e pressionando o botão Next. Uma vez finalizado o processo, basta pressionar o botão Finish para concluir a instalação com sucesso.

Para confirmar o êxito da instalação, execute novamente o seguinte comando:

```
$ git --version
```

Assim, uma vez instalado corretamente, será retornado o número da versão instalada.

2.1.3 MacOS

Inicie o browser de sua preferência e acesse o seguinte link: <https://sourceforge.net/projects/git-osx-installer/files/>. Esse link direciona para a do SourceForge (um repositório de código fonte baseado em Web).

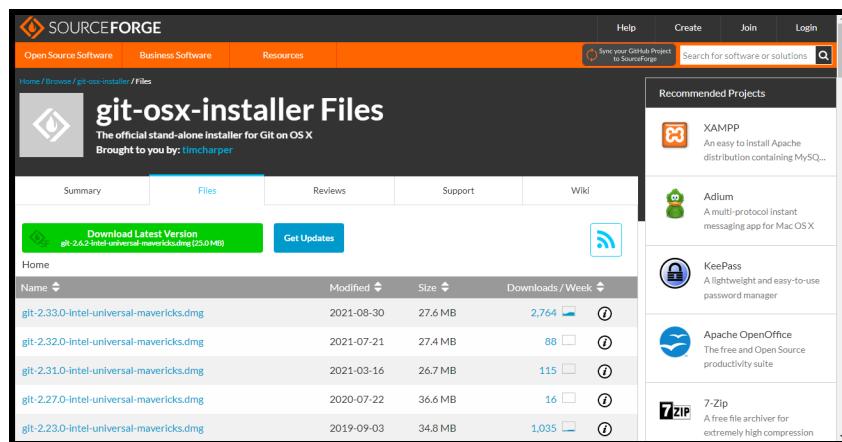


Figura 2.1.3: Site oficial SourceForge.

Assim, escolha a versão mais recente para Mac, siga as instruções da tela e realize a instalação.

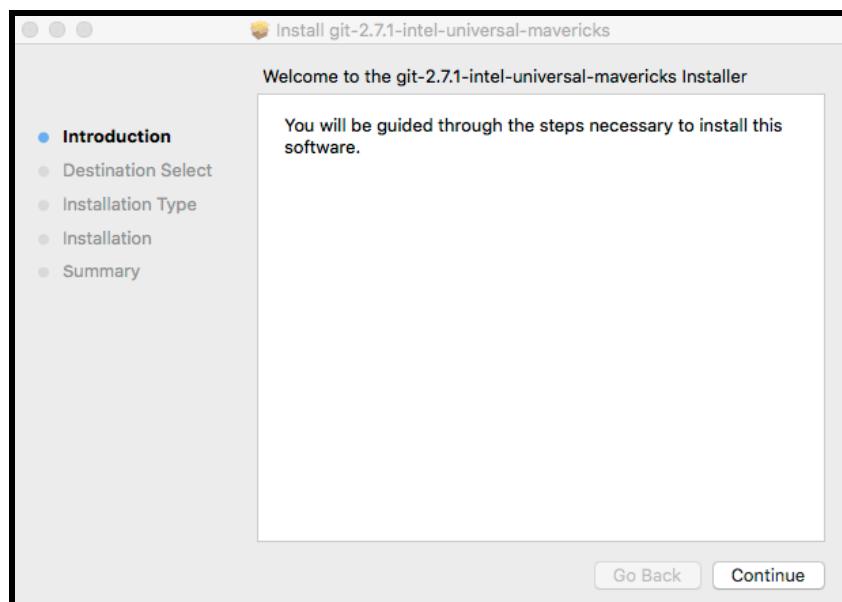


Figura 2.1.4: Instalador quando aberto.

Para concluir a instalação, basta seguir selecionando as configurações pessoais desejadas (na dúvida, não altere as configurações sugeridas pelo instalador) e pressionando o botão Continue.

Para confirmar o êxito da instalação, execute novamente o seguinte comando:

```
$ git --version
```

Assim, uma vez instalado corretamente, será retornado o número da versão instalada.

2.1.4 Linux

A instalação no Linux é mais simples e independe de um instalador.

Para usuários Debian/Ubuntu, basta inicializar o terminal e digitar os seguintes comandos (utilizando o apt-get):

```
$ sudo apt-get update  
$ sudo apt-get install git
```

Para usuários Fedora, basta inicializar o terminal e digitar os seguintes comandos para baixar pacotes Git (utilizando yum/dnf):

```
$ sudo dnf install git  
$ sudo yum install git
```

Para confirmar o êxito da instalação, execute novamente o seguinte comando:

```
$ git --version
```

Assim, uma vez instalado corretamente, será retornado o número da versão instalada.

2.2 Configurando um usuário

Uma vez realizada a instalação com sucesso, abra o seu terminal e, substituindo os dados que se encontram entre aspas pelos seus próprios, execute os seguintes comandos:

```
$ git config --global user.name "John Stevens"  
$ git config --global user.email "johnstevens@email.com.br"
```

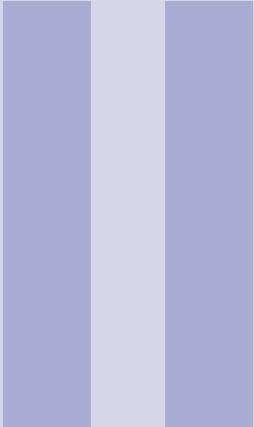
Esses comandos configurarão o seu nome de usuário e o seu e-mail associados à sua conta Git. Qualquer commit (será visto em Comandos Básicos) feito será creditado a esses dados.

Para checar os dados inseridos, execute:

```
$ git config user.name  
$ git config user.email
```

Os valores retornados deverão ser, respectivamente:

```
John Stevens  
johnstevens@email.com.br
```



Conceitos Básicos

3	Comandos Básicos	19
3.1	Inicialização e configuração	
3.2	Enviar arquivos	
3.3	Verificar informações	
3.4	Receber arquivos	
3.5	Alterar branches	
4	GitHub	32
4.1	Função	
4.2	Cadastro	
4.3	Utilização	

Basic Git Commands



3. Comandos Básicos

3.1 Inicialização e configuração

3.1.1 git init

Esse comando é responsável por inicializar um novo repositório. Em um processo de versionamento de código, esse é o primeiro comando básico.

Em sua máquina crie um novo diretório com o nome de "`git_init_petee`". Em seguida digite o seguinte comando:

```
$ git init
Initialized empty Git repository in /git_init_petee/.git/
```

Sendo, dessa forma, retornado o caminho do diretório inicializado.

3.1.2 git config

Assim como visto anteriormente, o comando `git config` é utilizado para configurar as opções de instalação e de usuário do git.

Para configurações básicas do git, consulte a seção 2.2.

3.2 Enviar arquivos

3.2.1 git add

Este é responsável por adicionar supostas mudanças no diretório do projeto à área de staging, dando uma oportunidade de preparar a snapshot antes de submeter o arquivo ao projeto oficial, ou seja, antes de fazer o `commit`.

Assim, no diretório criado anteriormente, crie um arquivo `hello_world.cpp`. Trata-se de um projeto básico na linguagem C++ que apenas imprime a frase

```
"Hello Word!"
```

na tela.

Assim, após criar o arquivo, copie o seguinte código e salve-o.

```
1 #include <iostream>
2
3 int main(){
4     std::cout << "Hello Word!" << std::endl;
5     return 0;
6 }
```

ATENÇÃO! Repare que há um erro: ao invés de estar escrito "World", está "Word". Esse erro deve ser mantido e seu propósito será justificado em breve.

Para adicionar as mudanças no arquivo acima, dentro do diretório raiz do projeto, basta digitar o comando:

```
$ git add hello_world.cpp
```

Para verificar o estado da área de staging, ou seja, ver o que está no atual snapshot, usa-se o seguinte comando:

```
$ git status
On branch master

    No commits yet

    Changes to be committed:

        (use "git rm --cached <file>..." to unstage)

        new file:   hello_world.cpp
```

3.2.2 git commit

Para submeter as mudanças inclusas no snapshot da área de staging, ou seja, aquelas ao adicionar o arquivo "hello_world.cpp", usa-se o comando **git commit**.

Para tal, digite, em seu terminal, o seguinte comando:

```
$ git commit -am "Adicionei o arquivo hello_world.cpp"
```

Acima, observa-se que há a flag *-am*. Esta permite que o usuário escreva uma mensagem, no caso, dando um título que justifique as alterações que foram incluídas na versão submetida do projeto.

```
$ git commit -am "Adicionei o arquivo hello_world.cpp"
[master (root-commit) 380e42d] Adicionei o arquivo
  hello_world.cpp
1 file changed, 6 insertions(+)
create mode 100644 hello_world.cpp
```

3.2.3 git fetch

O fetch faz o download a partir da ramificação de outro repositório, junto com todos os commits e arquivos associados. Mas, não tenta integrar nada em o repositório local. Assim, você tem a oportunidade de inspecionar as alterações antes de fazer o merge no projeto.

O formato é:

```
$ git fetch <remote>
```

Exemplo de uso:

```
$ git remote add coworkers_repo
git@bitbucket.org:cworker/coworkers_repo.git
```

Esse primeiro passo configura o repositório remoto.

```
$ git fetch coworkers_repo coworkers/feature_branch
fetching coworkers/feature_branch
```

Em seguida, deve-se passar esse nome remoto para git fetch baixar os conteúdos.

3.2.4 git pull

Git pull é a versão automatizada do git fetch. Ele baixa uma ramificação de um repositório remoto e a mescla imediatamente na ramificação atual. Este é o Git equivalente ao svn update.

Formato:

```
$ git pull <remote>
```

Exemplo:

```
$ git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3) done.
```

```

remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3) done.
From https://github.com/ImDwivedi1/GitExemple2
  f1ddc7c..0a1475 master      -> origin/master
Updating f1ddc7c..0a1a475
Fast-forward
 design2.css | 6 ++++++
 1 file changed, 6 insertions(+)
 create mode 100644 design2.css

```

3.2.5 git push

Fazer o push ("empurrar") é o oposto de fazer o fetch. Enquanto o comando git fetch é usado para importar commits para repositórios locais, o comando git push transfere commits de um repositório local para um repositório remoto.

Para adicionar um repositório remoto, basta usar o seguinte comando:

```
$ git remote add <name> <url>
```

Onde **<name>** refere-se ao nome do repositório remoto e **<url>** é seu respectivo link.

Assim, para fazer o push, utiliza-se a seguinte estrutura:

```
$ git push <name>
```

Onde **<name>** refere-se ao nome do repositório remoto.

3.2.6 Arquivos .gitignore

Esse tipo de arquivo serve para o Git saber quais arquivos ele deve ignorar na hora de fazer um commit. Dessa forma se agiliza o trabalho do usuário que não precisa ficar tirando de todo commit arquivos indesejados como, por exemplo, senhas, arquivos de configurações de IDE pessoais, etc.

Para utilizar esse tipo de arquivo, é necessário que após tê-lo criado e colocado dentro do repositório local se escreva dentro dele quais arquivos serão ignorados. Um exemplo de arquivo **.gitignore** é o seguinte:

```

senhas.txt
testes/
uploads/

```

Nesse exemplo o Git iria ignorar na hora de fazer commit os arquivos **senhas.txt** e as pastas **testes** e **uploads**.

Uma forma de facilitar a seleção de arquivos é utilizando caracteres coringas, sendo esses os seguintes:

*	Substitui qualquer coisa
?	Substitui apenas um caractere
[]	Para intervalos
!	Para negação
#	Indica um comentário

Tabela 3.2.1: Caracteres coringas

Um exemplo de arquivo `.gitignore` que se utiliza desse caracteres pode ser escrito da seguinte forma

```
#Ignora todos os arquivos .txt
*.txt
#Ignora os arquivos que começam com erro, tendo qualquer
    carácter a frente e sendo .log
erro?.log
#Ignora os arquivos teste- que terminam com números de 0 a 9
    e tenhas a extensão .log
teste-[0-9].log
#Faz com que o arquivo teste-4.log não seja ignorado
!teste-4.log
```

3.2.7 git tag

Essa função do Git serve para criar etiquetas de estados que sejam relevantes, como por exemplo, a versão final ou alguma versão já utilizável do projeto. Assim, o Git salva versões para organizar melhor o projeto e também para poder voltar se em uma versão posterior ocorrer algum problema.

Existem três formas de criar uma tag utilizando o Git, sendo elas: com anotações, sem anotações e após o commit já feito.

Para criar uma tag com mensagem se utiliza o comando abaixo colocando o nome da tag após o -a e uma mensagem depois do -m:

```
$ git tag -a v1.0 -m 'Minha versão 1.0'
```

Já para criar uma tag sem anotações usa-se apenas o comando base com nome da versão:

```
$ git tag v1.1
```

E para marcar um commit depois de já tê-lo feito se usa o comando `git tag` com o nome da versão e a soma de verificação do commit.

```
$ git tag v1.2 166ae0c4d3f420721acbb115cc33848dfcc2121a
```

Essa soma de verificação você pode conseguir usando o comando abaixo e localizando o número hexadecimal que está na mesma linha da mensagem do commit desejado.

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'Teste'
a6b4c97498bd301d84096da251c98a07c7723e65 Otimizacao
0d52aab4479697da7686c15f77a3d64d9165190 Corrigindo erros
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'Teste'
0b7434d86859cc7b8c3d5e1dddfe66ff742fcbe Adicionando videos
4682c3261057305bdd616e23b64b0857d832627b Adicionando imagens
166ae0c4d3f420721acbb115cc33848dfcc2121a Otimizacao
9fce802d0ae598e95dc970b74767f19372d61af8 Update readme
964f16d36dfccde844893cac5b347e7b3d44abbc Alteracoes
8a5cbc430f1a9c3d00faaeffd07798508422908a Update readme
```

Ademais, é possível ver informações da tag usando o comando git show e o nome da tag:

```
$ git show v1.1
tag nome_da_tag
Tagger: John Stevens <johnstevens@email.com.br>
Date:   Sun feb 12 13:15:21 2023 -0700

Minha versão v1.1

commit 0d52aab4479697da7686c15f77a3d64d9165190
Author: Fred Brown <fredbrown@email.com.br>
Date:   Wed jan 18 10:12:55 2023 -0700

    Change version number
```

Para finalizar o processo para criar uma tag ainda é necessário enviá-la para o servidor remoto usando o git push:

```
$ git push origin v1.1
```

Caso queira excluir uma tag do repositório remoto usa-se o comando:

```
$ git push origin --delete v1.2
```

Já para excluir do repositório local é usado:

```
$ git tag -d v1.2
```

3.3 Verificar informações

3.3.1 git log

Permite que você explore as revisões passadas de um projeto. Ele oferece muitas opções de formatação para indicar os snapshots commitados.

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700
```

Existem algumas flags úteis, como:

- **-p**: Mostra o patch introduzido com cada commit.
- **-stat**: Mostra estatísticas de arquivos modificados em cada commit.
- **-shortstat**: Exibe apenas a linha informando a alteração, inserção e exclusão do comando `-stat`.
- **-name-only**: Mostra a lista de arquivos modificados após as informações de commit.
- **-name-status**: Mostra também a lista de arquivos que sofreram modificação com informações adicionadas / modificadas / excluídas.
- **-relative-date**: Exibe a data em um formato relativo (por exemplo, ‘2 semanas atrás’) em vez de usar o formato de data completo.
- **-(n)**: Exibe somente os últimos n commits
- **-since, -after**: Limita os commits para aqueles feitos após a data especificada.
- **-until, -before**: Limita os commits aos feitos antes da data especificada.
- **-author**: Mostra apenas os commits nos quais a entrada do autor corresponde à string especificada
- **-grep**: Mostra apenas os commits com uma mensagem de commit contendo a string

3.3.2 git diff

Mostrar alterações entre a árvore de trabalho e o índice ou uma árvore, alterações entre o índice e uma árvore, alterações entre duas árvores, alterações resultantes de uma mesclagem, alterações entre dois objetos blob ou alterações entre dois arquivos no disco.

Em vermelho aparece o que foi excluído e em verde aparece o que foi adicionado.

```
$ git diff
diff --git a/test.txt b/sample.txt
```

```
index ad01390..5629924 10644
---- a/test.txt
+++ b/sample.txt
@@ -1,3 +1,3 @@
- This is a test
- \hspace{1cm}* Hello
- \hspace{1cm}* world!
+ This is a sample
+ * Hello
+ * universe!
```

3.4 Receber arquivos

3.4.1 git clone

Este comando cria uma cópia de um repositório já existente.

Digite o seguinte comando em seu terminal:

```
$ git clone https://github.com/santiagofdavi/git_petee_course
Cloning into 'git_petee_course'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (9/9), 2.00 KiB | 342.00 KiB/s, done.
```

Observa-se que no diretório onde você está foi criada uma nova pasta "*git_petee_course*", referente ao projeto "clonado".

ATENÇÃO! Repare que o projeto clonado localiza-se no GitHub.

Link de acesso: https://github.com/santiagofdavi/git_petee_course

3.5 Alterar branches

3.5.1 git branch

Esse grupo de comandos é utilizado para gerenciar as branches de um repositório. Suas principais funções são a de criar uma branch, listar todas as branches e deletar uma branch.

Uma branch pode nos auxiliar imensamente no processo de manutenção de nosso repositório. Por exemplo, atualmente nosso arquivo "*hello_world.cpp*" está printando uma informação com erro de digitação. Para fazer a alteração correndo o menor número de risco, podemos criar uma branch apenas para isso. Assim, criamos a branch usando o seguinte código:

```
$ git branch manutencao
```

Caso queira ver as branches que seu repositório tem usa-se:

```
$ git branch
*main
manutencao
```

Já para deletar uma branch usa-se:

```
$ git branch -d manutencao
Deleted branch manutencao (3a0874c).
```

ATENÇÃO! Não executar esse comando por agora, pois a branch em questão será importante.

3.5.2 git checkout

Esse comando do Git tem como função mudar de branch ou voltar para algum estado do seu projeto. Sendo de extrema importância para qualquer repositório que utilize branches, por exemplo, no repositório que estamos trabalhando podemos fazer a mudança para branch *manutencao* a fim de começar as alterações no arquivo "*hello_world.cpp*". Para isso utiliza-se o comando abaixo.

```
$ git checkout manutencao
```

Agora, para prosseguir na manutenção você pode fazer a correção utilizando o seguinte código:

```
1 #include <iostream>
2
3 int main(){
4     std::cout << "Hello World!" << std::endl;
5     return 0;
6 }
```

Após isso será necessário apenas fazer o commit na branch *manutencao* e realizar testes nas suas alterações para poder manda-las da branch atual para a principal.

Outra funcionalidade interessante do git checkout é sua utilização para voltar em algum commit anterior utilizando o comando base mais o código de soma de verificação do commit:

```
$ git checkout 166ae0c4d3f420721acbb115cc33848dfcc2121a
```

E para ir para uma tag se utiliza o comando base seguido de tag/ e o nome da tag:

```
$ git checkout tag/v1.1
```

DICA Para já criar um branch e depois trocar para ele automaticamente basta usar o comando:

```
$ git checkout -b nome_da_branch
```

3.5.3 git merge

A funcionalidade merge do git serve para fazer uma mesclagem entre branches, fazendo com que as alterações feitas em outra branch sejam enviadas para o branch em questão. Para ser usado é preciso apenas usar o comando base da função e em seguida a nome da branch que se deseja mesclar com a atual. Voltando ao nosso mini projeto, após já ter feito as correções no código podemos mesclar a branch *manutencao* com a *main*. Para isso primeiro voltamos até a branch *main* com o git checkout e depois realizamos o merge.

```
$ git checkout main
Switched to branch 'main'
$ git merge manutencao
Updating f42c576..3a0874c
Fast-forward
  hello_world.cpp | 1 ++
  1 file changed, 1 insertions(+)
```

Um problema comum quando falamos de merge são os conflitos, sendo estes resultado de alterações em mesmas partes de um determinado documento nas branches que farão parte do merge ou quando um arquivo de uma branch é excluído e na outra são feitas alterações. Um exemplo na prática desse tipo de comportamento seria, por exemplo, se caso você estivesse trabalhando em conjunto com outro desenvolvedor no arquivo que criamos e os dois fizessem alterações na mesma linha de código. Um mudasse a linha 4 na branch *main* e depois o outro na branch *manutencao*, da seguinte forma:

```
1 #include <iostream>
2 // Alteração feita na branch main
3 int main(){
4     std::cout << "Hello World! :)" << std::endl;
5     return 0;
6 }
```

```
1 #include <iostream>
2 // Alteração feita na branch manutencao
3 int main(){
4     std::cout << "Olá mundo!" << std::endl;
5     return 0;
6 }
```

O problema ocorreria quando um, pensando que as alterações feitas na branch manutenção estão boas, fizesse um merge na *main* com a branch *manutenção*, pois teria uma conflito no merge.

```
$ git merge manutencao
Auto-merging hello_world.cpp
CONFLICT (content): Merge conflict in hello_world.cpp
Automatic merge failed; fix conflicts and then commit the
result.
```

Isso ocorre porque o Git fica sem saber quais dessas alterações deve ser considerada como a correta e que deve ficar no arquivo após o merge. Para resolver esse problema é preciso alterar o arquivo manualmente e depois fazer o commit da alteração.

```
1 #include <iostream>
2
3 int main(){
4     <<<<< HEAD
5         std::cout << "Hello World! :)" << std::endl;
6     =====
7         std::cout << "Olá mundo!" << std::endl;
8     >>>>> manutencao
9         return 0;
10 }
```

Note que o código das duas branches aparecem no arquivo, sendo o da branch atual identificado com HEAD e a outra branch com o nome dela. Para solucionar o problema você pode escolher um dos dois códigos ou criar um fazendo uma mescla.

```
1 #include <iostream>
2
3 int main(){
4     std::cout << "Olá mundo! :)" << std::endl;
5     return 0;
6 }
```

Após a realização das alterações é necessário apenas fazer um commit para finalizar o merge.

```
$ git add hello_world.cpp
$ git commit -m "Resolução de conflito"
[main 57fc9f5] Resolução de conflito
```

3.5.4 git rebase

Rebase permite mover as branches, ajudando a evitar merge commits desnecessários. A história linear resultante geralmente é muito mais fácil de entender e explorar. Com o comando rebase, você pode pegar todas as alterações que foram confirmadas em um branch e reproduzi-las em outro.

Ele funciona indo para o ancestral comum dos dois branches (aquele em que você está e aquele em que você está fazendo o rebase), obtendo o diff introduzido por cada commit do branch em que você está, salvando esses diffs em arquivos temporários, redefinindo o branch atual para o mesmo commit do branch no qual você está fazendo o rebase e, finalmente, aplicando cada mudança por vez.

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

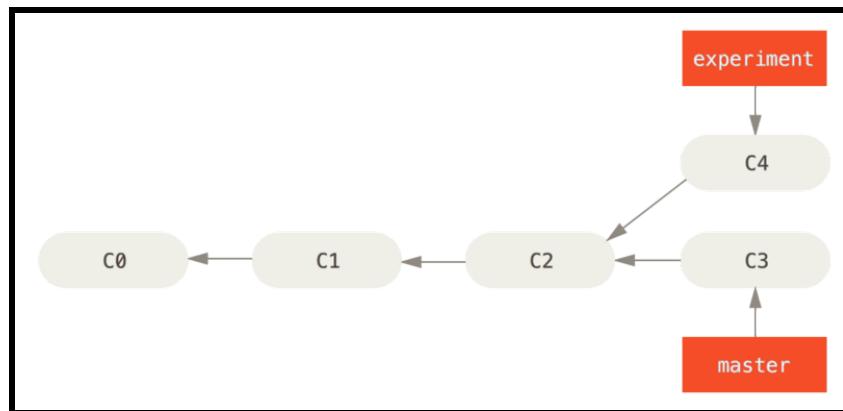


Figura 3.5.1: Situação inicial

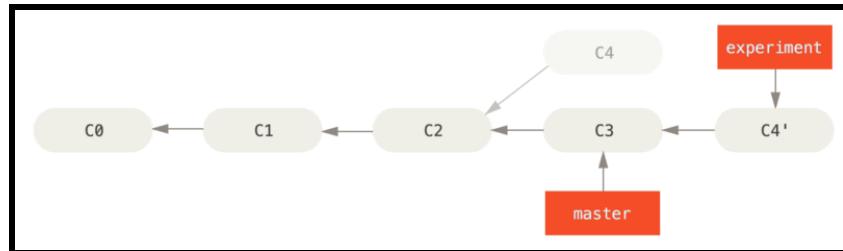


Figura 3.5.2: Situação final

3.5.5 git stash

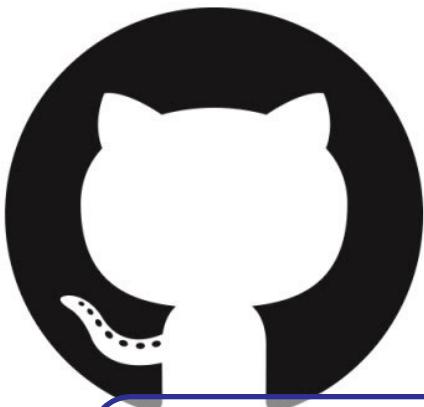
O Git stash arquiva alterações não commitadas do seu local de trabalho, ou seja, ele volta para o estado do seu último commit guardando as alterações adicionais que você tinha feito.

```
$ git stash
Saved working directory and index state WIP on main: 8117923
WIP
HEAD is now at 8117923 WIP
```

Existem algumas flags interessantes, como:

- **list:** para ver todos os stashes guardados.

- **pop**: para restaurar as modificações do último stash.
- **apply**: para aplicar as alterações sem deletar a referência.
- **show**: para ver um resumo do que tem de alterações no seu último stash.
- **show -p**: para ver as alterações completas e mais detalhadas do seu último stash.



GitHub

4. GitHub

4.1 Funcão

O GitHub é uma plataforma web de hospedagem de arquivos utilizando o Git. Dessa forma, ele acaba sendo um facilitador muito grande no trabalho de uma equipe, pois qualquer integrante da mesma, tendo internet, pode acessar os arquivos do projeto sem maiores problemas. Além disso, o GitHub também é uma espécie de rede social, assim, a partir dele é possível seguir outras pessoas, ver seus trabalhos e até interagir com elas.

Let's build from here

Harnessed for productivity. Designed for collaboration.
Celebrated for built-in security. Welcome to the platform
developers love.

Figura 4.1.1: Função segundo o próprio GitHub

4.2 Cadastro

Entrando no site <https://github.com/>, selecione **sign up**. O site te dará boas vindas e pedirá seu email e a senha a ser utilizada para o acesso ao site. Depois, será necessário escolher um nome de usuário que aparecerá para os demais utilizadores. Após esse processo, será necessário confirmar o email com um código enviado para este.

Para fornecer ao usuário um conteúdo mais próximo do procurado, a plataforma faz algumas perguntas sobre seu objetivo ao usar o GitHub.

4.2.1 Chave SSH

Uma configuração muito importante para se fazer no GitHub antes de se começar a usá-lo é a criação de uma chave SSH. Essa se faz necessária, pois o GitHub está descontinuando a funcionalidade de clonar um repositório a partir de um link HTTPS e fazer o *git clone* a partir de uma chave SSH acaba se tornando a forma mais simples de fazer esse processo.

Para gerar a sua chave no Windows você primeiro precisará abrir o Git Bash e depois utilizar o seguinte comando com o seu e-mail no fim para gerar sua chave SSH:

```
$ ssh-keygen -t ed25519 -C "seu_email@exemplo.com"
```

Após isso você precisará escolher onde sua chave será criada e seu nome, porém o mais aconselhável é apenas apertar enter para colocar sua chave no diretório padrão. Depois, de forma parecida, irá aparecer uma seção para criar uma senha ou apenas apertar enter duas vezes para deixar sua chave sem senha. Com a conclusão destes passos sua chave já estará criada e você precisará apenas adicioná-la ao SSH agent antes de registrar sua chave no GitHub. Para fazer isso você precisa iniciar o SSH agent com o seguinte código.

```
$ eval "$(ssh-agent -s)"
```

E registrar sua chave com:

```
ssh-add ~/.ssh/nome_da_chave
```

Já para criar uma chave em um computador MacOS ou linux você primeiro precisa abrir o terminal e digitar o mesmo comando usado no windows:

```
$ ssh-keygen -t ed25519 -C "seu_email@exemplo.com"
```

E o procedimento continua igual ao windows tirando o fato que você não precisa usar SSH agent. Com sua chave em mãos e pronta para uso você deve copiar a sua versão pública(termina com um .pub) com o comando:

```
#Para MacOS  
$ pbcopy < ~/.ssh/nome_da_chave.pub  
#Para Windows  
$ clip < ~/.ssh/nome_da_chave.pub  
#Para linux  
$ cat ~/.ssh/id_ed25519.pub
```

Agora para adicionar sua chave no GitHub você deve clicar no ícone com sua foto de perfil que fica no canto superior direito da tela inicial do GitHub e após isso clicar em *settings*. Na próxima página você deve procurar a opção “SSH and GPG Keys”, seleciona-la, clicar em “New SSH Key” e você entrará na página abaixo. Nela você coloca o nome para identificar sua chave no GitHub e em baixo cola a sua chave.

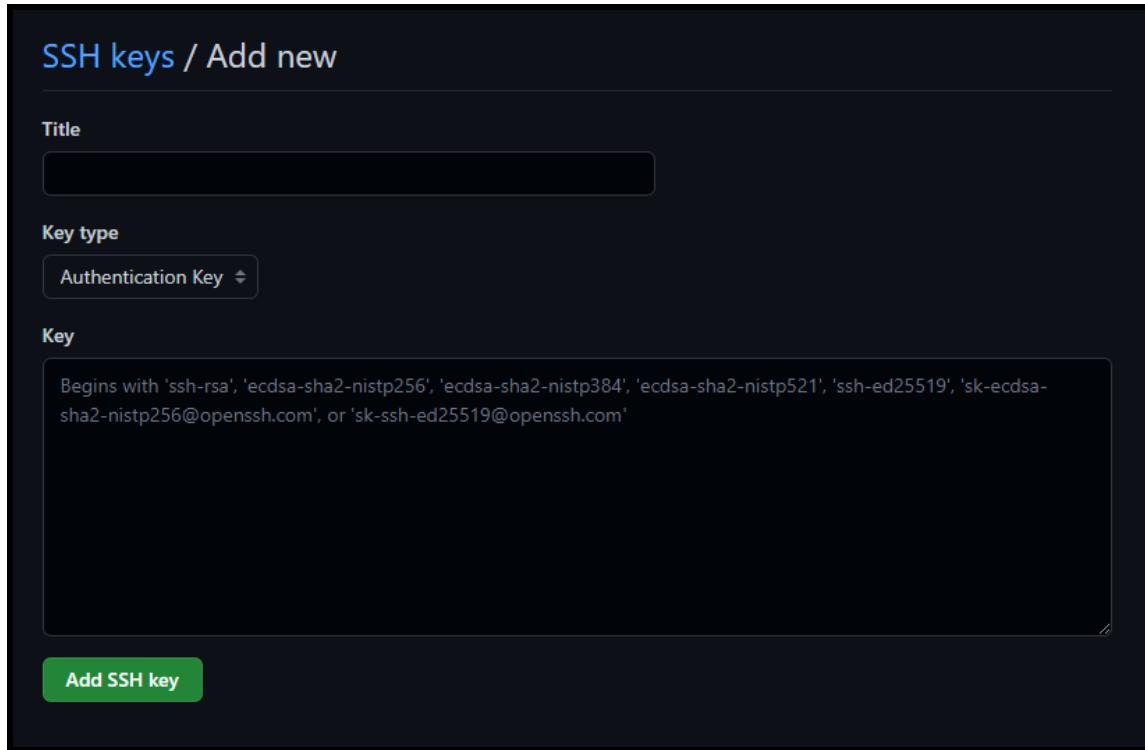


Figura 4.2.1: Tela para cadastro de chave SSH

4.3 Utilização

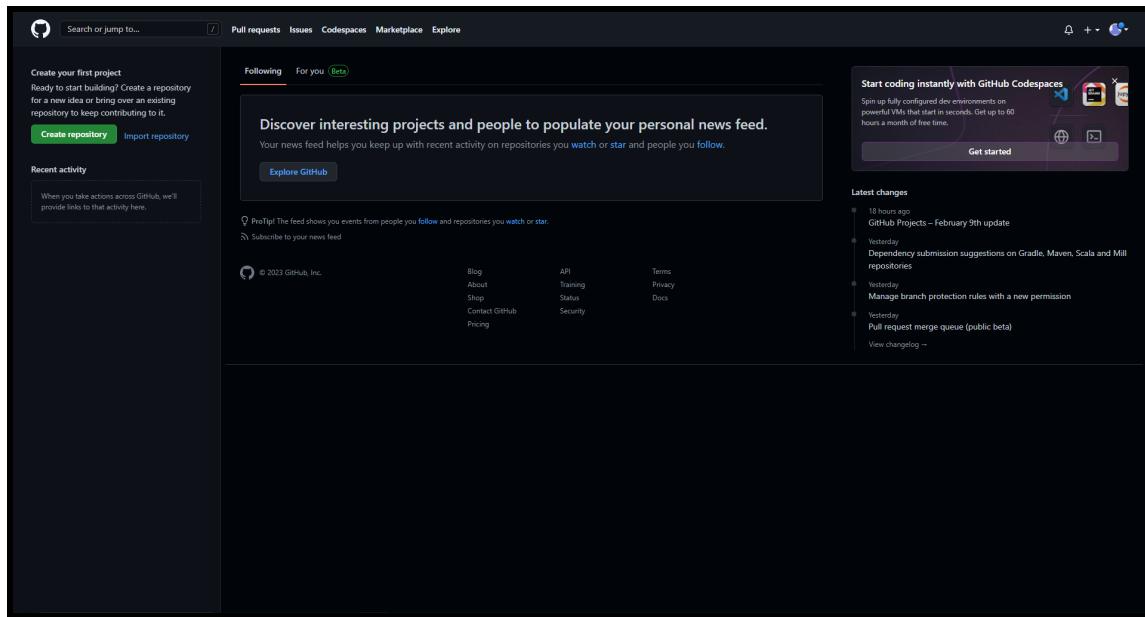


Figura 4.3.1: Tela inicial GitHub

A partir dessa tela inicial você pode criar um repositório, procurar seus repositórios, ver o seu perfil e procurar por perfis ou repositórios de outros usuários usando a barra de busca.

4.3.1 Criação de repositórios

Para criar um repositório selecione botão verde escrito **Create repository** que está no canto superior esquerdo da tela inicial. Após isso, a tela abaixo será mostrada e você poderá fazer as configurações iniciais do seu repositório para se enquadrar da melhor forma possível ao seu projeto.

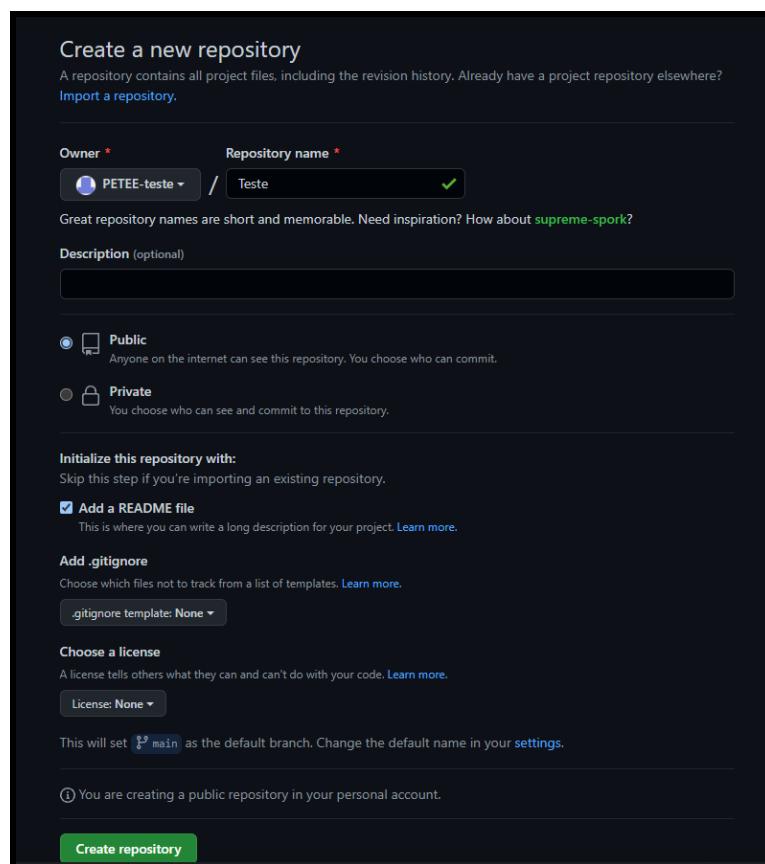


Figura 4.3.2: Tela para criação de repositórios

Com todo o formulário preenchido é necessário apenas selecionar **Create repository** novamente para o repositório ser criado.

4.3.2 Gerenciamento de repositórios

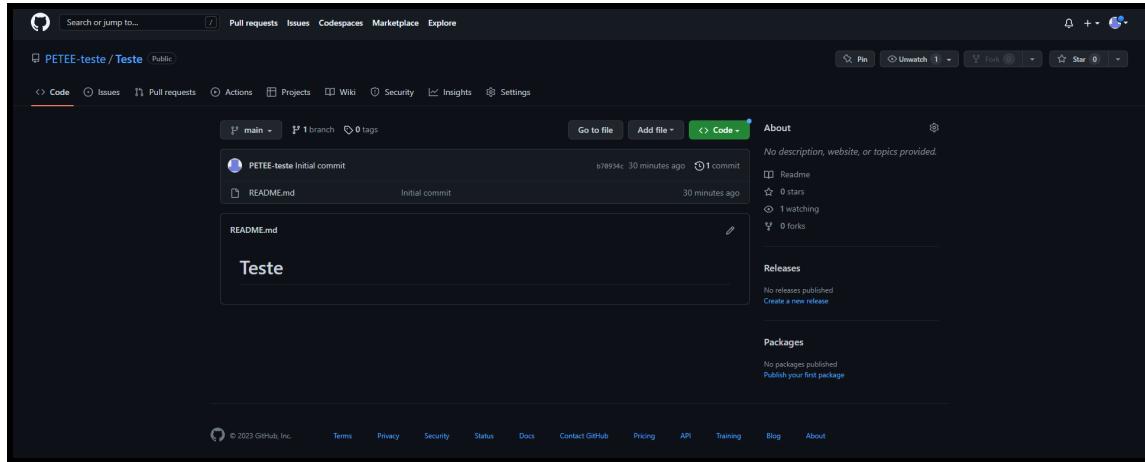


Figura 4.3.3: Tela para gerenciamento do repositório

Após a criação do repositório você pode gerenciá-lo por meio de uma interface bem intuitiva, sendo as abas de funcionalidades mais utilizadas: Code, Issues, Pull request e Settings.

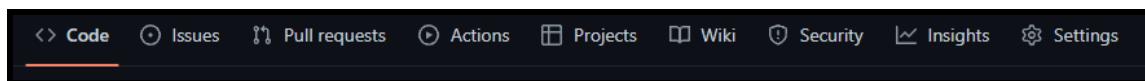


Figura 4.3.4: Abas de funcionalidades

Na aba **Code** é onde se faz a edição e cópia dos arquivos do repositório, se caso ela não queira usar o terminal ou alguma outra ferramenta interativa. Porém não é aconselhável fazer isso, porque existem formas mais rápidas de fazer o mesmo processo.

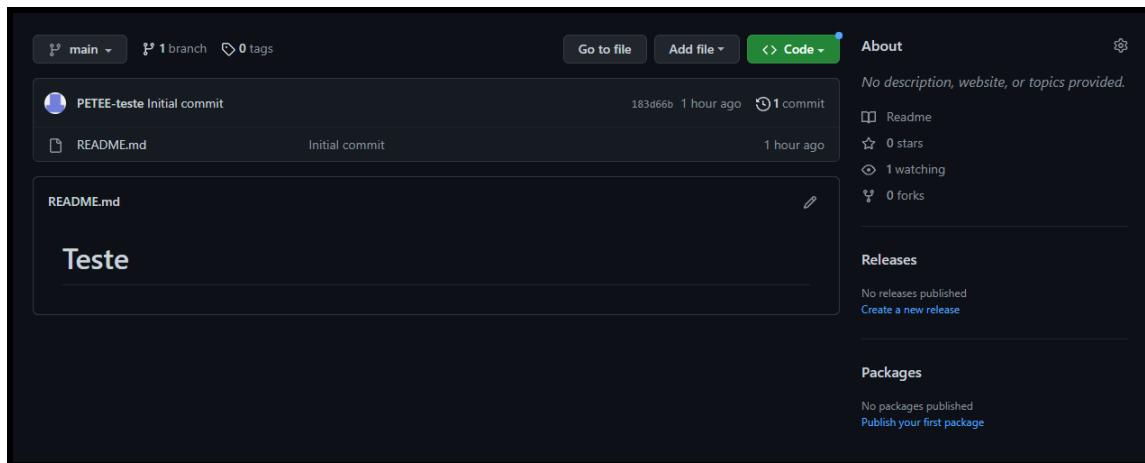


Figura 4.3.5: Code

Em **Issues** é onde se pode fazer sugestões para a melhoria do projeto e também ver sugestões de outras pessoas. Uma coisa interessante é que se o repositório for público pessoas de fora do projeto podem fazer sugestões.

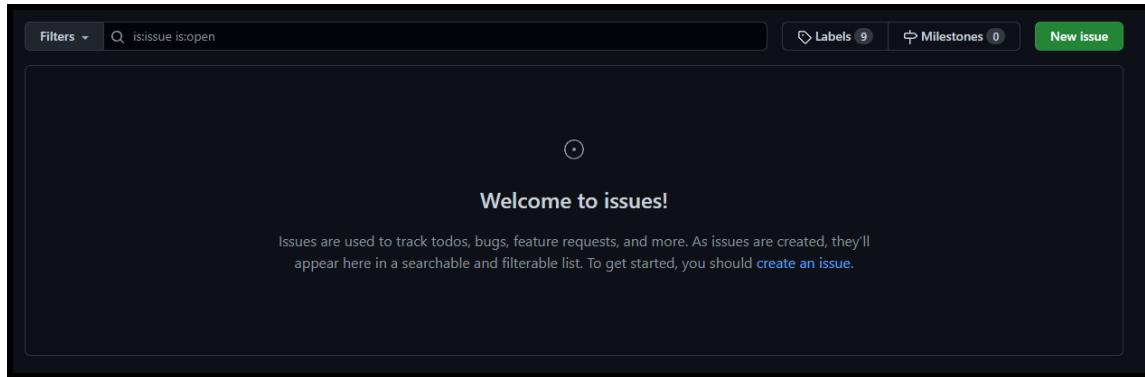


Figura 4.3.6: *Issues*

Em **Pull request** é onde se fazem requisições de merge para alguma branch. A partir dessas requisições, a equipe pode revisar o código antes de fazer o merge, assim, evitando muitos problemas. Além disso, nessa aba também ficam as requisições que outros usuários fizeram.

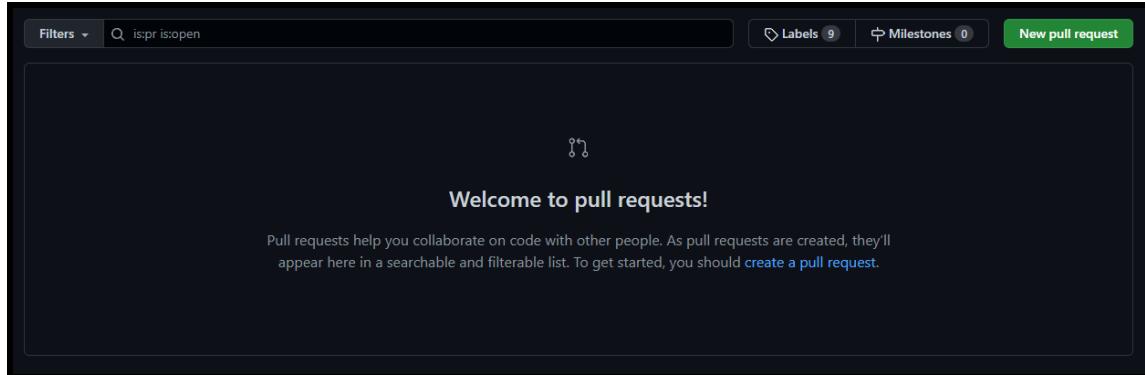


Figura 4.3.7: *Pull request*

Em **Settings** é onde são feitas as alterações nas configurações do repositório como, por exemplo, mudar o nome e a foto, gerenciar o acesso ao seu repositório, deletá-lo e entre outras coisas.

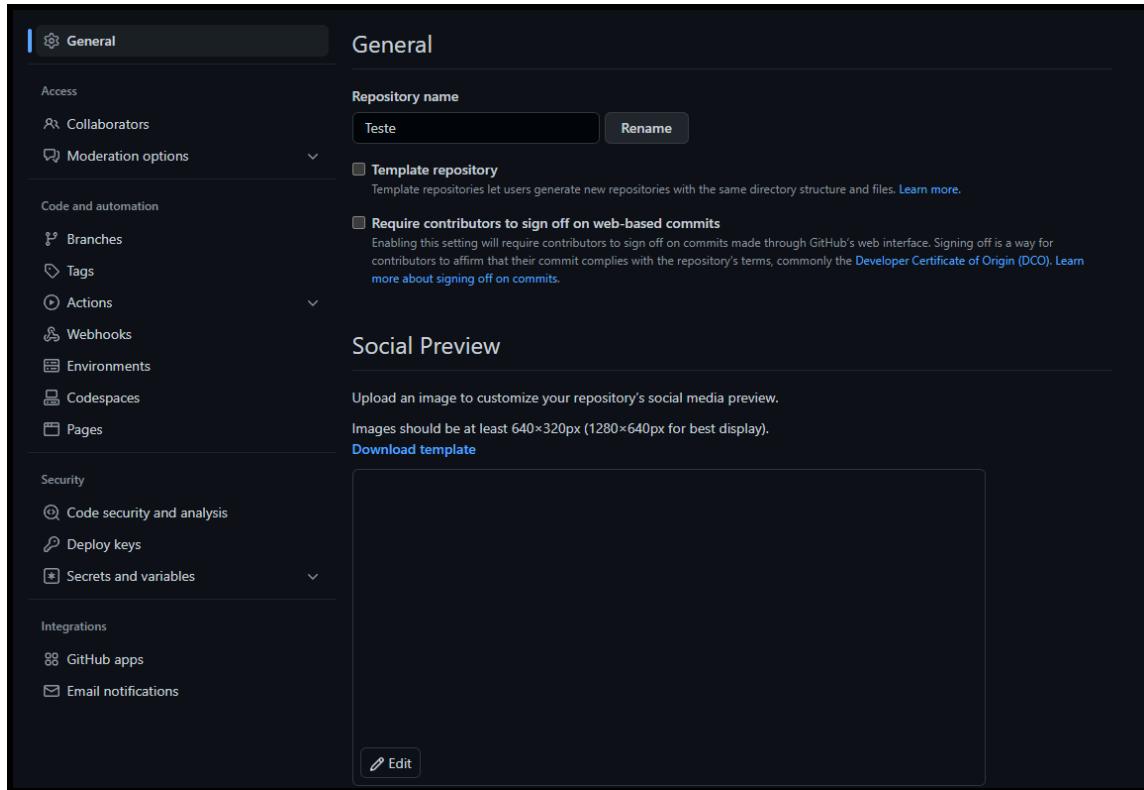


Figura 4.3.8: *Settings*

4.3.3 Perfil

Para acessar o seu perfil é preciso selecionar o ícone com sua foto de perfil que está no canto superior direito da tela inicial e depois selecionar **Your Profile**. Assim, será mostrado a página de perfil abaixo:

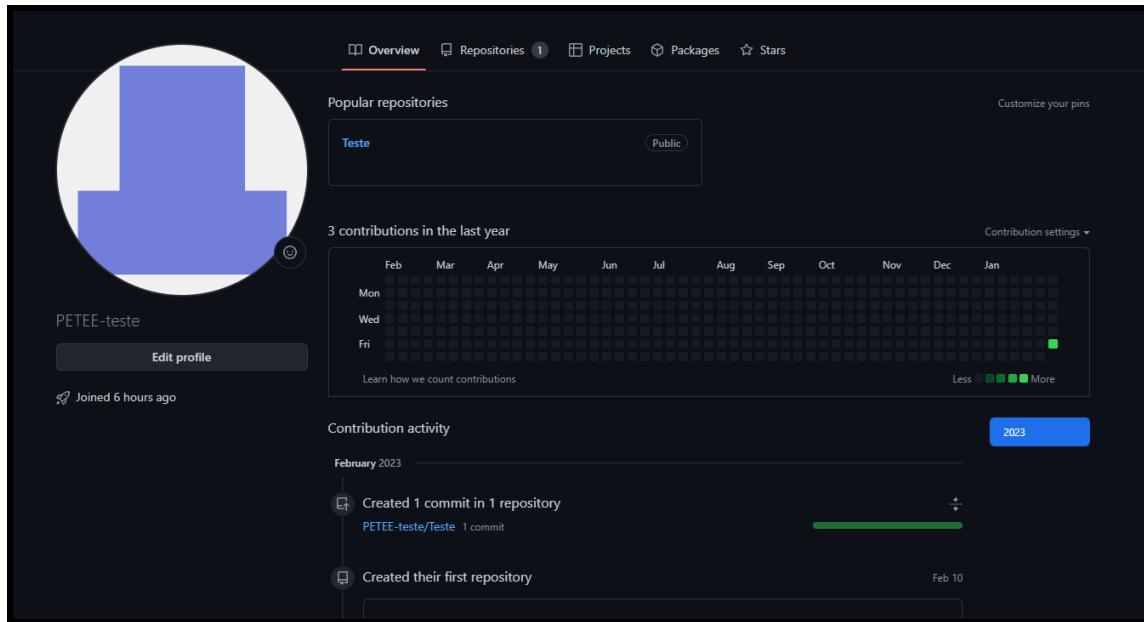
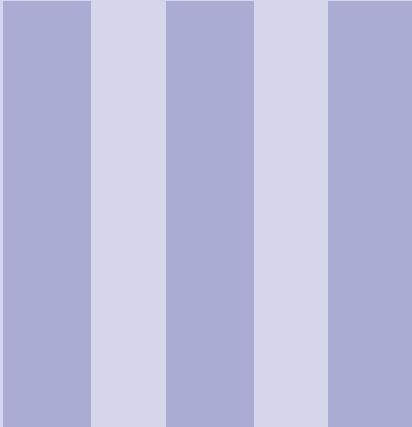


Figura 4.3.9: *Página de perfil*

Nessa página é possível editar seu perfil adicionando uma foto, um nome, uma bio, local em que trabalha, onde mora, seu próprio website e suas redes sociais. Além disso, é possível ver seus repositórios e o seu histórico de atividades no GitHub.



Exercícios

5	Configurando o Git Exercises	41
5.1	Configurações Iniciais	
5.2	Comandos do Git Exercises	
6	Exercícios	43
7	Dicas	47
8	Gabaritos	49

```
santiagofdavi@sANTIAGO-LAPTOP:~$ sudo apt install git
[sudo] password for santiagofdavi:
Reading package lists... Done
Building dependency tree
Reading state information... Done
Suggested packages:
  git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk gitweb git-cvs git-mediawiki git-svn
The following packages will be upgraded:
  git
1 upgraded, 0 newly installed, 0 to remove and 158 not upgraded.
Need to get 4557 kB of archives.
After this operation, 41.0 kB of additional disk space will be used.
Ign:1 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 git amd64 1:2.25.1-1ubuntu3.5
Err:1 http://security.ubuntu.com/ubuntu focal-updates/main amd64 git amd64 1:2.25.1-1ubuntu3.5
  404  Not Found [IP: 185.125.190.39 80]
E: Failed to fetch http://security.ubuntu.com/ubuntu/pool/main/g/git/git_2.25.1-1ubuntu3.5_amd64.deb  404  Not Found [IP: 185.125.190.39 80]
E: Unable to fetch some archives, maybe run apt-get update or try with --fix-missing?
```

5. Configurando o Git Exercises

O objetivo principal dessa seção é incentivar a prática dos tópicos discutidos no capítulo 3 (Comandos Básicos). Os exercícios sugeridos encontram-se no site Git Exercises (<https://gitexercises.fracz.com/>). Há 23 exercícios disponíveis para o aluno, sendo recomendados, segundo o conteúdo dessa apostila, a realização mínima dos oito primeiros.

5.1 Configurações Iniciais

Por início, com o Git instalado, abra o seu terminal ou prompt de comando e, substituindo coerentemente os campos entre aspas pelos seus dados próprios, execute os seguintes comandos:

```
$ git clone https://gitexercises.fracz.com/git/exercises.git
$ cd exercises
$ git config user.name "Your name here"
$ git config user.email "Your e-mail here"
$ ./configure.sh
$ git start
```

Lembre-se de seus dados cadastrados. Estes são fundamentais para acompanhar o progresso nas atividades através do campo no site:



Figura 5.1.1: Para ver seu progresso, basta digitar seu e-mail cadastrado e pressionar "Progress".

5.2 Comandos do Git Exercises

5.2.1 Executando os exercícios

Para iniciar um exercício, substitua **nome-do-exercicio** pelo respectivo nome do exercício desejado e execute o comando em seu terminal:

```
$ git start nome-do-exercicio
```

5.2.2 Corrigindo uma atividade

Uma vez finalizada uma atividade, basta executar o seguinte comando em seu terminal:

```
$ git verify
```

Assim, a atividade será corrigida e o êxito ou falha exibido na tela.

ATENÇÃO! Os comandos 'git start' e 'git verify' não pertencem ao Git como um todo, apenas à plataforma Git Exercises.

5.2.3 Executando exercícios em sequência

Uma vez iniciadas as atividades, pode-se iniciar imediatamente a atividade subsequente após a conclusão e correção da anterior. Para tal, basta executar o comando:

```
$ git start next
```

```
santiago@SANTIAGO-LAPTOP:~$ sudo apt install git  
[sudo] password for santiagofdavi:  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
Suggested packages:  
  git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk gitweb git-cvs git-mediawiki git-svn  
The following packages will be upgraded:  
  git  
1 upgraded, 0 newly installed, 0 to remove and 158 not upgraded.  
Need to get 4557 kB of archives.  
After this operation, 41.0 kB of additional disk space will be used.  
Ign:1 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 git amd64 1:2.25.1-1ubuntu3.5  
Err:1 http://security.ubuntu.com/ubuntu focal-updates/main amd64 git amd64 1:2.25.1-1ubuntu3.5  
  404  Not Found [IP: 185.125.190.39 80]  
E: Failed to fetch http://security.ubuntu.com/ubuntu/pool/main/g/git/git_2.25.1-1ubuntu3.5_amd64.deb  404  Not Found [IP:  
  : 185.125.190.39 80]  
E: Unable to fetch some archives, maybe run apt-get update or try with --fix-missing?
```

6. Exercícios

1. Push a commit you have made (master)

O primeiro exercício é fazer um push de um commit que é criado quando o comando seguinte é executado:

```
$ git start
```

Apenas execute o próximo comando depois de inicializar os exercícios e você finalizou o primeiro!

```
$ git verify
```

2. Commit one file (commit-one-file)

Existem dois arquivos criados no diretório raiz do projeto (A.txt e B.txt). A tarefa é fazer o commit de apenas um deles.

NOTA Lembre-se de que você deve submeter suas soluções com o comando `git verify` em vez do `git push`.

3. Commit one file of two currently staged (commit-one-file-staged)

Existem dois arquivos criados no diretório raiz do projeto (A.txt e B.txt). Ambos estão adicionados à área de staging. A tarefa é fazer o commit de apenas um deles.

4. Ignore unwanted files (ignore-them)

É sempre uma boa ideia dizer ao Git quais arquivos ele deve processar e quais não. Desenvolvedores quase sempre não querem incluir arquivos gerados, código compilado ou bibliotecas no histórico do projeto. O objetivo é criar e fazer o commit com uma configuração que ignore arquivos das extensões exe, o e jar; além de ignorar também todo o diretório libraries.

5. Chase branch that escaped (chase-branch)

Você está atualmente no ramo **chase-branch**. Também há o ramo **escaped** que tem mais dois commits.

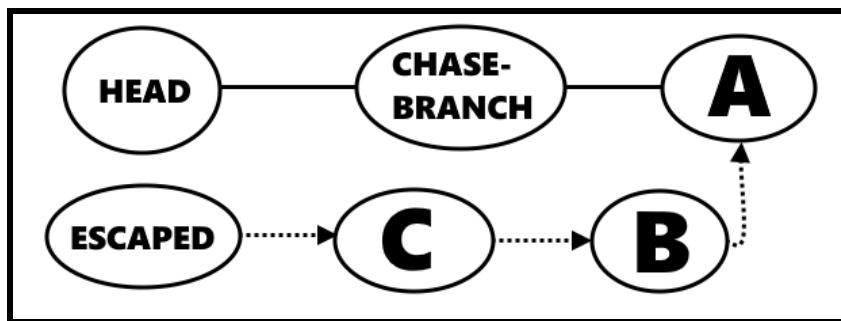


Figura 6.0.1: Situação atual

Você quer fazer **chase-branch** apontar para o mesmo commit que o ramo **escaped**.

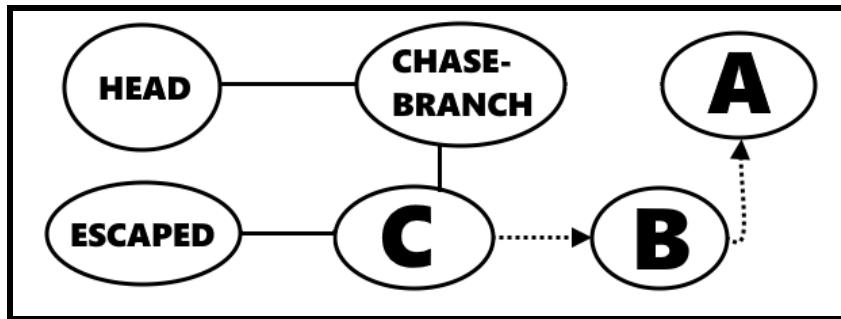


Figura 6.0.2: Situação desejada

6. Resolve a merge conflict (merge-conflict)

O merge conflict aparece quando você muda a mesma parte de um mesmo arquivo de forma diferente em dois ramos que você está fundindo. Os conflitos exigem que o programador os resolva manualmente. Seu repositório tem essa aparência:

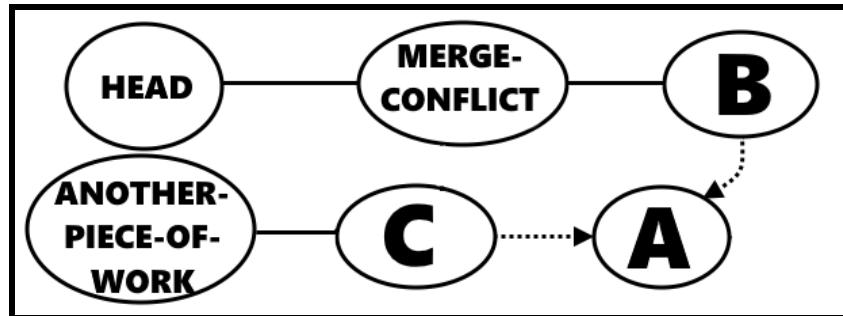


Figura 6.0.3: Situação atual

Você quer fundir **Another-piece-of-work** com o ramo atual. Isso vai gerar um merge conflict que você deve resolver. Seu repositório deve se parecer com isso:

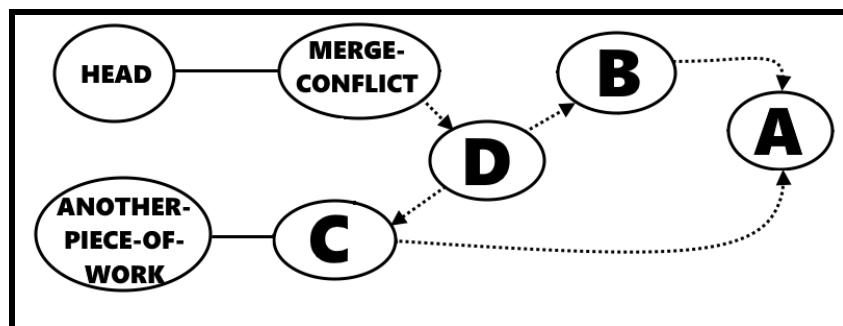


Figura 6.0.4: Situação desejada

7. Saving your work (save-your-work)

Você está trabalhando duro num problema regular no momento em que seu chefe chega e pede que você conserte um defeito. O estado da sua área trabalho atual está bagunçado e você ainda não pretende realizar um commit. Contudo, você precisa urgentemente consertar o defeito.

O Git permite que você deixe um trabalho de lado e o continue mais tarde. Encontre as ferramentas apropriadas do git e as use para manipular a situação corretamente.

Encontre o defeito e o remova em *bug.txt*.

Depois de realizar o commit em **bugfix**, retorne ao seu trabalho original. Termine-o adicionando adicionando uma nova linha em *bug.txt* com:

```
Finally, finished it!
```

Depois, realize um commit no seu trabalho após **bugfix**.

8. Change branch history (change-branch-history)

Você estava trabalhando num problema regular quando seu chefe chegou e te pediu para consertar um defeito recente num aplicativo. Como seu trabalho sobre o problema ainda não foi concluído, você decidiu voltar para onde começou e fazer uma correção de bug lá. Seu repositório tem essa aparência:

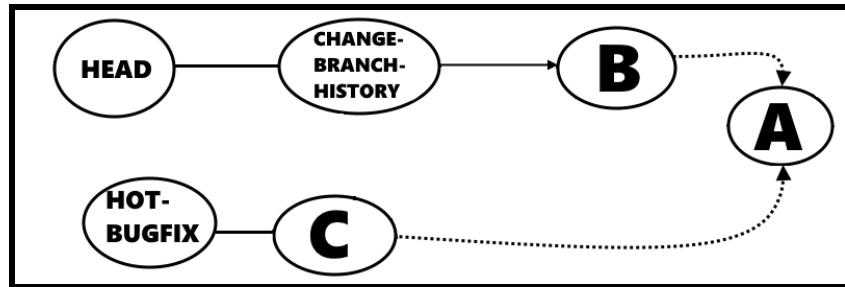


Figura 6.0.5: Situação atual

Agora você percebeu que o defeito é realmente irritante e não quer continuar seu trabalho sem a correção que fez. Você gostaria que seu repositório parecesse que você começou depois de corrigir um defeito.

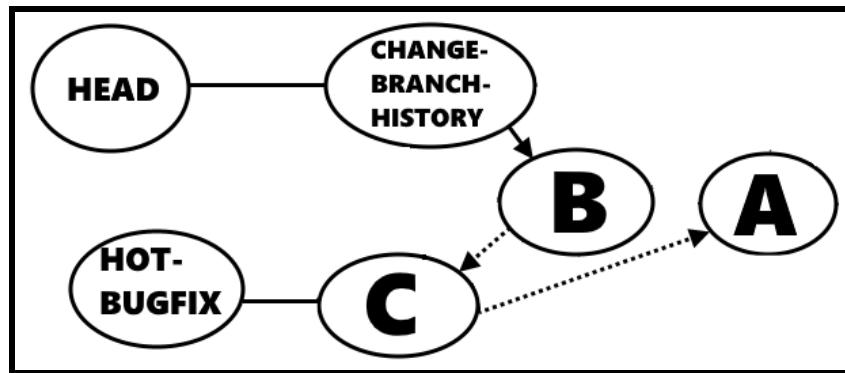


Figura 6.0.6: Situação desejada

Consiga isso.

```
santiagofdavi@sANTIAGO-LAPTOP:~$ sudo apt install git  
[sudo] password for santiagofdavi:  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
Suggested packages:  
  git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk gitweb git-cvs git-mediawiki git-svn  
The following packages will be upgraded:  
  git  
1 upgraded, 0 newly installed, 0 to remove and 158 not upgraded.  
Need to get 4557 kB of archives.  
After this operation, 41.0 kB of additional disk space will be used.  
Ign:1 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 git amd64 1:2.25.1-1ubuntu3.5  
Err:1 http://security.ubuntu.com/ubuntu focal-updates/main amd64 git amd64 1:2.25.1-1ubuntu3.5  
  404  Not Found [IP: 185.125.190.39 80]  
E: Failed to fetch http://security.ubuntu.com/ubuntu/pool/main/g/git/git_2.25.1-1ubuntu3.5_amd64.deb  404  Not Found [IP:  
  : 185.125.190.39 80]  
E: Unable to fetch some archives, maybe run apt-get update or try with --fix-missing?
```

7. Dicas

Caso você encontre dificuldades para concluir os exercícios, consulte essa seção antes de ir direto ao gabarito e tente novamente.

1. Push a commit you have made (master)

Siga as instruções do próprio exercício.

2. Commit one file (commit-one-file)

Precisa-se adicionar os arquivos escolhidos para a área de staging com o comando **git add** e então fazer o commit com o comando **git commit**.

3. Commit one file of two currently staged (commit-one-file-staged)

Para remover arquivos da área de staging deve-se usar o comando **git reset**. Arquivos que não estão na área de staging não serão processados no próximo commit.

4. Ignore unwanted files (ignore-them)

Faz-se necessária a criação de um arquivo **.gitignore**, inserir as regras para ignorar os arquivos e, então, fazer o commit.

5. Chase branch that escaped (chase-branch)

Faz-se necessário o uso do comando **git merge**.

6. Resolve a merge conflict (merge-conflict)

Quando executado o comando **git merge**, haverá um conflito. Pode-se (e deve-se) resolver com as mãos. Então salve as mudanças no seu arquivo, adicione à área de staging e, então, faça o commit.

7. Saving your work (save-your-work)

- I. Use o comando **git stash** para salvar seu trabalho atual no plano de fundo e limpar a área de trabalho.
- II. Conserte o problema.
- III. Use o comando **git stash pop** para reaplicar suas mudanças à área de trabalho.
- IV. Termine seu trabalho.

8. Change branch history (change-branch-history)

Faz-se necessário o uso do comando **git rebase**.

```
santiagofdavi@sANTIAGO-LAPTOP:~$ sudo apt install git  
[sudo] password for santiagofdavi:  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
Suggested packages:  
  git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk gitweb git-cvs git-mediawiki git-svn  
The following packages will be upgraded:  
  git  
1 upgraded, 0 newly installed, 0 to remove and 158 not upgraded.  
Need to get 4557 kB of archives.  
After this operation, 41.0 kB of additional disk space will be used.  
Ign:1 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 git amd64 1:2.25.1-1ubuntu3.5  
Err:1 http://security.ubuntu.com/ubuntu focal-updates/main amd64 git amd64 1:2.25.1-1ubuntu3.5  
  404  Not Found [IP: 185.125.190.39 80]  
E: Failed to fetch http://security.ubuntu.com/ubuntu/pool/main/g/git/git_2.25.1-1ubuntu3.5_amd64.deb  404  Not Found [IP:  
  : 185.125.190.39 80]  
E: Unable to fetch some archives, maybe run apt-get update or try with --fix-missing?
```

8. Gabaritos

1. Push a commit you have made (master)

```
$ git start  
$ git verify
```

2. Commit one file (commit-one-file)

```
$ git add A.txt  
# or git add B.txt  
$ git commit -m "add one file"  
$ git verify
```

3. Commit one file of two currently staged (commit-one-file-staged)

```
$ git reset HEAD A.txt  
# or git reset HEAD B.txt  
$ git commit -m "destage one file"  
$ git verify
```

4. Ignore unwanted files (ignore-them)

```
$ nano .gitignore
```

```
*.exe  
*.o  
*.jar  
libraries/
```

```
$ git add .  
$ git commit -m "commit useful files"  
$ git verify
```

5. Chase branch that escaped (chase-branch)

```
$ git checkout chase-branch  
$ git merge escaped  
$ git verify
```

6. Resolve a merge conflict (merge-conflict)

```
$ git checkout merge conflict  
$ git merge another-piece-of-work  
$ nano equation.txt
```

```
2 + 3 = 5
```

```
$ git add equation.txt  
$ git commit -m "merge and resolve"  
$ verify
```

7. Saving your work (save-your-work)

```
$ git stash
# or git stash push
$ nano bug.txt
# in the text editor delete the bud line
$ git commit -am "remove bug"
$ git stash apply
# or git stash apply stash@{0}
$ nano bug.txt
# in the text editor add the line "Finally, finished it!"
    to the end
$ git commit -am "finish"
$ git verify
```

8. Change branch history (change-branch-history)

```
$ git checkout change-branch-history
$ git rebase hot-bugfix
$ git verify
```

Apêndice



9	GitKraken	53
9.1	Função	
9.2	Instalação	
9.3	Integração com o GitHub	
9.4	Utilização	



9. GitKraken

9.1 Função

O GitKraken é uma interface gráfica para Git que permite observar de forma intuitiva e mais simples o status atual do repositório, os commits e branches. Para fazer um commit, por exemplo, é possível verificar quais mudanças foram feitas em relação à versão anterior, com linhas vermelhas representando o código anteriormente, e linhas verdes representando o código atual, e somente clicar num botão para criar uma nova versão.

9.2 Instalação

O GitKraken pode ser adquirido gratuitamente através do site www.gitkraken.com.

9.3 Integração com o GitHub

É possível conectar o GitKraken ao GitHub e facilmente acessar e clonar os repositórios remotos ou fazer um fork de um repositório já existente. Outros serviços de hospedagem como BitBucket, GitLab e Azure DevOps também estão disponíveis para conexão.

Com essa conexão, facilita-se a trazer as novas versões do código simplesmente selecionando **Pull**, ou enviar as alterações do código clicando em **Push**.

9.4 Utilização

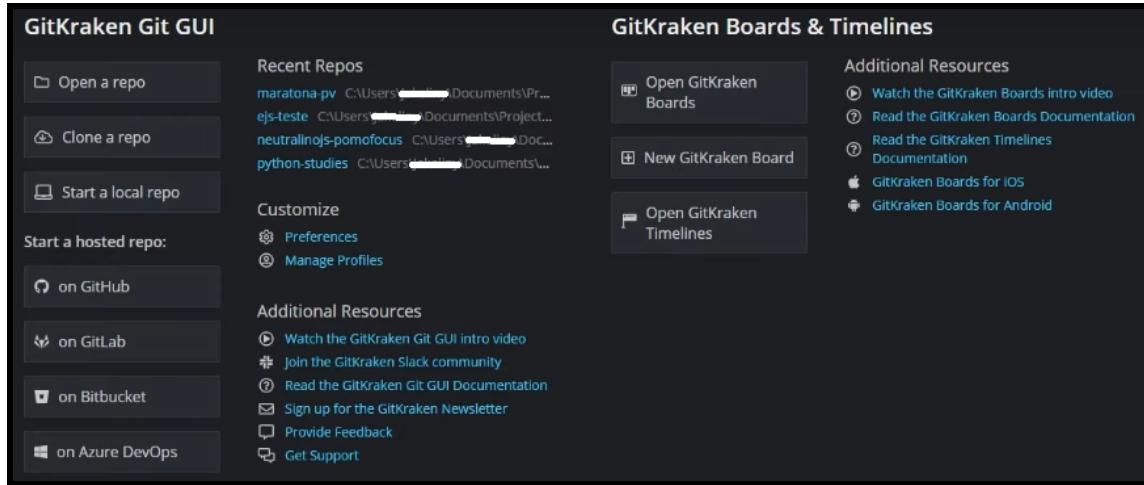


Figura 9.4.1: Tela inicial Gitkraken

Na tela inicial, é possível abrir um repositório já clonado em **Open a repo**, clonar um repositório em **Clone a repo**, ou começar um novo em **Start a local repo**.

9.4.1 Abrir um repositório

Quando já há um trabalho sendo feito, é possível abri-lo em **Open a repo**.

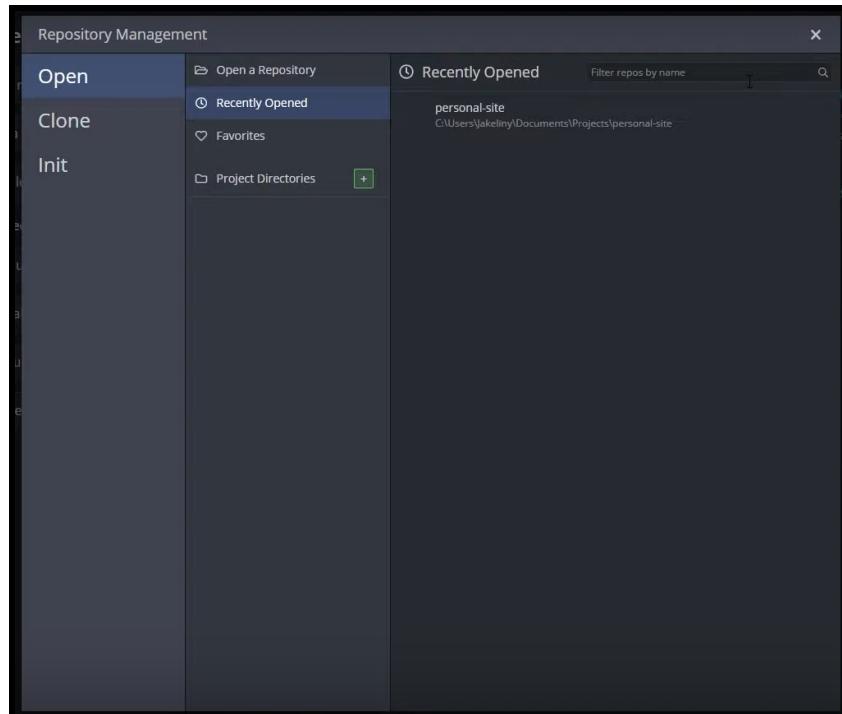


Figura 9.4.2: Abrindo uma pasta

DICA Um outro modo bastante simples de abrir um projeto no GitKraken é entrando na pasta desejada e, com o botão direito, selecionar "abrir com GitKraken"

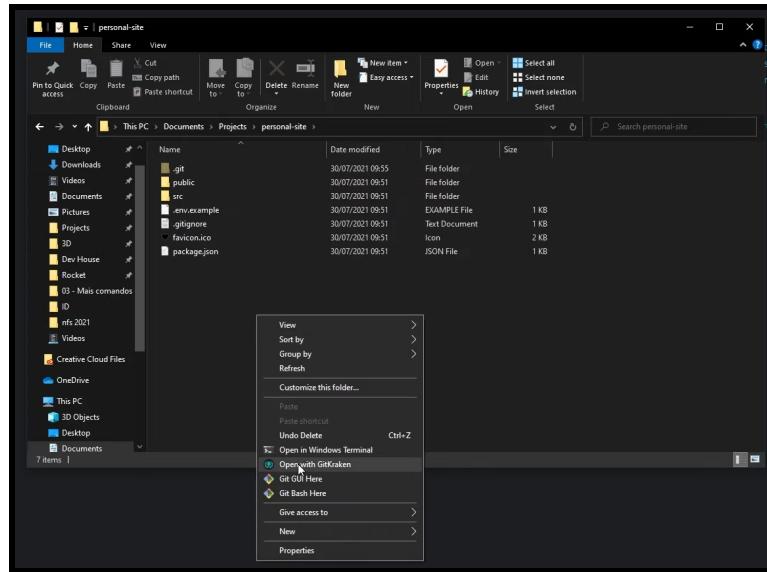


Figura 9.4.3: Criando um repositório local

9.4.2 Clonar repositório

Depois de selecionar **Clone a repo**, a tela que aparece é a seguinte:

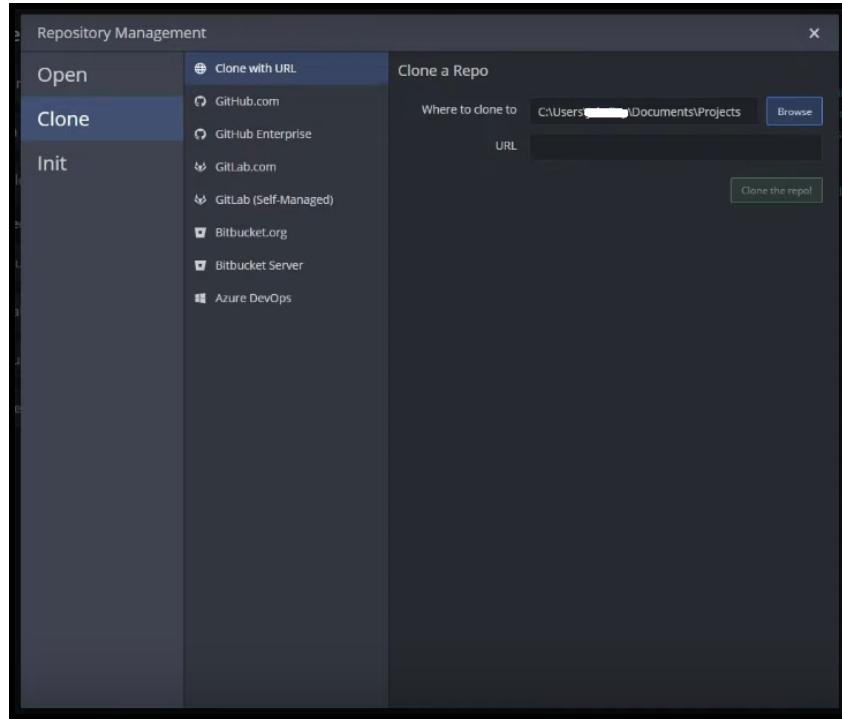


Figura 9.4.4: Clonando o repositório

É possível clonar a partir de URL ou acessando uma plataforma de hospedagem.

9.4.3 Criar repositório

Para iniciar um repositório, seleciona-se **Start a local repo**, a tela que aparece é a seguinte:

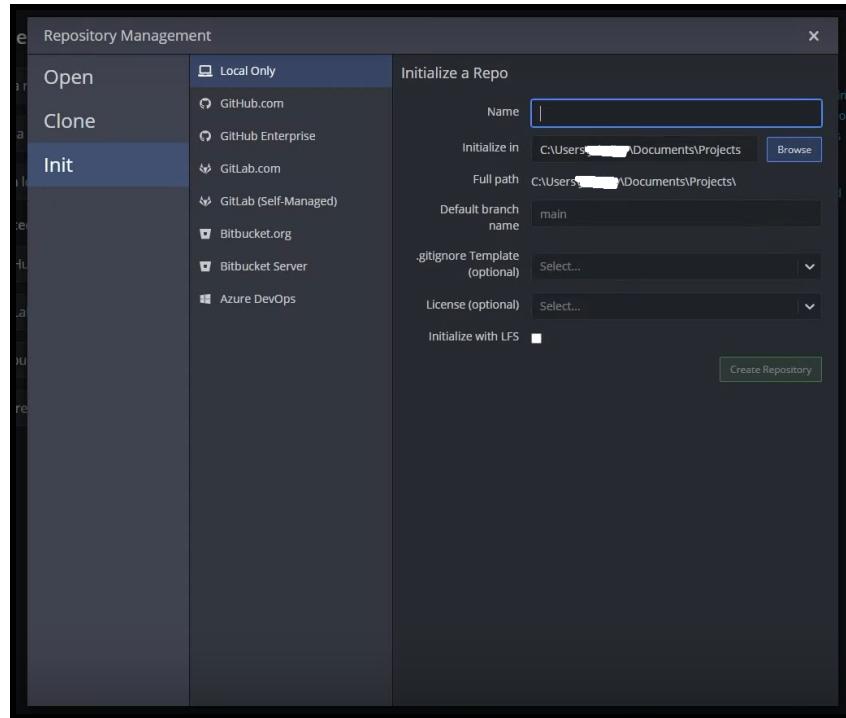


Figura 9.4.5: Criando um repositório local

Pode-se criar um repositório na própria máquina ou em alguma das plataformas de hospedagem.

Índice





Bibliografia

- Chacon, Scott, and Ben Straub. "Pro Git". Springer Nature, 2014.
- Blischak, John D., Emily R. Davenport, and Greg Wilson. "A quick introduction to version control with Git and GitHub." PLoS computational biology 12.1 (2016).



PETEE UFMG

U F *m* G