

Desenvolvimento de Sistemas

Comandos de versionamento: operações e funcionalidades

Introdução

Neste conteúdo, você conhecerá como funciona o processo de versionamento e entender a importância de cada uma dessas etapas. Para isso, serão digitados vários comandos na interface de linha de comando, para ver, na prática, como se trabalha com a ferramenta Git. Então, para aproveitar melhor este conteúdo, certifique-se de já ter instalado a ferramenta GitBash no seu computador.

Após compreender o ciclo de versionamento, você conhecerá algumas possíveis expansões dos comandos principais que aprenderá. Além disso, também aprenderá como realizar o versionamento de *software* utilizando ferramentas com interfaces gráficas, neste caso, o GitHub Desktop. Recomenda-se que você já tenha também essa ferramenta instalada.

Você pode consultar o conteúdo **Repositórios remotos e sistemas de gerenciamento de configuração** para informações sobre instalação e primeiros passos com GitHub Desktop e GitBash.

Operações e funcionalidades

Para exemplificar o versionamento de um projeto Java, será criado um novo projeto no Apache NetBeans IDE (*integrated development environment*, em português ambiente de desenvolvimento integrado) e, dentro da pasta do projeto, será iniciado o versionamento. Para exemplo, o projeto será chamado de **GitExemplo**.

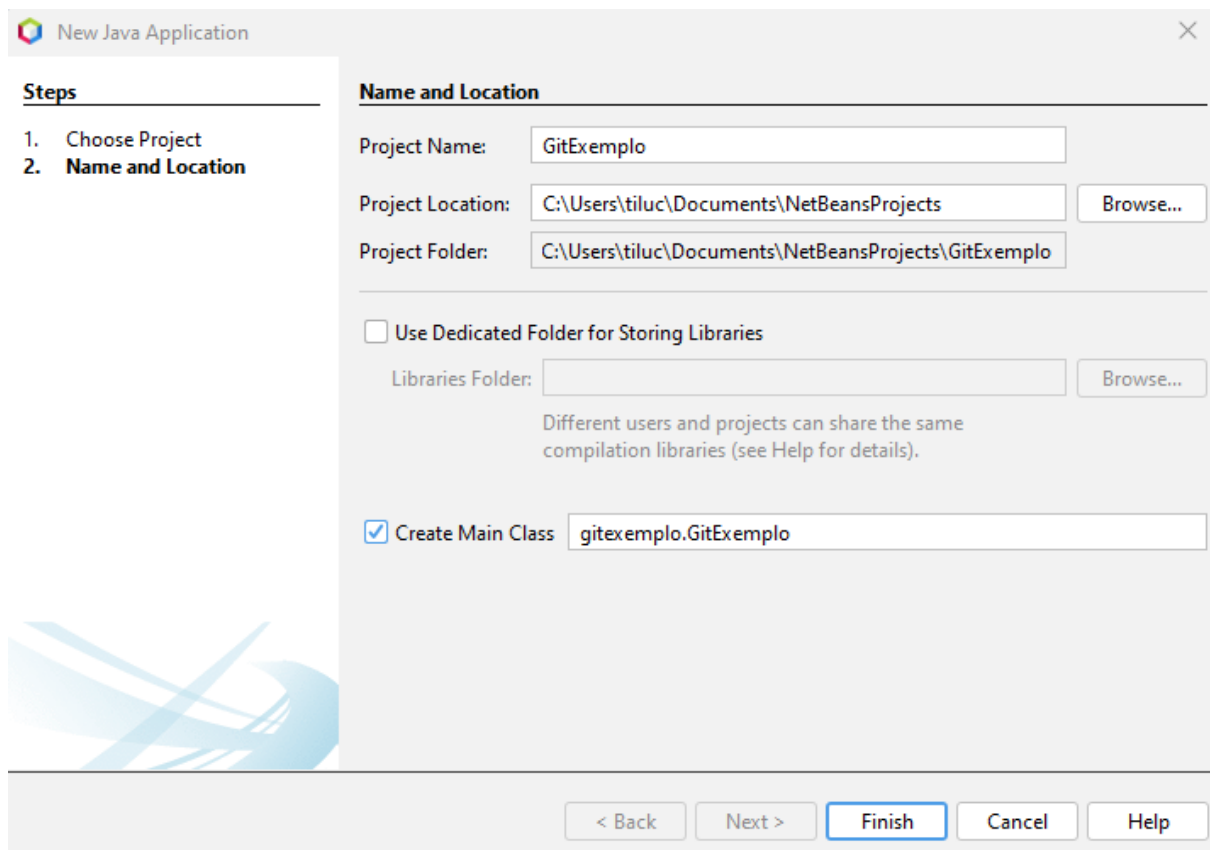


Figura 1 – Criação do projeto **GitExemplo** no NetBeans

Fonte: Apache NetBeans IDE (2023)

Durante a criação do projeto, lembre-se de verificar o local onde o projeto será criado por meio do campo **Project Location**.

Após criar o projeto, acesse esse diretório e abra o GitBash, clicando com o botão direito do *mouse* e selecionando a opção **GitBash here**.

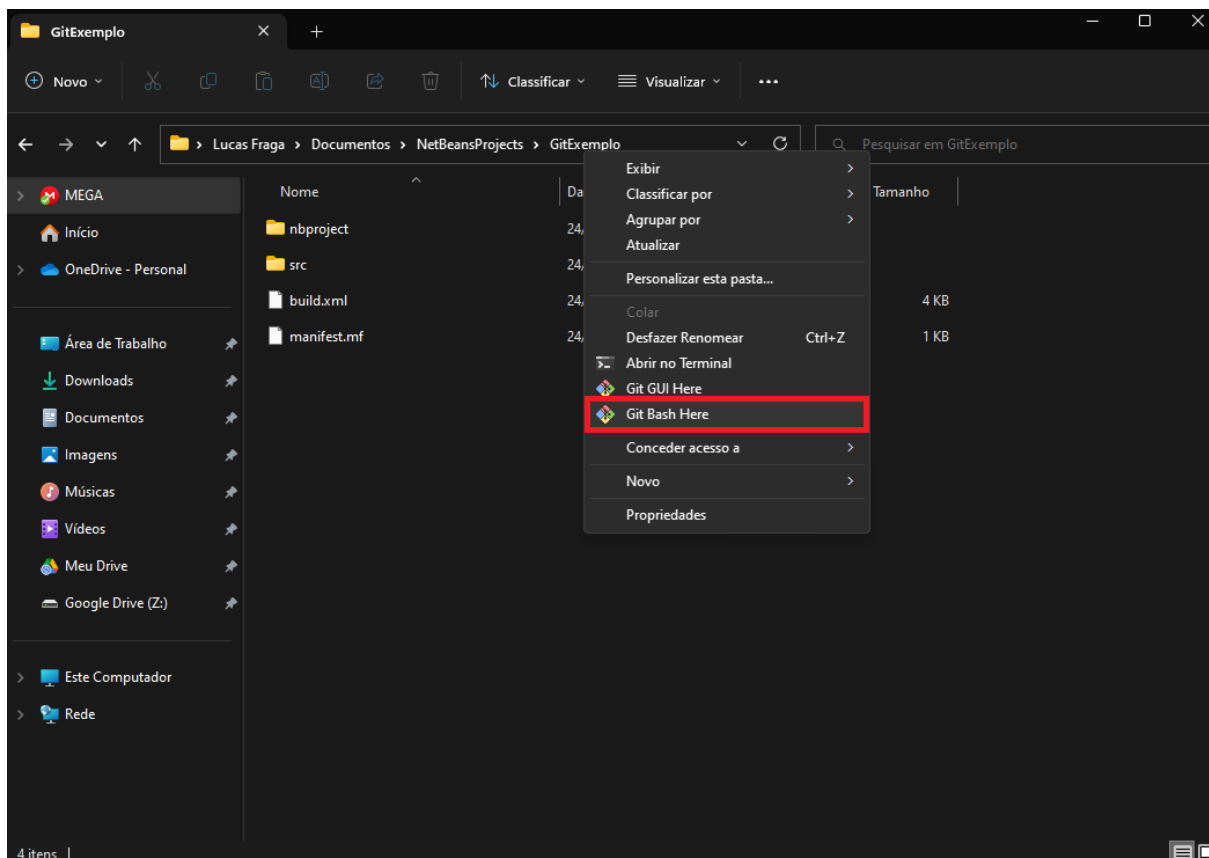


Figura 2 – Acessando o GitBash no diretório do projeto

Fonte: Senac EAD (2023)

Iniciando o versionamento

Um repositório Git rastreia e salva o histórico de todas as alterações feitas nos arquivos em um projeto. Ele salva esses dados em um diretório chamado **.git**, que é responsável por rastrear e salvar o histórico de todas as alterações feitas nos arquivos do diretório.

Para iniciar o versionamento em um diretório, utiliza-se o seguinte comando:

git init

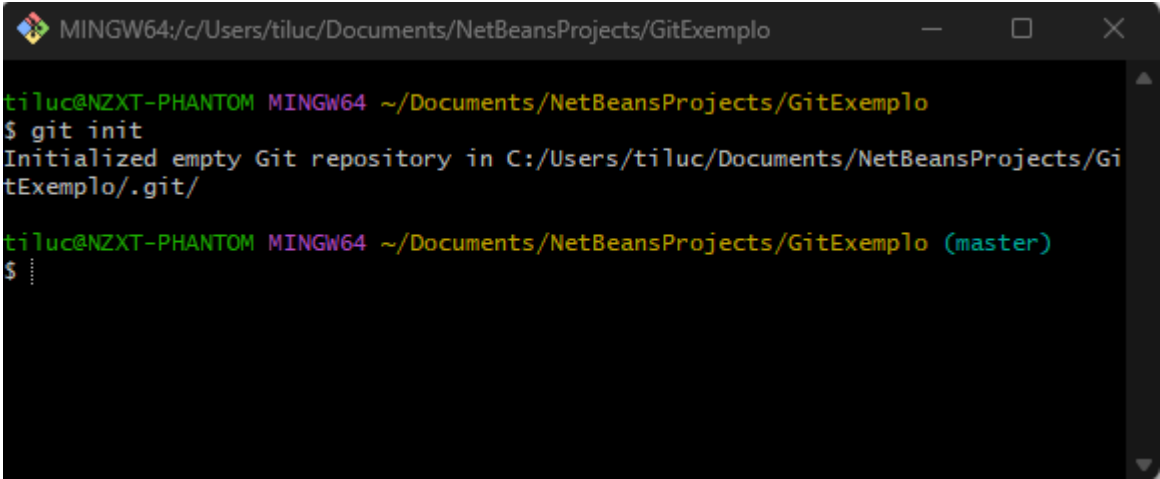


Figura 3 – Terminal do GitBash
Fonte: GitBash (2023)

Após executá-lo, será criada a pasta **.git**.

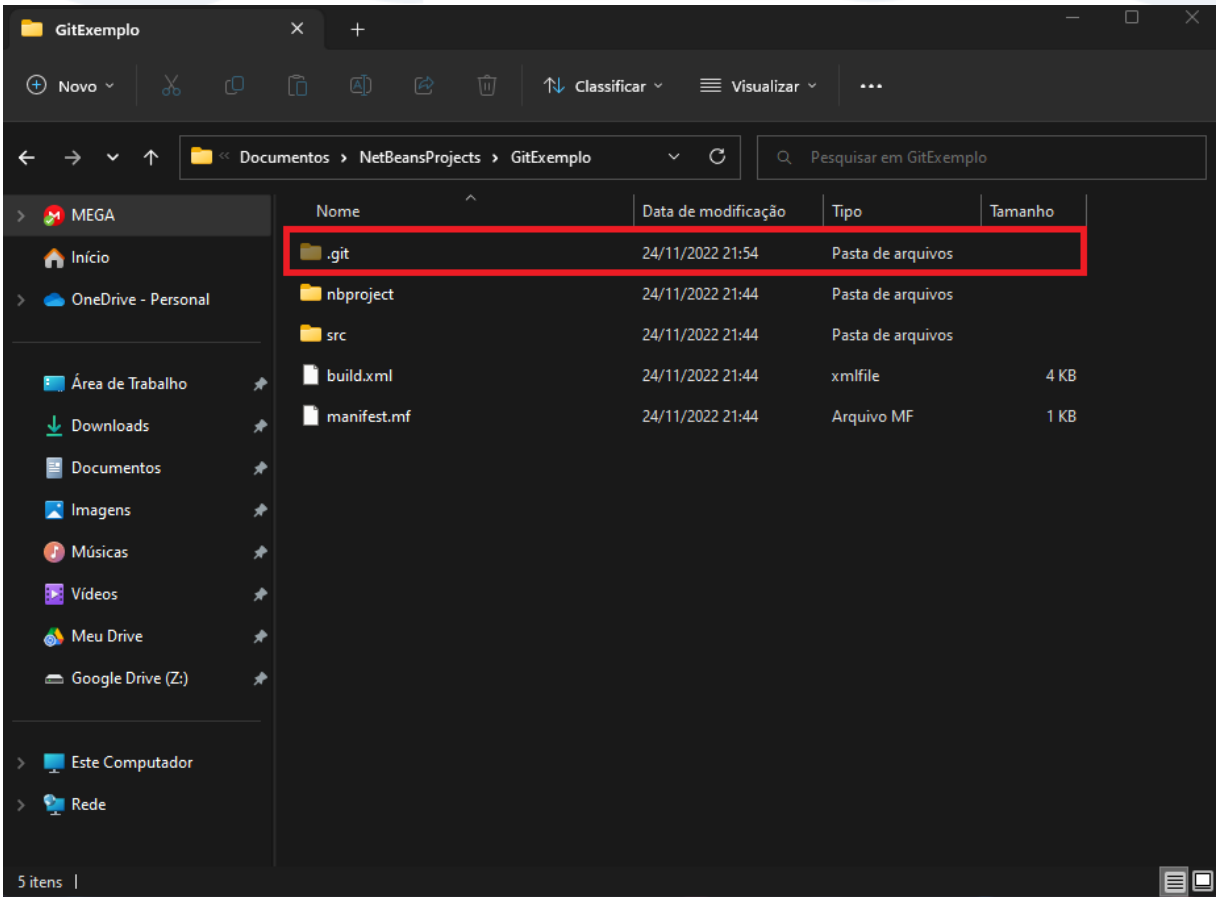


Figura 4 – Pasta do projeto com a pasta **.git**
Fonte: Senac EAD (2023)

Caso a pasta **.git** não apareça no seu diretório, isso pode significar que o seu explorador de arquivos está configurado para não exibir itens ocultos. Se deseja habilitar a exibição de itens ocultos, abra o **Explorador de Arquivos** e selecione **Exibir > Mostrar > Itens ocultos**.

Vale lembrar que a exibição dos itens ocultos não interferirá no versionamento do projeto. Então, caso não deseje habilitar essa opção, você pode ignorá-la.

Configurando o versionamento

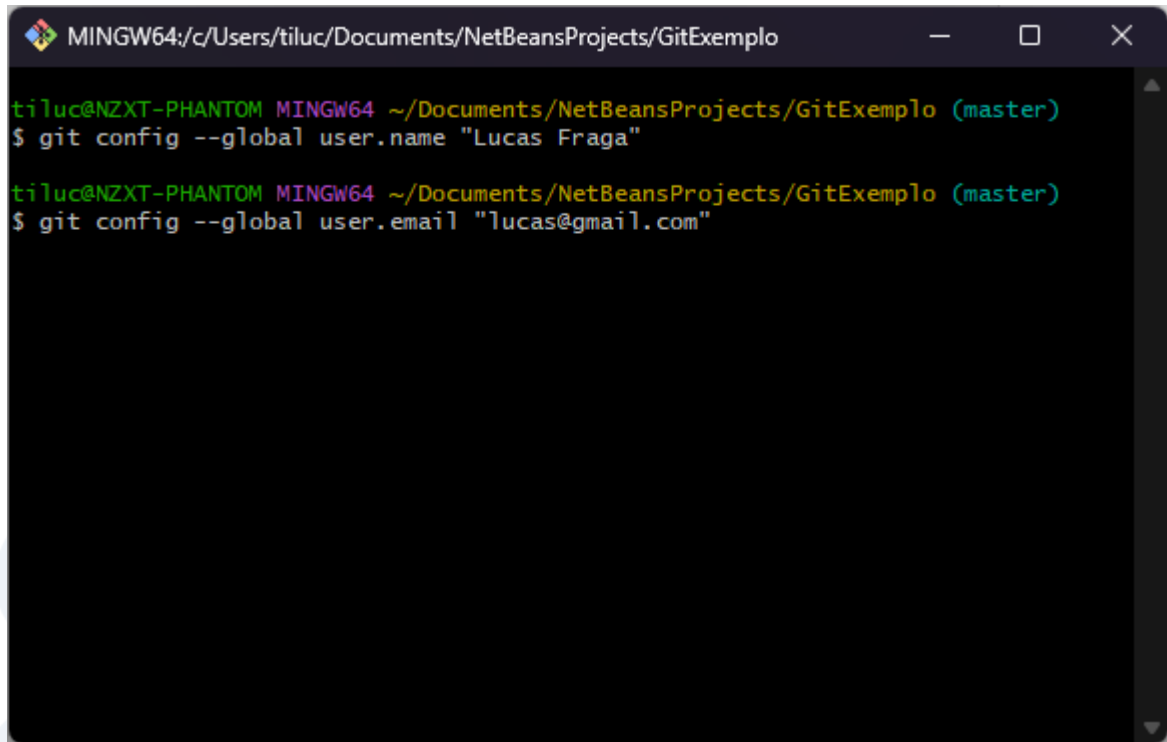
Se é a primeira vez que você está utilizando o GitBash no seu computador, então é necessário configurar o nome e o *e-mail* do seu usuário para que o versionamento reconheça quem é o responsável pelas alterações nos arquivos. Para isso, use dois comandos, um para configurar o nome e outro para configurar o *e-mail*.

Para adicionar um nome de autor do usuário atual:

```
git config --global user.name "Nome Sobrenome"
```

Para adicionar um endereço de *e-mail* a todos os **commits** do usuário atual:

```
git config --global user.email "exemplo@email.com"
```

A screenshot of a Git Bash terminal window. The title bar at the top reads "MINGW64; c:/Users/tiluc/Documents/NetBeansProjects/GitExemplo". The terminal shows two commands being executed: first, "git config --global user.name 'Lucas Fraga'", and second, "git config --global user.email 'lucas@gmail.com'". The prompt is "tiluc@NZXT-PHANTOM MINGW64 ~/Documents/NetBeansProjects/GitExemplo (master)".

```
tiluc@NZXT-PHANTOM MINGW64 ~/Documents/NetBeansProjects/GitExemplo (master)
$ git config --global user.name "Lucas Fraga"

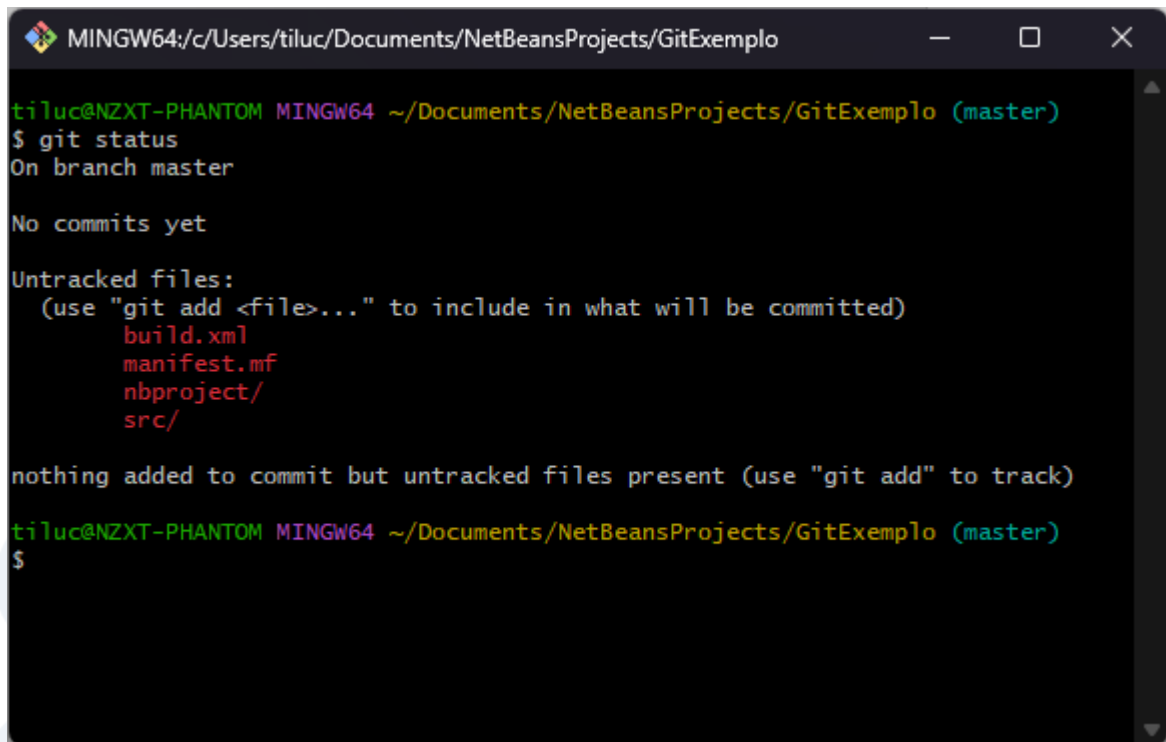
tiluc@NZXT-PHANTOM MINGW64 ~/Documents/NetBeansProjects/GitExemplo (master)
$ git config --global user.email "lucas@gmail.com"
```

Figura 5 – Terminal do GitBash

Fonte: GitBash (2023)

Verificando o *status*: git status

Para verificar o *status* do versionamento, utiliza-se o comando **git status**. O comando é relativamente simples e retorna informações muito úteis sobre o que está acontecendo no versionamento dos arquivos e diretórios. Se o comando **git status** for executado no diretório do projeto, você terá a seguinte saída:



```
MINGW64; c:/Users/tiluc/Documents/NetBeansProjects/GitExemplo

tiluc@NZXT-PHANTOM MINGW64 ~/Documents/NetBeansProjects/GitExemplo (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    build.xml
    manifest.mf
    nbproject/
    src/

nothing added to commit but untracked files present (use "git add" to track)
tiluc@NZXT-PHANTOM MINGW64 ~/Documents/NetBeansProjects/GitExemplo (master)
$
```

Figura 6 – Terminal do GitBash

Fonte: GitBash (2023)

Perceba que, atualmente, o versionamento identificou os arquivos **build.xml** e **manifest.mf** e os diretórios **nbproject** e **src** com o *status* **untracked**, que significa “não rastreado”.

Dentro de um repositório, os seus arquivos e as suas pastas podem ter dois *status*: **untracked** (do inglês, “não rastreado”) e **tracked** (do inglês, “rastreado”).

O *status* **untracked** é o estado inicial de todo arquivo adicionado ou editado. Logo, sempre que você iniciar o versionamento em um repositório local, os arquivos serão reconhecidos com o estado de “não rastreado”.

Sabe-se que o Git é uma ferramenta de rastreamento de versão. Essa ferramenta nunca começará a rastrear um arquivo até que você peça especificamente a ela para fazer isso. Se você fizer alguma alteração nos arquivos, o estado deles permanecerá “não rastreado” até usar o comando para rastrear.

Preparando os arquivos

O comando **git add** adiciona arquivos novos ou alterados em seu diretório à área de preparo do Git, marcando-os para inclusão no próximo rastreamento.

Às vezes, esse comando pode passar a impressão de ser uma etapa desnecessária. Afinal, por que é preciso preparar o rastreamento dos arquivos e só depois realizá-lo de fato? Pois bem. O que faz o comando **git add** tão importante é que, por meio dele, é possível moldar a história sem alterar a forma como se trabalha.

Imagine que, no seu dia a dia como desenvolvedor de *software*, você precise programar uma determinada funcionalidade para um sistema (a qual será chamada de **funcionalidade A**). Por ser algo simples, você aproveitou para fazer a implementação de mais um recurso para adiantar o seu trabalho (**funcionalidade B**).

Após implementar e testar, você deve fazer um relatório de tudo aquilo que você fez. Esse relatório terá todo o registro do que foi feito e desfeito dentro do projeto, incluindo cada linha de código escrita ou modificada. Se algo no sistema projetado der errado, basta seguir o relatório para desfazer tudo que foi feito. Aqui, há as seguintes possibilidades:

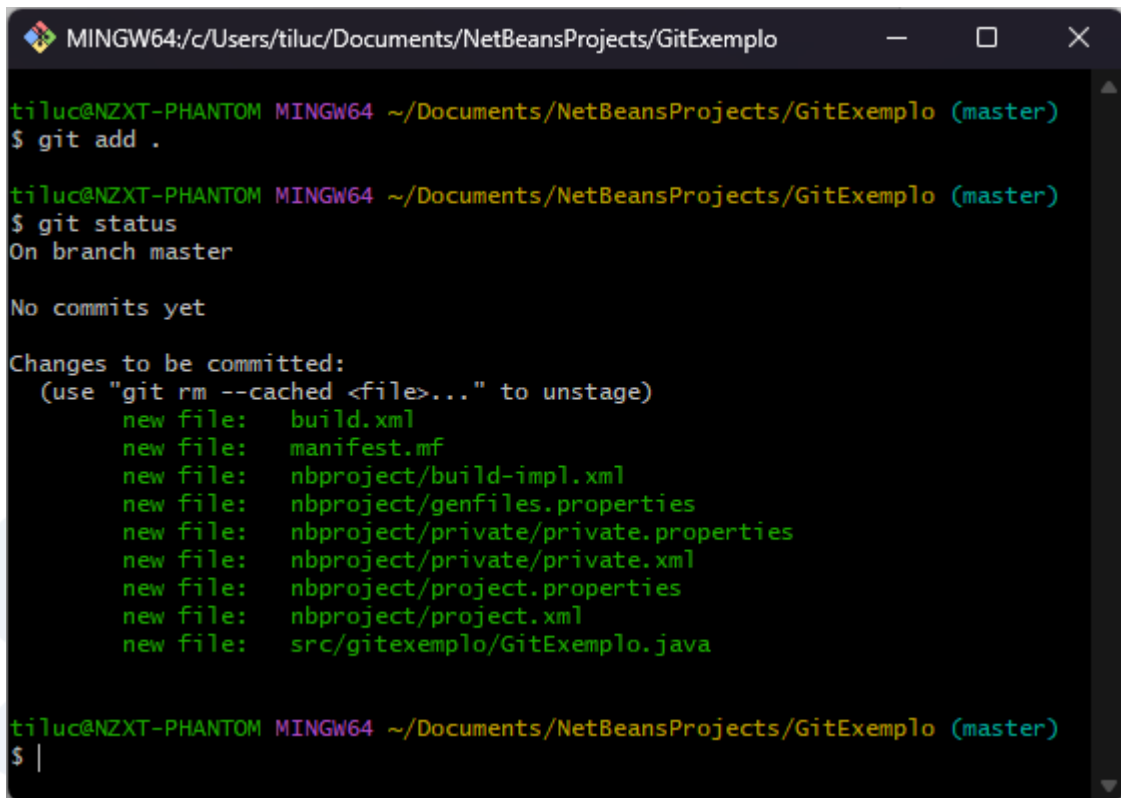
1. Você pode implementar as duas funcionalidades e fazer um único relatório.
2. Você pode implementar a **funcionalidade A** e fazer um relatório; depois pode implementar a **funcionalidade B** e fazer mais um relatório.
3. Você pode implementar as duas funcionalidades e fazer dois relatórios separados.

Se você optar pela opção 1, caso algo dê errado, você terá perdido um dia inteiro de trabalho, pois terá de desfazer tudo o que foi feito caso o sistema apresente algum problema. Se você optou pelas opções 2 e 3, você poderá corrigir o problema do sistema sem afetar a outra funcionalidade implementada. Então, se o problema parou de funcionar durante a implementação da **funcionalidade B**, você poderá desfazer tudo relacionado a essa funcionalidade sem afetar o que foi implementado na **funcionalidade A**.

Esta é exatamente uma das principais vantagens que o versionamento de projetos proporciona: o registro de tudo o que foi feito em um projeto, que pode ser usado para restaurar a versão anterior caso algum problema aconteça. Esse “registro” é chamado, na linguagem técnica, de **commit**, e o que se faz com o comando **git add** é dizer quais arquivos serão adicionados ao registro – em outras palavras, quais arquivos serão rastreados (**tracked**).

Como você está na fase inicial de desenvolvimento do seu projeto (afinal, apenas criou o projeto no Apache NetBeans IDE), é conveniente adicionar todos os arquivos e todas as pastas ao registro do **commit**. Para isso, é possível informar individualmente todos os arquivos ou usar o caractere ponto-final, que se refere a “tudo que estiver dentro dessa pasta”.

Nesse exemplo, você executará o segundo comando **git add** . e depois verificará o estado do versionamento com o comando **git status**.



```
MINGW64:/c/Users/tiluc/Documents/NetBeansProjects/GitExemplo
tiluc@NZXT-PHANTOM MINGW64 ~/Documents/NetBeansProjects/GitExemplo (master)
$ git add .

tiluc@NZXT-PHANTOM MINGW64 ~/Documents/NetBeansProjects/GitExemplo (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   build.xml
    new file:   manifest.mf
    new file:   nbproject/build-impl.xml
    new file:   nbproject/genfiles.properties
    new file:   nbproject/private/private.properties
    new file:   nbproject/private/private.xml
    new file:   nbproject/project.properties
    new file:   nbproject/project.xml
    new file:   src/gitexemplo/GitExemplo.java

tiluc@NZXT-PHANTOM MINGW64 ~/Documents/NetBeansProjects/GitExemplo (master)
$ |
```

Figura 7 – Terminal do GitBash

Fonte: GitBash (2023)

Perceba que, agora, seus arquivos estão na cor verde, o que significa que eles estão prontos para ser rastreados.

Se você quisesse informar os arquivos e as pastas individualmente, o comando ficaria da seguinte maneira:

```
git add build.xml manifest.mfnbproject/* src/*
```

Perceba que, no caso das pastas, usa-se o asterisco no final. O motivo disso é simples: na interface de linha de comandos, o asterisco é conhecido como um curinga (também chamado de *wildcard*, no inglês) para completar o nome de algo. Pensando no seu comando, é preciso informar cada um dos arquivos que você quer, digitando o nome de cada um deles manualmente. Porém, graças ao asterisco, você está dizendo para o comando que usará **nome da pasta/qualquer coisa que estiver dentro dela**, o que faz economizar bastante tempo ao digitar os comandos.

Se esses conceitos são novos para você, pesquise na internet por “manipulação de arquivos e diretórios em linha de comando”, para conhecer e aprofundar-se mais no uso de interfaces de linha de comando, também conhecido como CLI (Command Line Interface).

Salvando rastreamento

No Git, um **commit** é um *snapshot* do seu repositório em um ponto específico no tempo, que permite guardar o estado do seu repositório. Dessa forma, ao longo do tempo, haverá uma “linha do tempo” do repositório, na qual estará registrado tudo o que aconteceu para que se chegasse ao resultado final.

Após selecionar quais arquivos irão para o registro de **commit**, você pode salvar o rastreamento com o comando **git commit**. Esse comando tem algumas possíveis sintaxes, mas a forma mais utilizada é **git commit -m “Uma mensagem aqui...”**, na qual se inclui uma breve mensagem para que se saiba a que esse **commit** se refere (o argumento **-m** é a sintaxe que diz ao comando que será adicionada uma mensagem). Procure sempre ser breve e claro nessa mensagem, para que você, futuramente, saiba o que foi implementado nessa versão do *software*. Futuramente, você trabalhará com outros programadores no mesmo projeto, e essa informação será muito importante para que vocês consigam diferenciar uma versão da outra.

Veja algumas dicas para escrever uma boa mensagem nos seus **commits**.

Tenha cuidado com a gramática, para não escrever palavras que possam causar confusão.

Tente realizar um **commit** para cada alteração feita no código. Dessa forma, caso seja necessário retonar algum **commit** anterior, poucos trechos serão afetados.

Seja transparente e breve em relação ao que foi feito. Caso implemente apenas parte de uma funcionalidade, procure especificar exatamente o que foi feito com poucas palavras.

Como você está realizando o seu primeiro **commit**, realize-o com a mensagem “Criação do projeto”, pois foi exatamente isso que você fez nesta etapa. Então, o seu comando ficará com a seguinte sintaxe:

git commit -m “Criação do projeto”

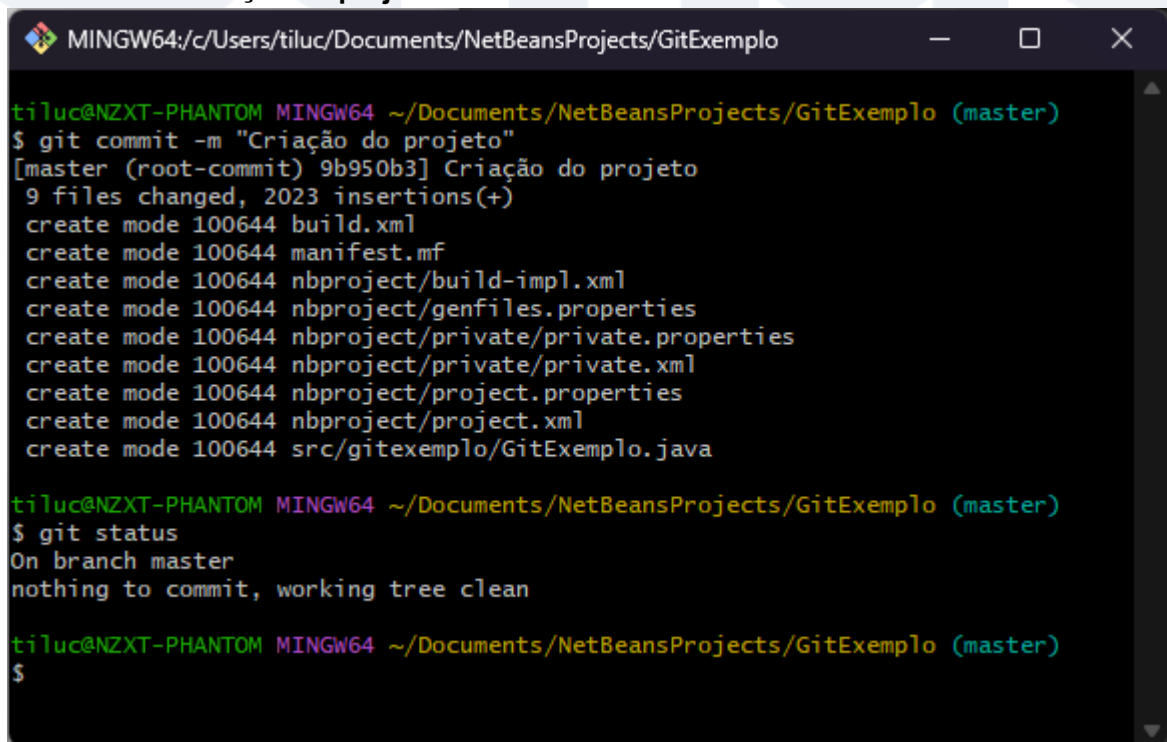
A screenshot of a terminal window titled "MINGW64:/c/Users/tiluc/Documents/NetBeansProjects/GitExemplo". The terminal shows the user 'tiluc@NZXT-PHANTOM' in a 'MINGW64' environment at the directory '~/Documents/NetBeansProjects/GitExemplo' on the 'master' branch. The user enters the command '\$ git commit -m "Criação do projeto"'. The output shows the commit was successful, with a hash of 9b950b3 and the message 'Criação do projeto'. It lists 9 files changed with 2023 insertions, including build.xml, manifest.mf, and various nbproject files. After the commit, the user enters '\$ git status', and the output shows 'On branch master' and 'nothing to commit, working tree clean'. The terminal prompt returns to '\$'.

Figura 8 – Comando **git commit** no terminal do GitBash

Fonte: GitBash (2023)

Caso o nome e o *e-mail* do usuário não tenham sido corretamente configurados, uma mensagem (“Author identity unknown”) aparecerá como resposta ao comando. Basta usar os comandos **git config --global user.email** e **git config --global user.name**, explicados anteriormente.

Após isso, conclui-se o rastreamento da versão do projeto. A partir daqui, sempre que você fizer alguma mudança no projeto, pode realizar um novo *snapshot* dele com os comandos **git add** e **git commit**.

O termo *snapshot* refere-se a uma captura instantânea de volume ou um retrato fiel de uma situação do ambiente. O termo vem de uma analogia à fotografia, já que é a captura de um estado em um determinado ponto no tempo.

Agora, volte para o seu código do Apache NetBeans IDE. Altere o código do seu arquivo principal e escreva um “Hello World”. Aproveitando, também limpe os comentários gerados na criação do projeto para deixar o seu código mais limpo. O código ficará da seguinte maneira:

```
package gitexemplo;

public class GitExemplo {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

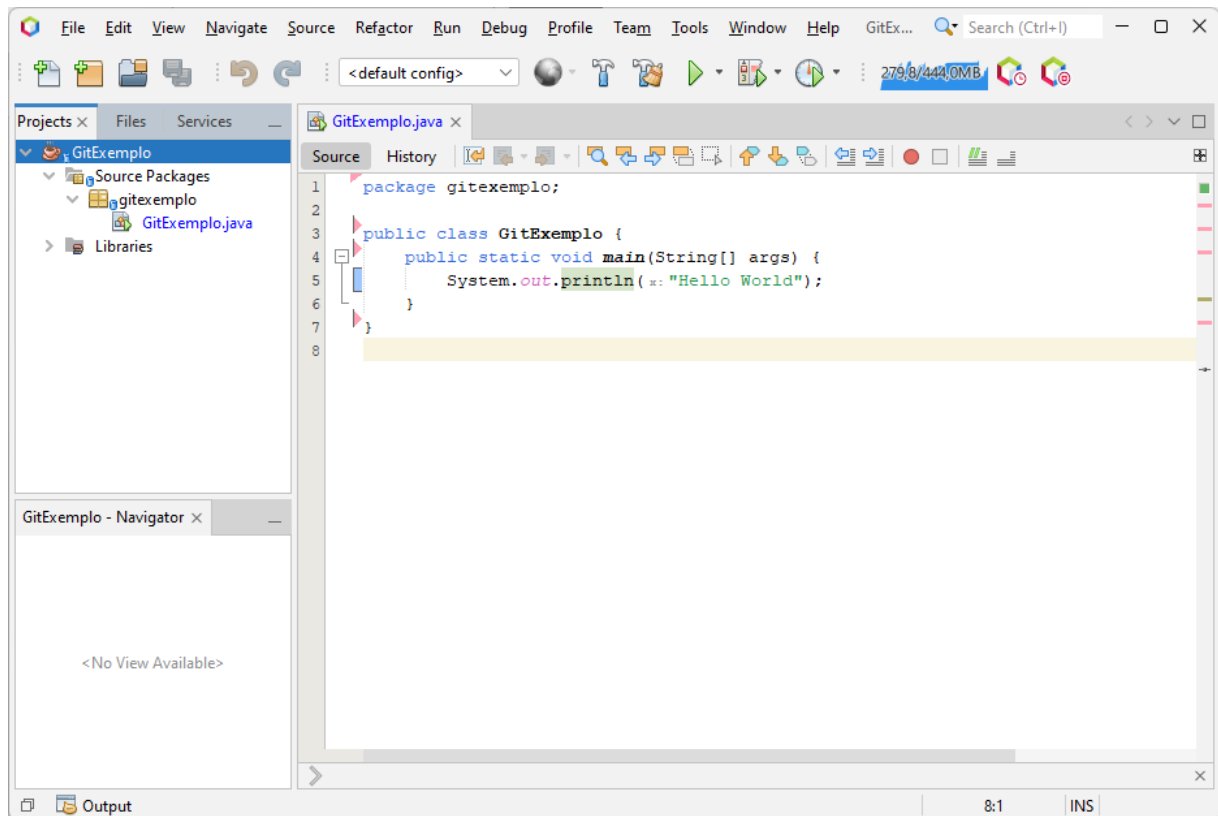


Figura 9 – Código-fonte de Hello World em Java

Fonte: Apache NetBeans IDE (2023)

No Apache NetBeans IDE, você pode ter percebido que, ao lado das linhas de código, apareceram algumas setas vermelhas. Se você clicar em alguma delas, verá que o editor de código está apontando quais trechos de código foram removidos em comparação ao último **commit** realizado. Esse recurso pode ser muito útil caso você precise consultar rapidamente as mudanças feitas no código direto pelo Apache NetBeans IDE.

Voltando ao terminal do GitBash, agora que você já concluiu as alterações do seu projeto, precisa digitar apenas dois comandos:

```
gitadd .
gitcommit -m "Hello World"
```

Pronto! Com isso, você conclui o seu segundo **commit**. Guarde bem essa informação, pois voltará nesse assunto em breve.

Criando repositório remoto

Repositórios remotos são versões do seu projeto que estão hospedadas na Internet ou em outra rede. Você pode ter vários deles e, também, ter várias pessoas trabalhando no mesmo repositório, o que torna possível a colaboração de vários programadores no desenvolvimento de um mesmo projeto de *software*.

Até aqui, você já criou um repositório local, fez o versionamento do seu projeto e tem todas essas informações guardadas no computador. Agora, você criará um repositório remoto em que guardará todos os registros de versão do projeto, incluindo o código-fonte. Para isso, você utilizará o GitHub, uma plataforma de repositórios remotos e versionamento bastante popular e acessível para todos, já que conta com várias ferramentas para gerenciar o versionamento de projetos de forma gratuita.

Antes de continuar, é importante que você já tenha uma conta criada na plataforma GitHub. Caso ainda não tenha, faça uma pausa na leitura deste conteúdo, crie a sua conta e depois volte para seguir com esta prática.

É bem simples criar uma conta no GitHub. Basta acessar o *site* oficial e clicar sobre o botão **Sign up** localizado no canto superior direito da página. Um formulário será aberto para você preencher com seus dados e cadastrar o seu usuário na plataforma.

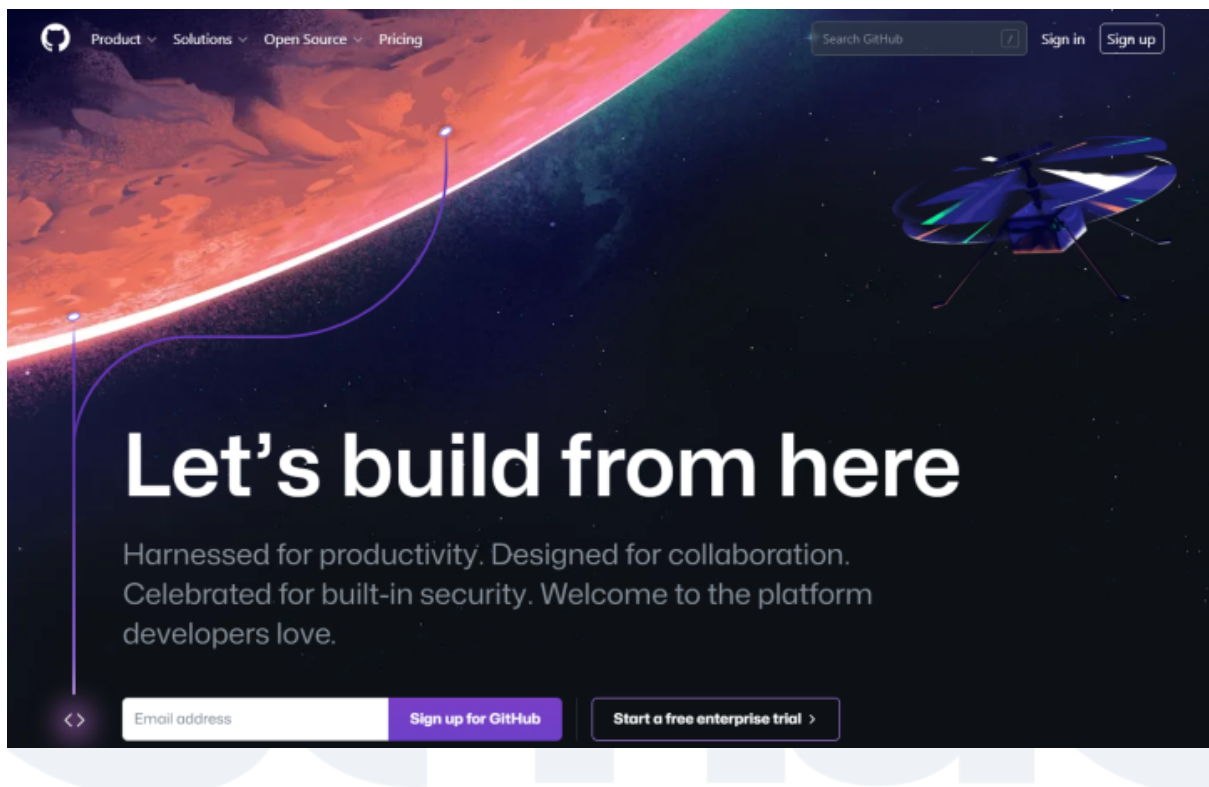


Figura 10 – Página inicial do *site* GitHub

Fonte: GitHub (2023)

Criando um repositório remoto no GitHub

Com seu usuário autenticado na plataforma do GitHub, clique no símbolo de mais (+) localizado na barra superior da página, mais precisamente ao lado direito, e clique na opção **New repository** para acessar à página de criação de repositórios remotos.

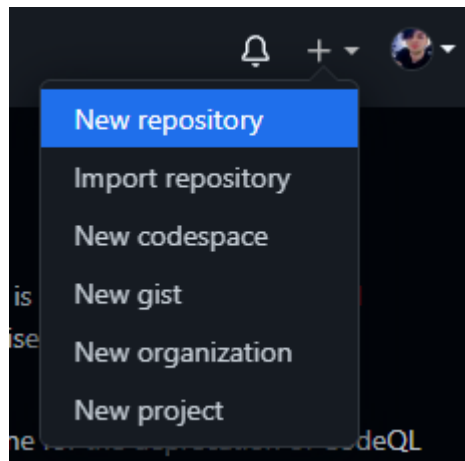


Figura 11 – Criação de novo repositório no GitHub

Fonte: GitHub (2023)

Você será direcionado a uma página chamada **Create a new repository**, na qual terá um breve formulário para preencher com as informações do seu repositório.

Create a new repository


A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Repository template

Start your repository with a template repository's contents.


No template ▾


Owner * **Repository name ***

 lucasfrag ▾ / ✓

Great repository names are short, lowercase, and contain only numbers, lowercase letters, hyphens, and underscores. Your new repository will be created as **Git-Exemplo**. v about **super-duper-succotash?**

Description (optional)

☒  **Public**
Anyone on the internet can see this repository. You choose who can commit.


☐  **Private**
You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

☒ **Add a README file**
This is where you can write a long description for your project. [Learn more.](#)

Add .gitignore
Choose which files not to track from a list of templates. [Learn more.](#)

Choose a license
A license tells others what they can and can't do with your code. [Learn more.](#)

 You are creating a public repository in your personal account.

Create repository

Figura 12 – Criação de novo repositório no GitHub

Fonte: GitHub (2023)

A seguir, entenda melhor o que cada uma dessas opções significa.

Repository name

Aqui, você informará o nome do repositório que será criado. É uma boa prática o nome do repositório ter o mesmo nome do projeto, mas isso não é uma regra. As regras que precisam ser lembradas são: o nome do repositório não pode conter espaços ou caracteres especiais (se houver, o GitHub inserirá um hífen no lugar do caractere); e o nome do repositório não pode ser igual ao de outro repositório que você já tenha criado.

Description

Neste campo, você pode trazer uma breve descrição do repositório para que você e outras pessoas saibam do que se trata esse projeto.

Public/Private

Você pode escolher se deseja que o repositório fique visível apenas para você (**private**) ou para todos que acessarem o seu perfil (**public**). Outros poderão visualizar e baixar o seu projeto caso a opção **public** seja selecionada. Para a criação de um portfólio, isso é ótimo! Porém, caso esteja lidando com um projeto mais sensível, a melhor opção é deixá-lo como **private**.

Initialize this repository with

Este é um trecho referente à inicialização do repositório remoto. É como se você estivesse criando uma pasta no servidor do GitHub e usando o **git init** para iniciar o versionamento. Logo, é possível selecionar a inicialização com um arquivo **README.md** e configurar um arquivo **.gitignore** (um arquivo de regras que diz ao repositório quais arquivos, pastas ou extensões serão ignorados no versionamento).

Choose a license

Os repositórios públicos no GitHub costumam ser usados para compartilhar *software* de código aberto. Para que seu repositório seja realmente de código aberto, você precisará licenciá-lo para que outras pessoas tenham liberdade para usar, alterar e distribuir o *software*. Nessa opção, você seleciona o licenciamento do seu repositório.

Para esse exemplo, o **Repository name** foi preenchido com o nome do projeto, “Git Exemplo”, e foi usado o espaço para demonstrar a correção que o GitHub faz quando se usa o espaço nesse campo. Em **Description**, o projeto foi descrito como “Um repositório para exemplificar o uso dos comandos Git e repositório remoto do GitHub”. A última configuração feita foi selecionar **Public** na visibilidade do repositório. Após isso, deve-se clicar em **Create repository** para finalizar a criação.

Assim que o repositório for criado, você será redirecionado para uma nova página. Perceba que o endereço dessa página tem a seguinte estrutura:

<https://github.com/Nome do usuário/Nome do repositório>

Esse é o endereço de acesso do seu repositório, onde “nome do usuário” será o nome do usuário que você definiu durante o processo de criação de conta no GitHub e “nome do repositório” será o nome que você definiu durante a criação do repositório remoto. Lembre-se de que, caso tenha usado espaço ou algum caráter especial, o GitHub substituirá esses caracteres por um hífen, já que estes não são permitidos na codificação de endereços eletrônicos.

Para ter mais informações sobre como funcionam as regras de codificação de endereços eletrônicos na *web*, pesquise por “Codificação de URL” ou “Codificação por cento”.

Nessa nova página, a plataforma do GitHub trará algumas sugestões de comandos para executar caso o seu repositório local ainda não exista ou caso ele já exista. Dentre as opções, siga a segunda abordagem **...or push an existing repository from the command line**, que se refere ao envio de um projeto já versionado para o repositório remoto.

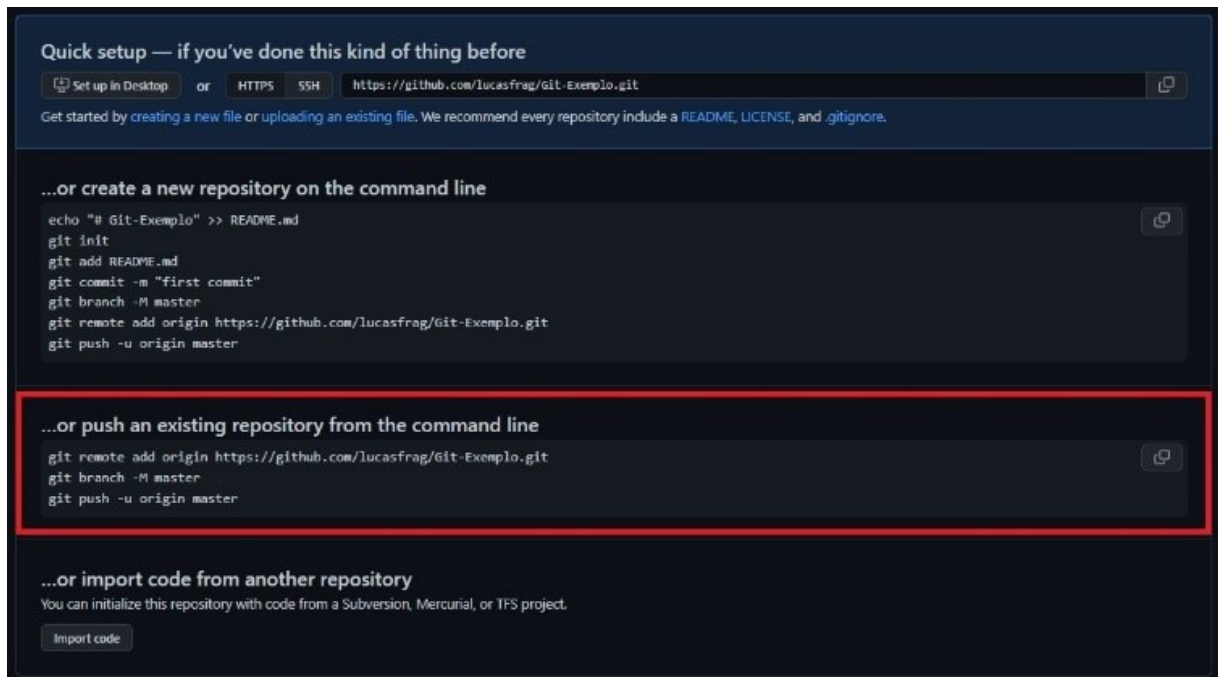


Figura 13 – Repositório remoto no GitHub

Fonte: GitHub (2023)

É possível copiar os comandos sugeridos pelo GitHub ou digitá-los manualmente. Por questões didáticas, a recomendação é que você digite cada comando para entender melhor cada etapa do processo. Futuramente, você pode aproveitar esse recurso para agilizar o processo no seu dia a dia como desenvolvedor. O primeiro comando que você usará será o **git remote**.

Adicionando repositório remoto

O comando **git remote** permite criar, visualizar e excluir conexões com outros repositórios. As conexões remotas são mais como favoritos do que *links* diretos para outros repositórios. Em vez de fornecer acesso em tempo real a outro repositório, eles servem de nomes convenientes que podem ser usados para referenciar uma URL (*uniform resource locator*) não tão conveniente.

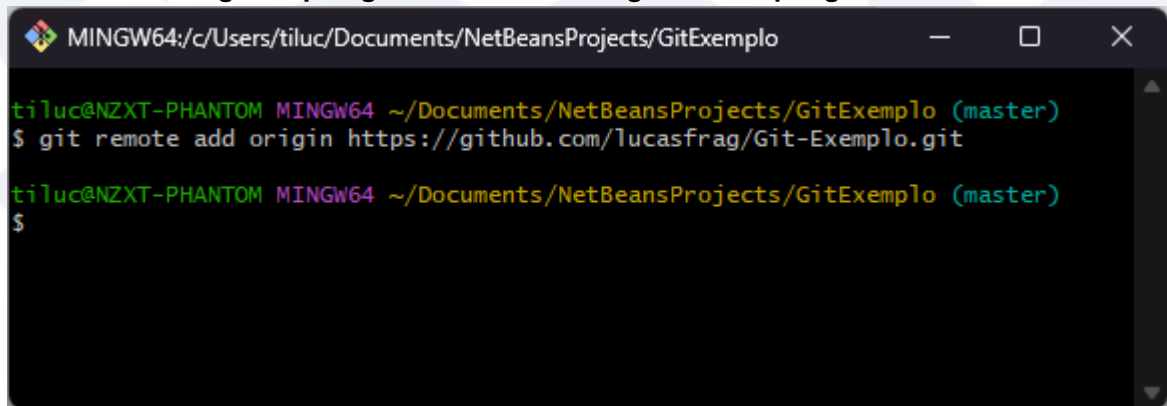
Na prática, é preciso dizer “para onde” serão enviados os arquivos do seu repositório. Então, seguindo o exemplo, seria necessário enviar para o endereço:

<https://github.com/lucasfrag/Git-Exemplo.git>

Porém, digitar esse endereço manualmente na linha de comando seria algo muito trabalhoso e tornaria a operação de envio para o repositório remoto mais complexa e cansativa. Portanto, com o comando **git remote add <nome> <endereço do repositório remoto>**, você vinculará um nome simples que servirá de “nome alternativo” para esse endereço.

Seguindo o exemplo, o comando ficará:

git remote add origin https://github.com/lucasfrag/Git-Exemplo.git

A screenshot of a terminal window titled "MINGW64:/c/Users/tiluc/Documents/NetBeansProjects/GitExemplo". The terminal shows the user "tiluc@NZXT-PHANTOM" in a "MINGW64" environment at the directory "~/Documents/NetBeansProjects/GitExemplo (master)". The command "\$ git remote add origin https://github.com/lucasfrag/Git-Exemplo.git" has been entered and executed, resulting in a new prompt "\$".

```
MINGW64:/c/Users/tiluc/Documents/NetBeansProjects/GitExemplo
tiluc@NZXT-PHANTOM MINGW64 ~/Documents/NetBeansProjects/GitExemplo (master)
$ git remote add origin https://github.com/lucasfrag/Git-Exemplo.git
tiluc@NZXT-PHANTOM MINGW64 ~/Documents/NetBeansProjects/GitExemplo (master)
$
```

Figura 14 – Terminal do GitBash

Fonte: GitHub (2023)

O comando que você deve executar no seu computador deve conter o nome do seu usuário do GitHub e o nome do seu repositório remoto.

Nesse comando, “origin” é o nome alternativo definido. Por boas práticas, esse é um nome padrão utilizado para se referir ao repositório remoto principal. Então, sempre que você usar o termo “origin” nos comandos do Git, a ferramenta entenderá que está se referindo a esse endereço eletrônico que foi vinculado.

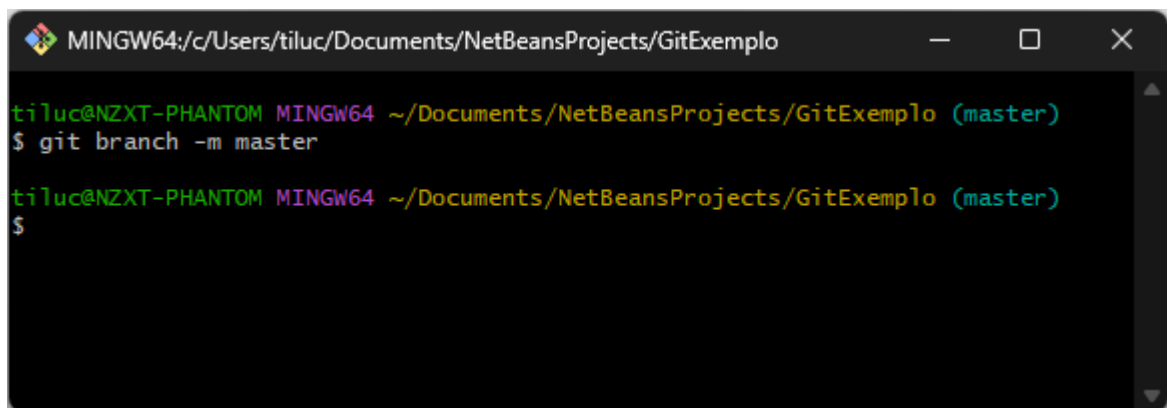
Selecionando a ramificação principal

Uma das funções mais importantes no controle de versões do Git são as ramificações, também conhecidas como *branches*.

Imagine que você tem um projeto 100% funcional e surge a necessidade de modificar uma das funcionalidades desse projeto. Se fizer essa mudança diretamente, corre o risco de quebrar o que já está funcionando e causar outros problemas posteriores que podem comprometer todo o *software*. Porém, quando se cria uma ramificação do código principal, é possível fazer alterações sem modificar o que está funcionando. Então, pode-se pensar nas ramificações como áreas isoladas onde o código é duplicado e é possível modificar e testar o projeto sem afetar outras versões já funcionais.

Futuramente, os conceitos e as práticas de ramificações serão mais aprofundados. Por ora, tudo o que você precisa é definir a ramificação principal com o comando **git branch** (para fazer essa definição, use o argumento **-m**). Então, execute o seguinte comando:

git branch -m master

A screenshot of a Git Bash terminal window. The title bar shows the path 'MINGW64:/c/Users/tiluc/Documents/NetBeansProjects/GitExemplo'. The terminal content shows a user prompt 'tiluc@NZXT-PHANTOM MINGW64 ~/Documents/NetBeansProjects/GitExemplo (master)' followed by the command '\$ git branch -m master'. The prompt repeats, indicating the command was executed successfully.

```
MINGW64:/c/Users/tiluc/Documents/NetBeansProjects/GitExemplo
tiluc@NZXT-PHANTOM MINGW64 ~/Documents/NetBeansProjects/GitExemplo (master)
$ git branch -m master
tiluc@NZXT-PHANTOM MINGW64 ~/Documents/NetBeansProjects/GitExemplo (master)
$
```

Figura 15 – Terminal do GitBash

Fonte: GitBash (2023)

Dependendo da data de criação da sua conta no GitHub, é possível que o seu usuário esteja configurado para usar a ramificação **main** como *branch* principal.

Para confirmar essa informação, você pode acessar, no *site* do GitHub, a opção **Settings > Repositores** e verificar o nome definido em **Repository default branch**.

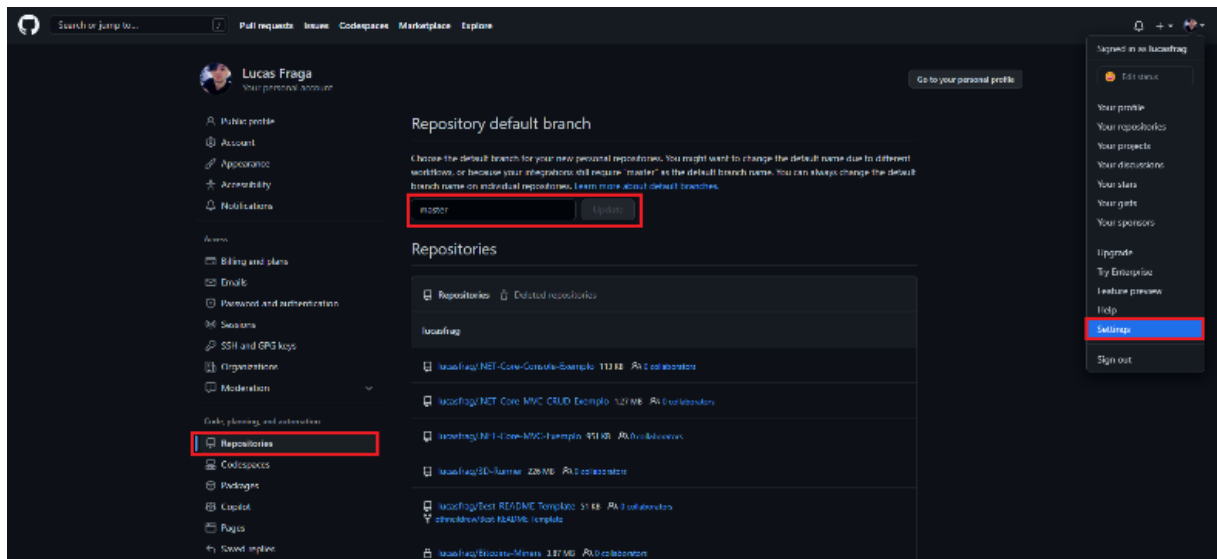


Figura 16 – Configurações de usuário no GitHub

Fonte: GitHub (2023)

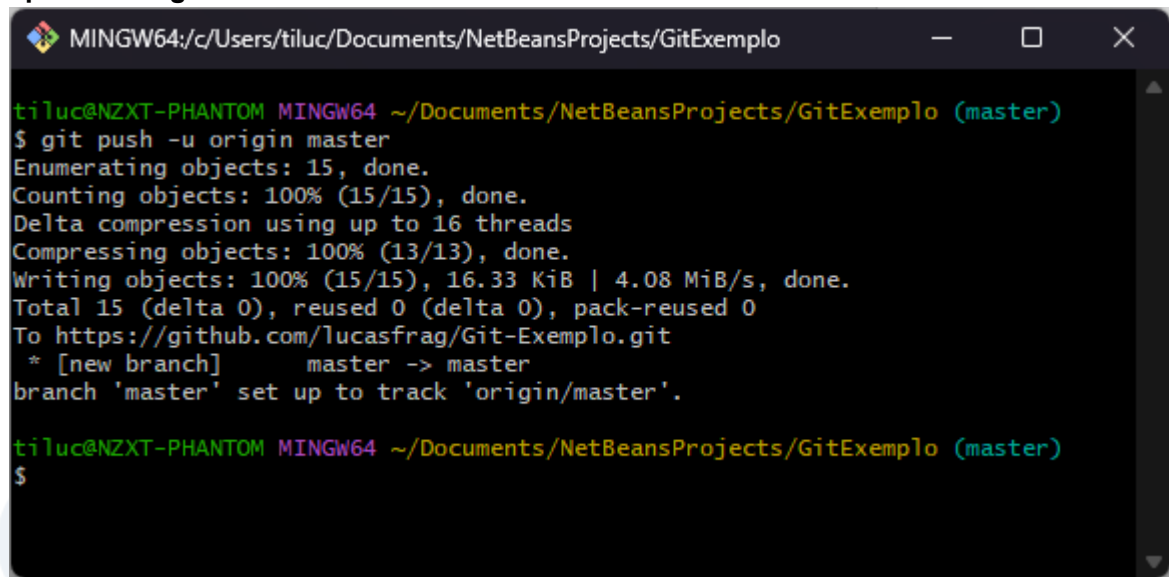
Se este for o caso, você pode executar o comando anteriormente apontado para essa ramificação. Ele ficará: **git branch -m main**.

Enviando para o repositório remoto

Agora, você tem tudo pronto para, de fato, enviar tudo que está no repositório local para o repositório remoto. Para isso, use o comando **git push**. O termo *push* vem do inglês e significa “empurrar”. Essa é uma analogia para o ato de mover algo de um local para outro, sendo, então, os arquivos do repositório local para o repositório remoto.

O processo de *pushing* fará o *upload* não apenas dos seus arquivos e pastas, mas também de todo o histórico de versionamento (no caso, os registros de **commit**). Para executar esse comando, é preciso dizer para qual local e *branch* “empurrará” os dados. Como você já executou os comandos **git remote** e **git branch** para definir o repositório remoto e a ramificação principal, basta apontar para esses locais respectivamente (para essa definição, usa-se o argumento **-u**). Logo, o comando terá a seguinte sintaxe:

git push -u origin máster



```
MINGW64:/c/Users/tiluc/Documents/NetBeansProjects/GitExemplo

tiluc@NZXT-PHANTOM MINGW64 ~/Documents/NetBeansProjects/GitExemplo (master)
$ git push -u origin master
Enumerating objects: 15, done.
Counting objects: 100% (15/15), done.
Delta compression using up to 16 threads
Compressing objects: 100% (13/13), done.
Writing objects: 100% (15/15), 16.33 KiB | 4.08 MiB/s, done.
Total 15 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/lucasfrag/Git-Exemplo.git
 * [new branch]      master -> master
branch 'master' set up to track 'origin/master'.
```

Figura 17 – Terminal do GitBash

Fonte: GitBash (2023)

Assim que o comando for processado, você pode voltar à página do repositório lá no GitHub e atualizá-la. Você verá que agora todos os arquivos do projeto estarão lá, incluindo os registros de **commit** (para acessá-los, basta clicar no *link* de contagem dos **commits** abaixo do botão verde **Code**).

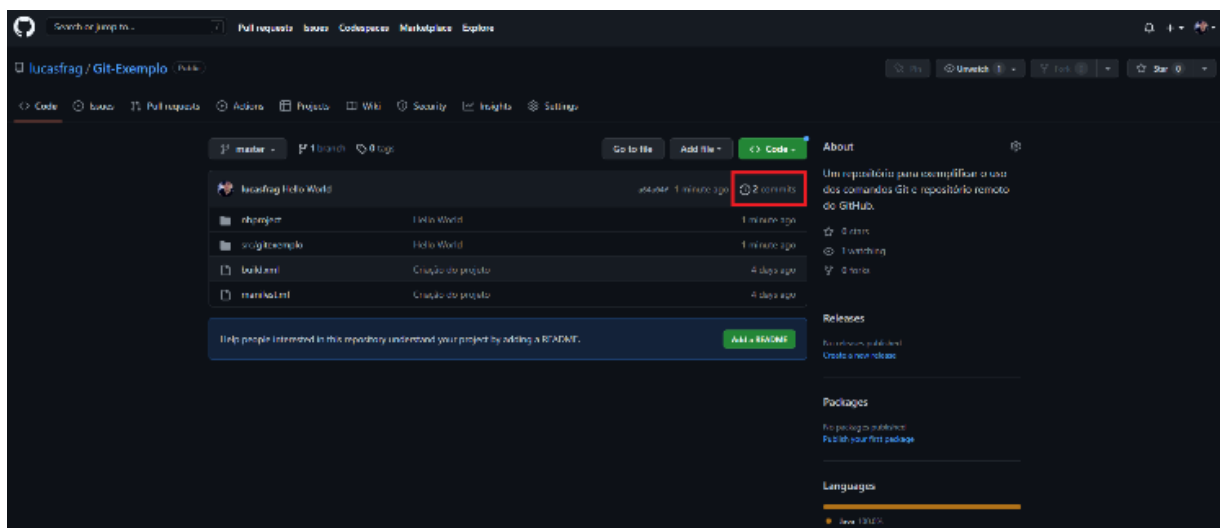
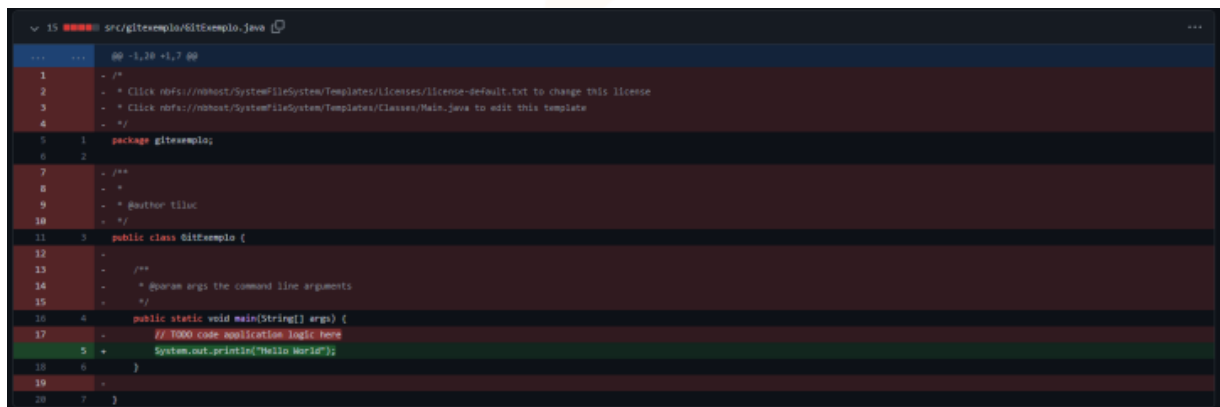


Figura 18 – Página do repositório remoto no GitHub

Fonte: GitHub (2023)

Você recorda que já foi comentado para não se esquecer do segundo **commit**? Se você acessar os registros de **commits** e selecionar o segundo, encontrará o histórico de absolutamente tudo que foi alterado no projeto desde o primeiro **commit**.

A seguir, em destaque vermelho, veja o que foi removido e, em destaque verde, o que foi alterado.

A screenshot of a code editor window showing a Java file named 'src/gitexemplo/GitExemplo.java'. The code is displayed with line numbers from 1 to 20. The editor uses a dark theme. Red highlights indicate lines that were removed in the current commit, and green highlights indicate lines that were added. The code includes a package declaration, a class declaration, and a main method. The main method contains a comment and a line of code that prints 'Hello World!'.

```
1  /**  
2   * Click ref: //localhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license  
3   * Click ref: //localhost/SystemFileSystem/Templates/Classes/Main.java to edit this template  
4   */  
5  package gitexemplo;  
6  
7  /**  
8   *  
9   * @author titoc  
10  */  
11  public class GitExemplo {  
12  
13      /**  
14       * @param args the command line arguments  
15       */  
16      public static void main(String[] args) {  
17          // TODO code application logic here  
18          System.out.println("Hello World!");  
19      }  
20  }
```

Figura 19 – Detalhes sobre o **commit** no GitHub

Fonte: GitHub (2023)

E é por meio desse histórico sobre o que foi alterado em cada arquivo que o Git permite desfazer alguma mudança que tenha comprometido o funcionamento do *software*. Além disso, vale destacar que o GitHub não informa o horário em que os **commits** chegaram no repositório remoto, mas, sim, o horário em que eles foram feitos localmente. Portanto, caso você fique sem conexão com a Internet, pode realizar os **commits** no seu computador e enviar para o GitHub mais tarde sem medo de causar a impressão que você fez as coisas de última hora.

Sincronizando repositório local com o repositório remoto

Quando se trabalha com o versionamento de projetos em equipes, é comum que o repositório remoto tenha informações que não estão sincronizadas com o repositório local. Isso pode ocorrer por causa de um **git push** realizado por outro programador que realizou alguma mudança no código-fonte. Nesses casos, é preciso trazer essas mudanças do repositório remoto para o repositório local.

Seguindo a analogia de “empurrar” os dados, quando se quer trazer dados remotos para o ambiente local, você está “puxando” essas informações. Daí o termo *pull* (em português, “puxar”) no comando **git pull**, que se refere ao processo de baixar o registro de versionamento ausente no repositório local.

Para ver, na prática, como esse comando funciona, crie um arquivo diretamente no repositório remoto por meio da página do GitHub. O processo aqui será bem simples: na página do repositório, clique no botão **Add a readme** para adicionar um arquivo **README.md**.

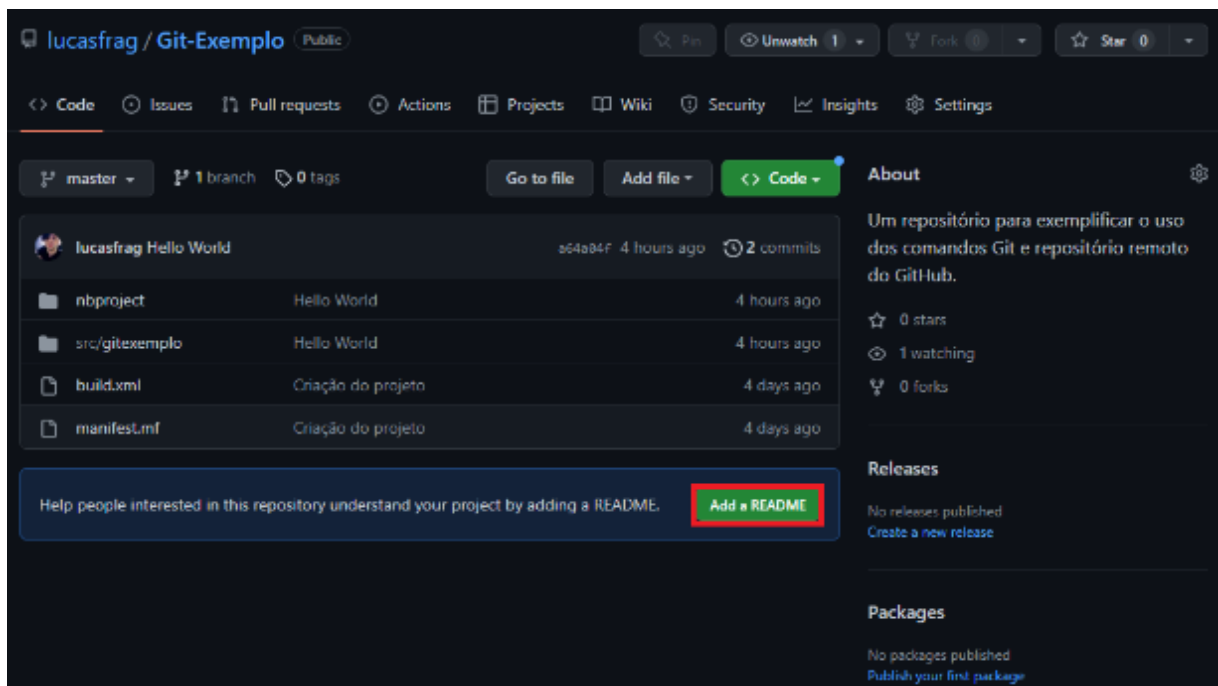


Figura 20 – Página do repositório remoto no GitHub

Fonte: GitHub (2023)

Arquivos **README.md** são a apresentação do repositório no GitHub. Ele é escrito em **Markdown**, uma linguagem de marcação utilizada para converter o texto em um HTML (sigla para *hypertext markup language*, que significa linguagem de marcação de hipertexto) válido. Você pode usar a linguagem de Markdown ou o próprio HTML para escrever esses arquivos. Como estes funcionam como uma “carta de apresentação” sobre o projeto, é importante que esses arquivos contenham informações como:

Descrição do projeto

Funcionalidades

Instalação (caso necessário) e como utilizá-lo

Tecnologias utilizadas

Autores do projeto

Ao clicar no botão, você será redirecionado a uma página com um modelo de **README.md** pré-construído baseado no nome e na descrição do repositório. Por enquanto, aproveite a estrutura montada e crie o arquivo do jeito que está para não perder o foco dessa prática. Para concluir a criação, desça a página e clique no botão **Commit new file**. Você perceberá que o próprio GitHub já preenche automaticamente a mensagem do novo **commit** como “Create README.md”. Caso você queira, você pode alterá-la.

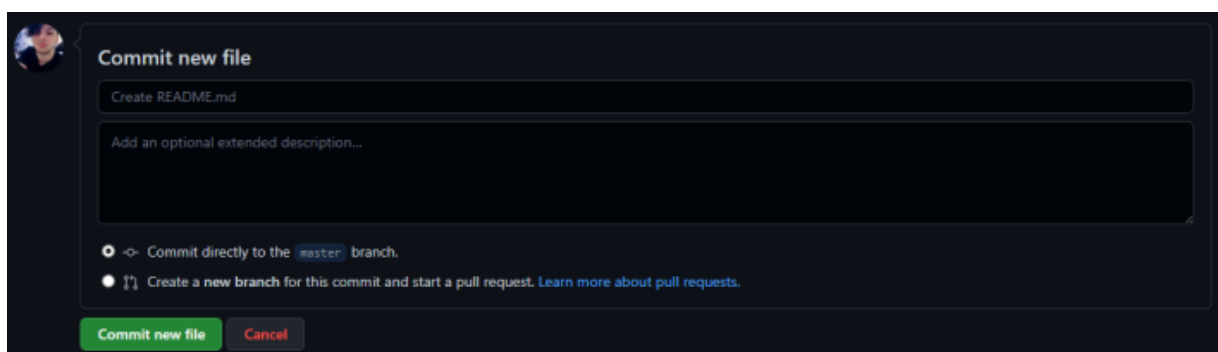


Figura 21 – **Commit** de novo arquivo no GitHub

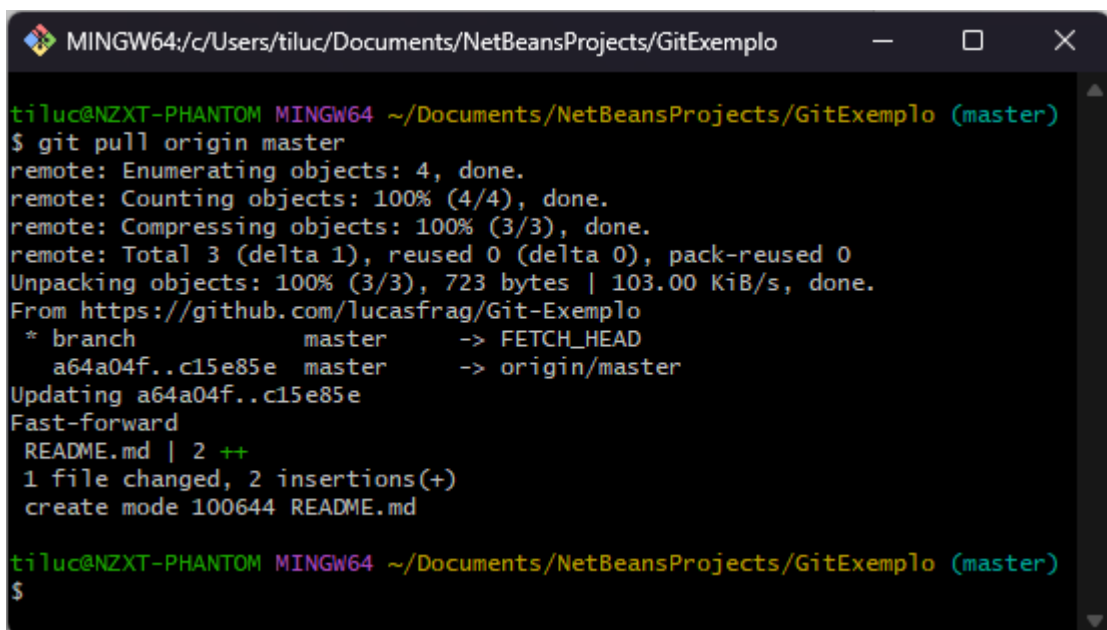
Fonte: GitHub (2023)

Agora o repositório remoto tem um **commit** a mais em relação ao repositório remoto. Enquanto essas informações não forem sincronizadas, você não será capaz de fazer envios do repositório local para o repositório remoto. Então, é uma boa prática sempre começar rodando o comando **git pull** antes de fazer qualquer alteração no projeto, quando estiver trabalhando em equipe, para receber todas as atualizações.

Agora, volte ao terminal do GitBash e execute o seguinte comando:

git pull origin master

Perceba que, assim como no comando **git push**, é preciso informar de qual repositório e ramificação você baixará as mudanças.



```
MINGW64:/c/Users/tiluc/Documents/NetBeansProjects/GitExemplo
tiluc@NZXT-PHANTOM MINGW64 ~/Documents/NetBeansProjects/GitExemplo (master)
$ git pull origin master
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), 723 bytes | 103.00 KiB/s, done.
From https://github.com/lucasfrag/Git-Exemplo
* branch      master      -> FETCH_HEAD
   a64a04f..c15e85e  master  -> origin/master
Updating a64a04f..c15e85e
Fast-forward
 README.md | 2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 README.md

tiluc@NZXT-PHANTOM MINGW64 ~/Documents/NetBeansProjects/GitExemplo (master)
$
```

Figura 22 – Terminal do GitBash

Fonte: GitBash (2023)

Após isso, o arquivo **README.md** que você acabou de criar aparecerá no diretório de arquivos do projeto.

Desfazendo alterações

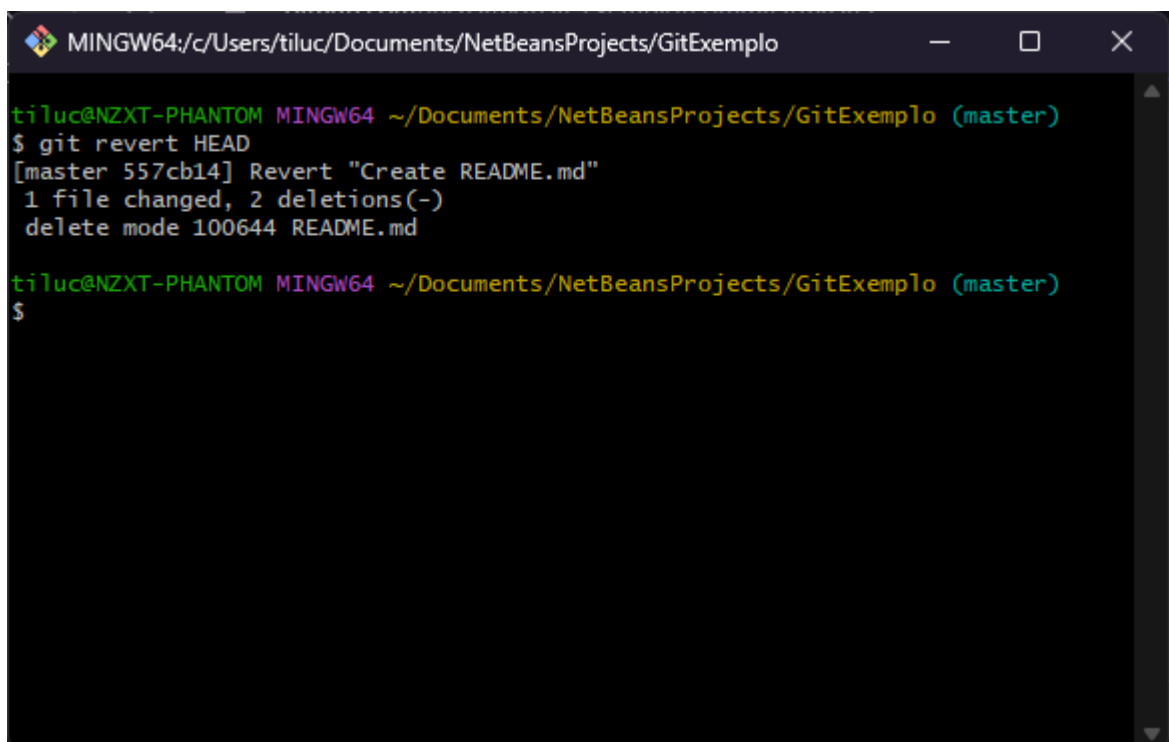
Existem várias formas diferentes para desfazer as alterações no projeto com o versionamento. Por questões didáticas, aborde a estratégia mais simples, que é reverter um **commit** anterior.

Para isso, tudo o que você precisa fazer é digitar o seguinte comando:

git revert HEAD

O *HEAD* é um ponteiro na sua ramificação. Por padrão, ele sempre apontará para o último **commit**. Nesse caso, **commit Create README.md** está sendo revertido.

Quando você executa o comando, será aberto um arquivo no editor de texto que você selecionou como padrão durante o processo de instalação do GitBash. Você não precisa fazer nenhuma mudança aqui. Apenas feche o arquivo, assim o terminal do GitBash retornará a seguinte mensagem:



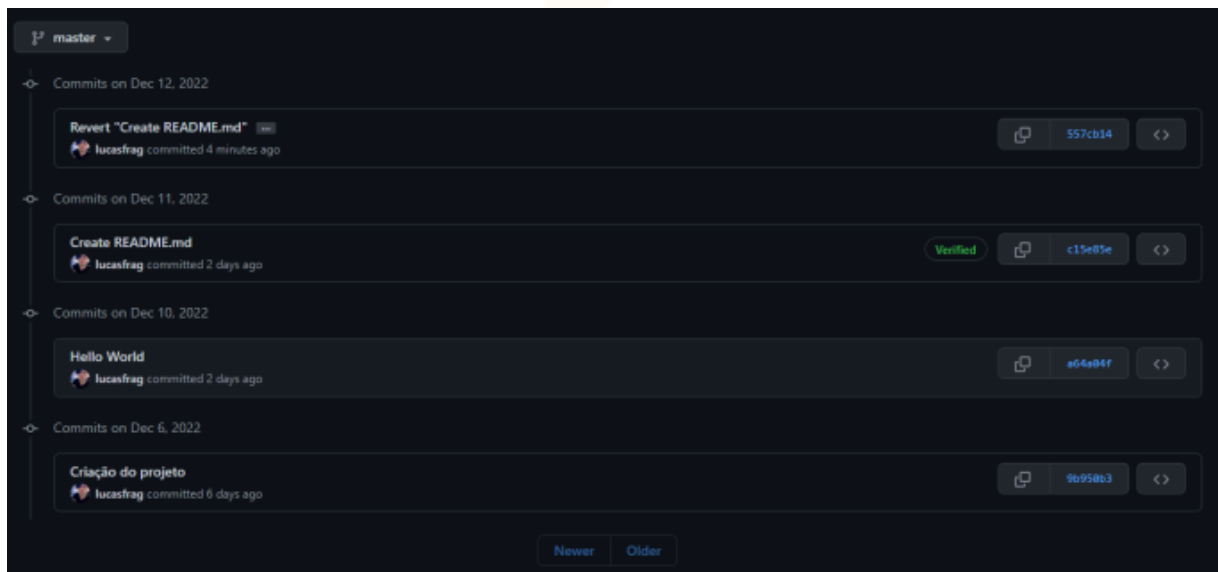
```
MINGW64; c:/Users/tiluc/Documents/NetBeansProjects/GitExemplo
tiluc@NZXT-PHANTOM MINGW64 ~/Documents/NetBeansProjects/GitExemplo (master)
$ git revert HEAD
[master 557cb14] Revert "Create README.md"
1 file changed, 2 deletions(-)
delete mode 100644 README.md

tiluc@NZXT-PHANTOM MINGW64 ~/Documents/NetBeansProjects/GitExemplo (master)
$
```

Figura 23 – Terminal do GitBash

Fonte: GitBash (2023)

Agora que tudo o que foi feito no repositório local foi revertido, é preciso sincronizar essas informações com o repositório remoto. Digite o comando **git push -u origin master** e você verá no histórico de **commits** do repositório remoto um novo **commit** referente à “reversão” que acabou de fazer.

Figura 24 – Histórico de **commits** no GitHub

Fonte: GitHub (2023)

Caso você quisesse reverter um **commit** mais antigo, por exemplo, o penúltimo, usaria o comando:

git revert HEAD~1

Se você quisesse o antepenúltimo **commit**, o comando ficaria:

git revert HEAD~2

Perceba que aqui há um cálculo sendo feito, em que o *HEAD* representa o último **commit** feito, o caractere *~* representa uma subtração e os números **1** e **2** apontam a contagem de **commits** que se quer voltar em relação ao último **commit** feito. Por

isso essa visualização do histórico de **commits** no GitHub é tão importante, pois, por meio dela, é possível identificar facilmente qual registro de **commit** você deseja reverter.

Clonando o repositório remoto

Imagine que você tenha feito todo o processo de versionamento no seu computador e agora precisar dar continuidade ao seu trabalho em outro computador. Se você sincronizou o seu repositório local com o repositório remoto, então tudo o que você precisa é dar continuidade ao seu trabalho que está lá no GitHub. Para isso, o primeiro passo é baixar o repositório remoto no outro computador. Para fazer isso, você pode usar a seguinte sequência de passos:

1. Criar uma pasta no computador.
2. Iniciar o versionamento com **git init**.
3. Apontar para o repositório remoto com o comando **git remote add origin <url>**.
4. “Puxar” o repositório completo com o commando **git pull origin master**.

Entretanto, nem sempre você desejará baixar o seu projeto para dar sequência à programação de um projeto. Às vezes, você pode querer apenas baixar o seu projeto em outro computador ou baixar o projeto de outro desenvolvedor disponível para uso livre do GitHub. Nesses casos, é possível fazer o *download* diretamente da página oficial do repositório remoto ou utilizar o comando **git clone**, que, literalmente, clonará o repositório.

GitHub Desktop

O GitHub Desktop é um aplicativo de código aberto que tem uma interface visual criada para simplificar e tornar o processo de versionamento mais produtivo. Apesar de a ferramenta tornar você independente da interface de linha de comando, pois não precisa mais digitar comandos para realizar o versionamento, ainda é necessário ter conhecimento sobre cada passo no ciclo de versionamento.

O GitHub Desktop, como o próprio nome sugere, só permite interagir com repositórios remotos do GitHub. Sendo assim, caso você se encontre trabalhando com um repositório remoto de outro local (como GitLab, Bitbucket etc.), a ferramenta não conseguirá realizar as tarefas de versionamento. Como alternativa, você pode utilizar a própria interface de linha de comando ou buscar um *software* que tenha compatibilidade com o repositório remoto em questão.

Configurando o GitHub Desktop

Na primeira vez que você abre o GitHub Desktop, é recebido com uma tela de boas-vindas e algumas opções para continuar. Para seguir, é recomendado já vincular a sua conta do GitHub ao aplicativo. Portanto, nessa primeira tela, selecione a opção **Sign in to GitHub.com** e conclua o processo de autenticação na janela que será aberta no seu navegador de Internet.

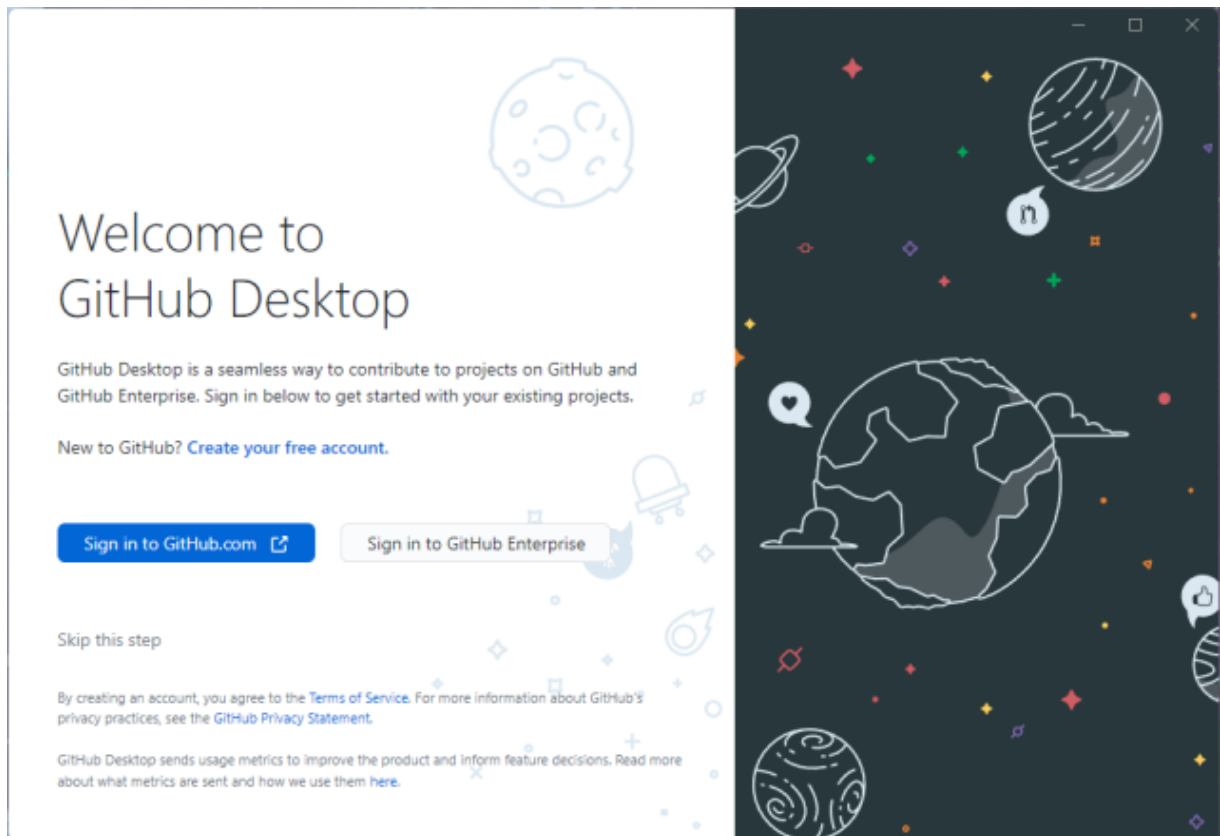


Figura 25 – Tela inicial do GitHub Desktop

Fonte: GitHub Desktop (2023)

Após isso, abrirá a tela de configuração do Git. Você se lembra quando configurou o nome e o *e-mail* do usuário autenticado no GitBash? Para isso, você utilizou o comando **git config**. Aqui, será o mesmo processo. Porém, por meio dos dados do GitHub, o aplicativo do GitHub Desktop já preencherá automaticamente o nome e o *e-mail* com base nos dados que você informou ao criar a sua conta na plataforma. Caso deseje alterar alguma informação, você pode selecionar a opção **Configure manually**. Caso esteja tudo certo, mantenha a opção **Use my GitHub account name and e-mail address** selecionada e clique no botão **Finish**.

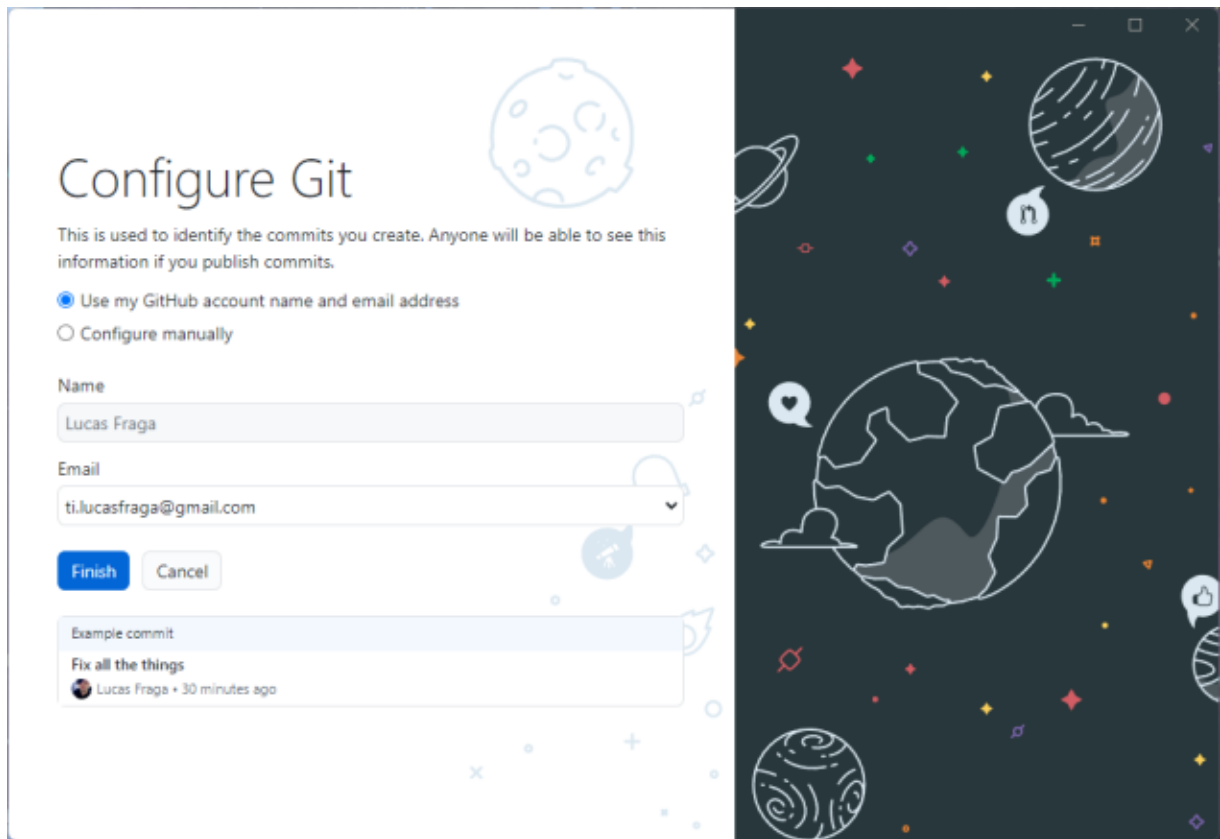


Figura 26 – Tela de configuração do GitHub Desktop

Fonte: GitHub Desktop (2023)

Iniciando um repositório local

Após concluir as configurações, você será levado à tela principal do GitHub Desktop.

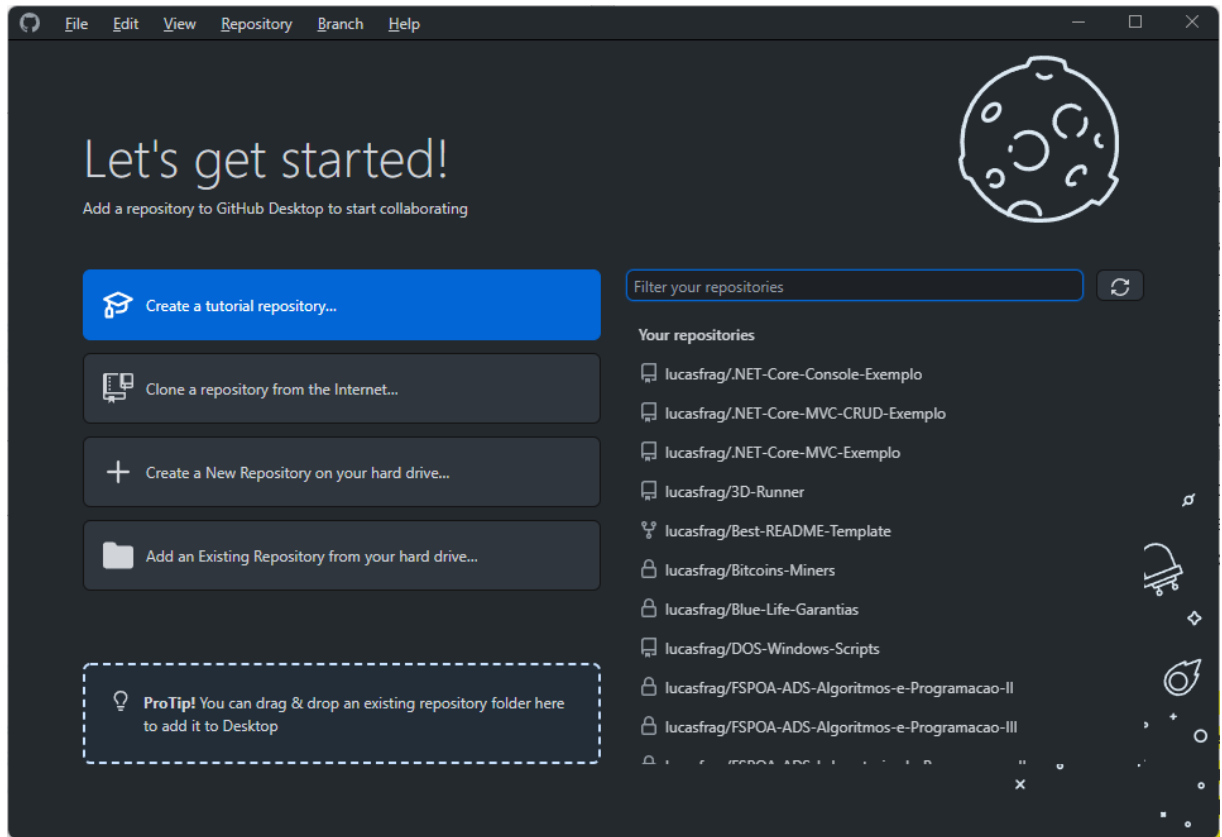


Figura 27 – Tela principal do GitHub Desktop

Fonte: GitHub Desktop (2023)

Nessa nova tela, haverá uma listagem dos repositórios que existem no GitHub e quatro opções para escolher:

Create a tutorial repository

Clone a repository from the Internet

Create a New Repository on your hard drive

Add an existing repository from your hard drive

Para realizar uma prática paralela à que você realizou com o GitBash, comece tudo do zero. Então, selecione a opção **Create a New Repository on your hard drive**.

Na nova janela que será aberta, haverá um formulário similar ao de criação de repositório remoto do GitHub. Aqui, preencha o nome do repositório como “GitHub Desktop Exemplo”, a descrição como “Um repositório para exemplificar o uso do GitHub Desktop”, selecione uma pasta onde fará o versionamento do projeto (para isso, crie uma nova pasta onde você gostaria de realizar essa prática e lembre-se bem dela, pois voltará nesse ponto depois) e deixe desmarcada a opção de iniciar o repositório com um arquivo **README**.

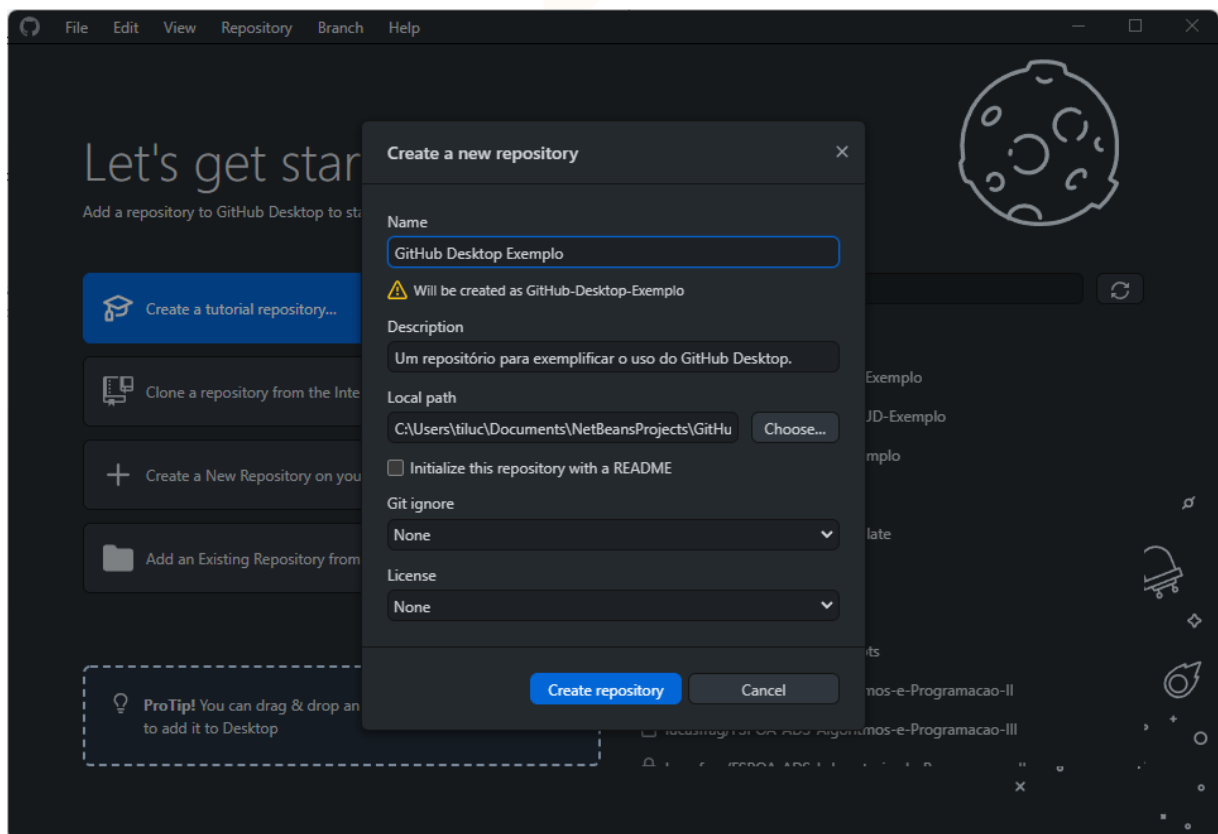


Figura 28 – Tela de criação de repositório do GitHub Desktop

Fonte: GitHub Desktop (2023)

Após preencher tudo, clique no botão **Create repository**. Agora, você terá uma pasta com o versionamento funcionando (esse passo é equivalente à execução do comando **git init**). Uma nova tela será apresentada informando que ainda não há nenhuma mudança no repositório local.

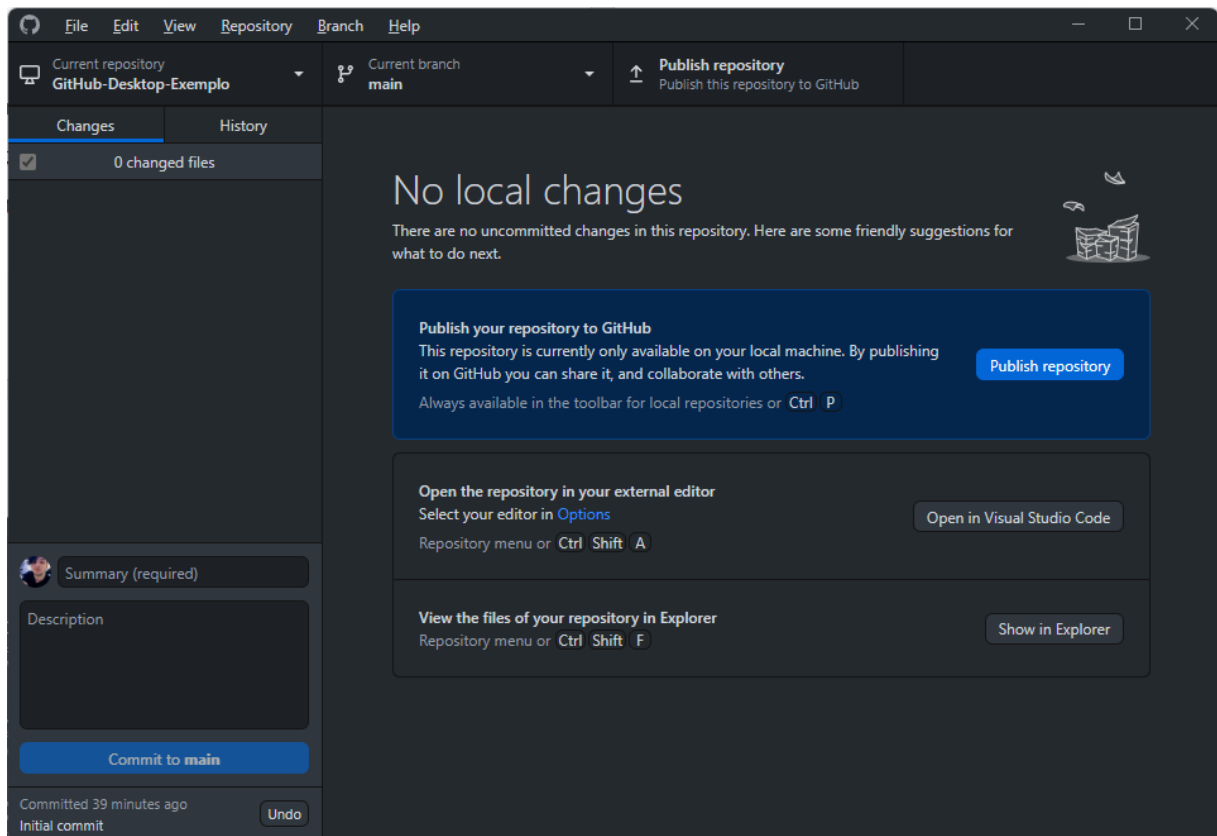


Figura 29 – Tela do repositório no GitHub Desktop

Fonte: GitHub Desktop (2023)

Criando projeto no repositório local

Agora, você criará um novo projeto no Apache NetBeans IDE dentro da pasta que está sendo versionada. Volte ao Apache NetBeans IDE e crie um novo projeto chamado “GitHub Desktop Exemplo” (se desejar, você pode usar outro nome; o importante é ter um novo projeto Java). Antes de concluir a criação do projeto, certifique-se de alterar o local de criação para o diretório que acabou de criar com o GitHub Desktop.

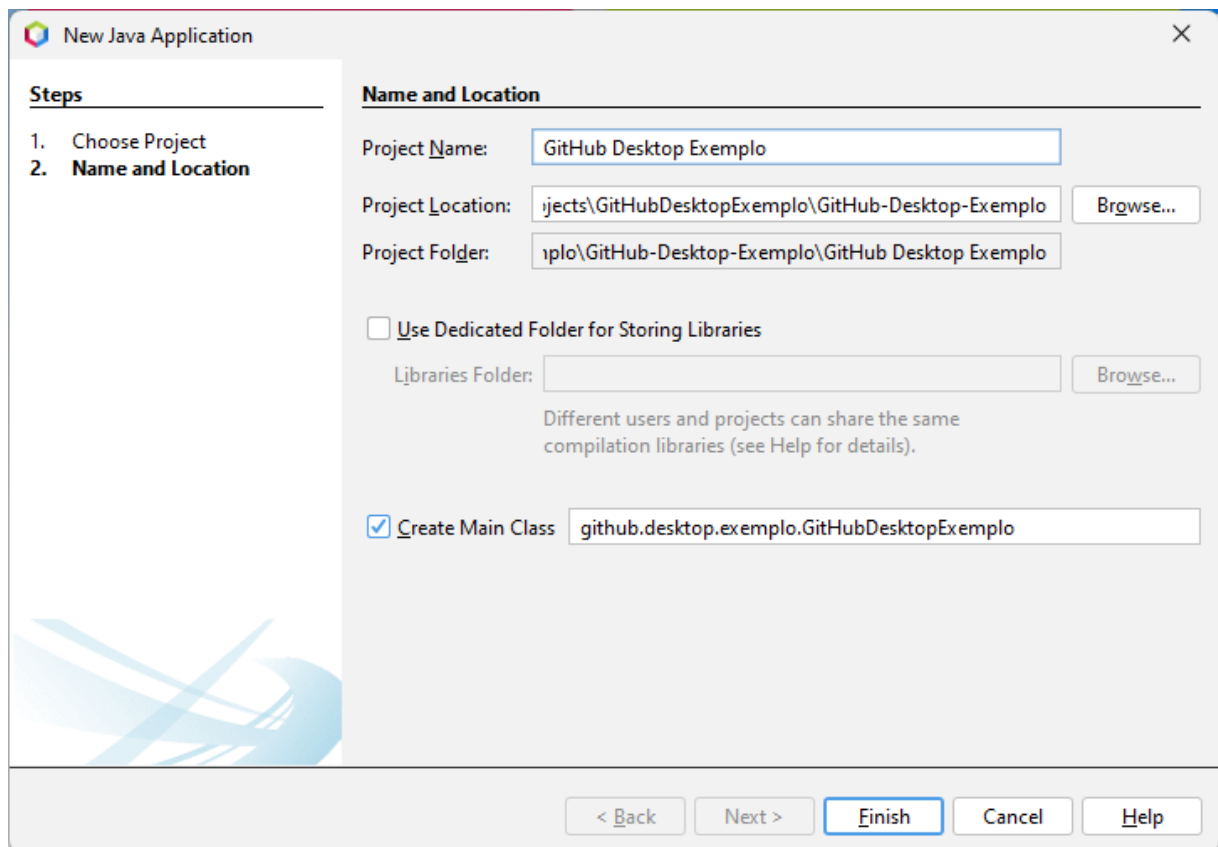


Figura 30 – Criação do projeto "GitHub Desktop Exemplo" no NetBeans

Fonte: Apache NetBeans IDE (2023)

Após o projeto ser criado com sucesso, volte ao GitHub Desktop e perceba algumas mudanças em relação à tela que você tinha anteriormente. Agora, os arquivos criados dentro da pasta versionada estão sendo listados como arquivos novos (pode-se associar isso ao comando **git status**).

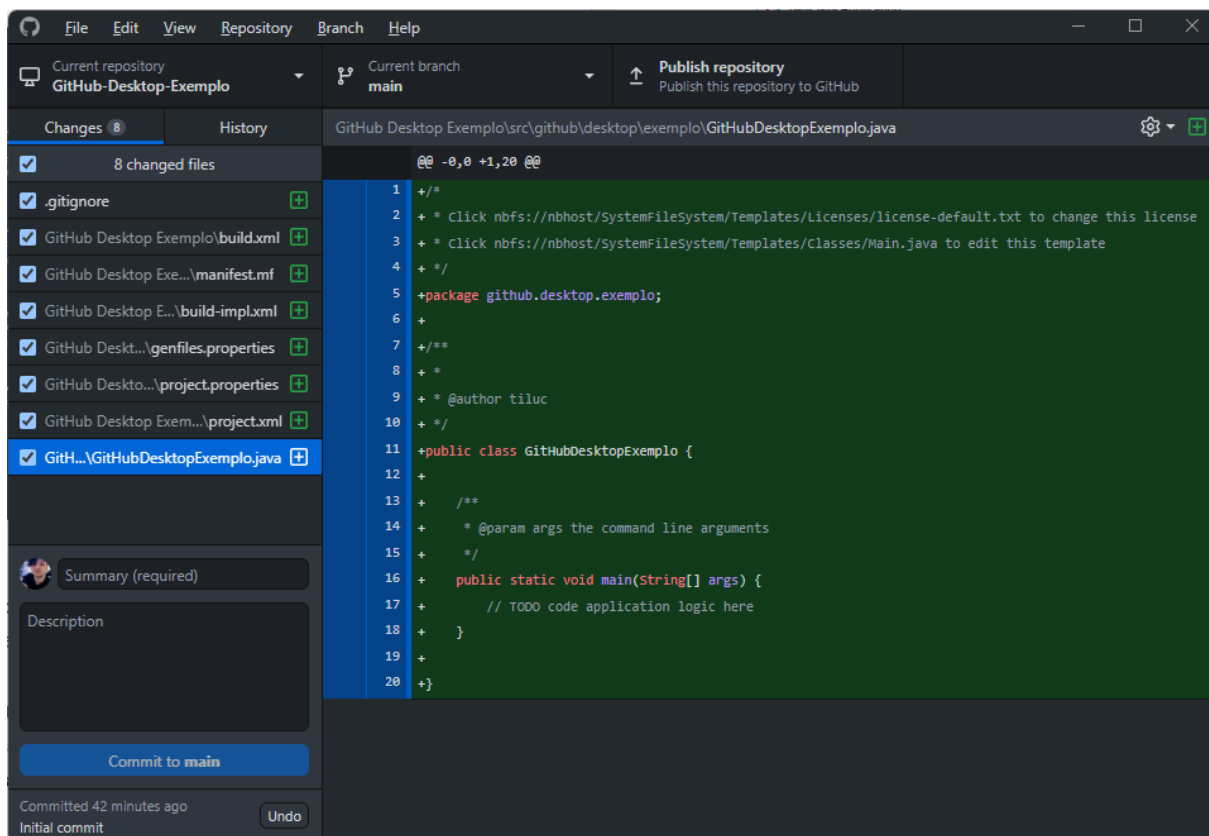


Figura 31 – Tela de histórico no GitHub Desktop

Fonte: GitHub Desktop (2023)

Registrando commit

Logo abaixo da listagem dos arquivos, há uma área para preencher os dados de registro do **commit**. Para este exemplo, o **commit** será chamado de “Criação do projeto”. Após preencher o campo, clique no botão **Commit to main** para concluir o **commit** (essa etapa é equivalente aos comandos **git add .** e **git commit**).

Após isso, a tela apresentará a mensagem de “No local changes” mais uma vez.

Enviando para o repositório remoto

Como você já tem o projeto versionado localmente, é a hora perfeita para enviar todas as informações para o repositório remoto. Para isso, clique no botão **Publish repository**.

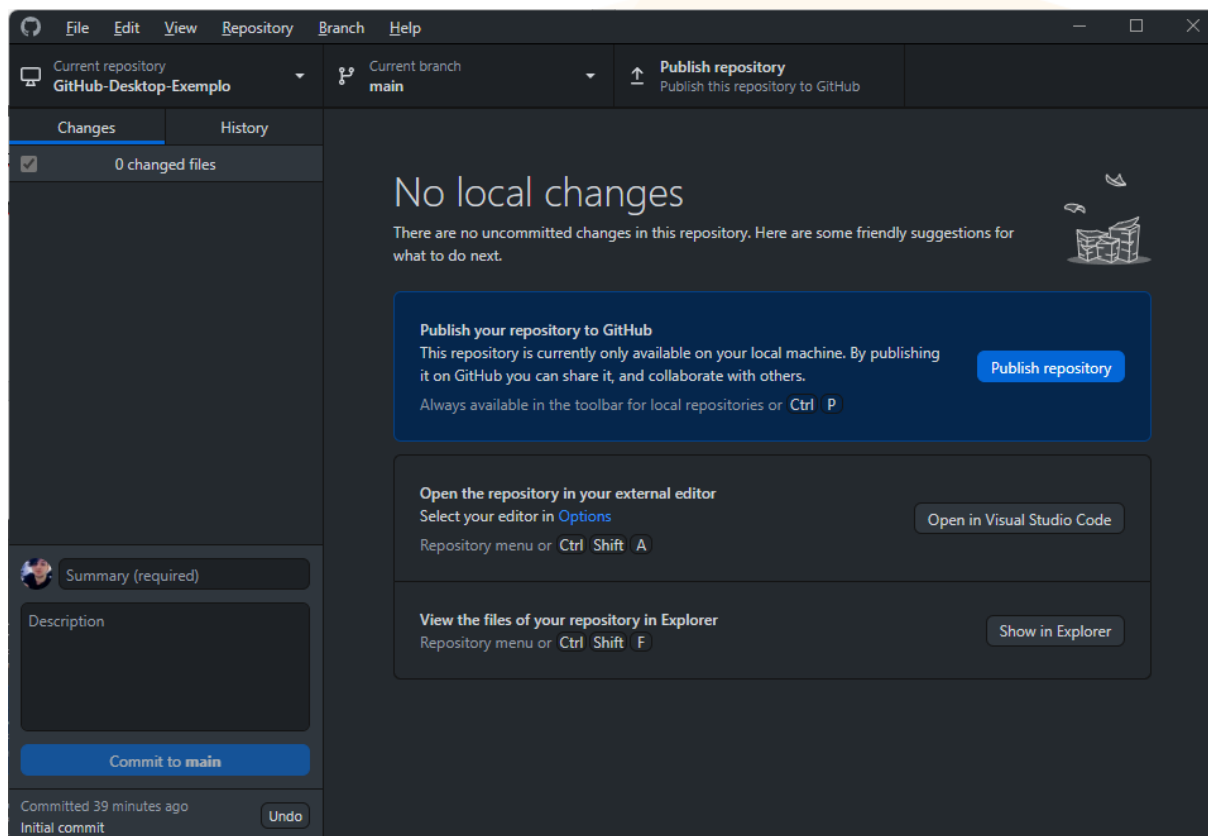


Figura 32 – Tela do repositório no GitHub Desktop

Fonte: GitHub (2023)

Essa ação criará automaticamente um repositório remoto no GitHub, usando todas as informações passadas na criação do repositório local. Porém, antes, há uma tela de confirmação para o GitHub Desktop saber se ele deve usar essas informações ou não. Caso queira alterar alguma informação, você pode fazer isso agora. Antes de continuar, apenas se certifique de que a opção **Keep this code private** está desmarcada, para criar o repositório remoto com visibilidade pública. Caso deseje mantê-lo privado, você pode deixar a opção marcada.

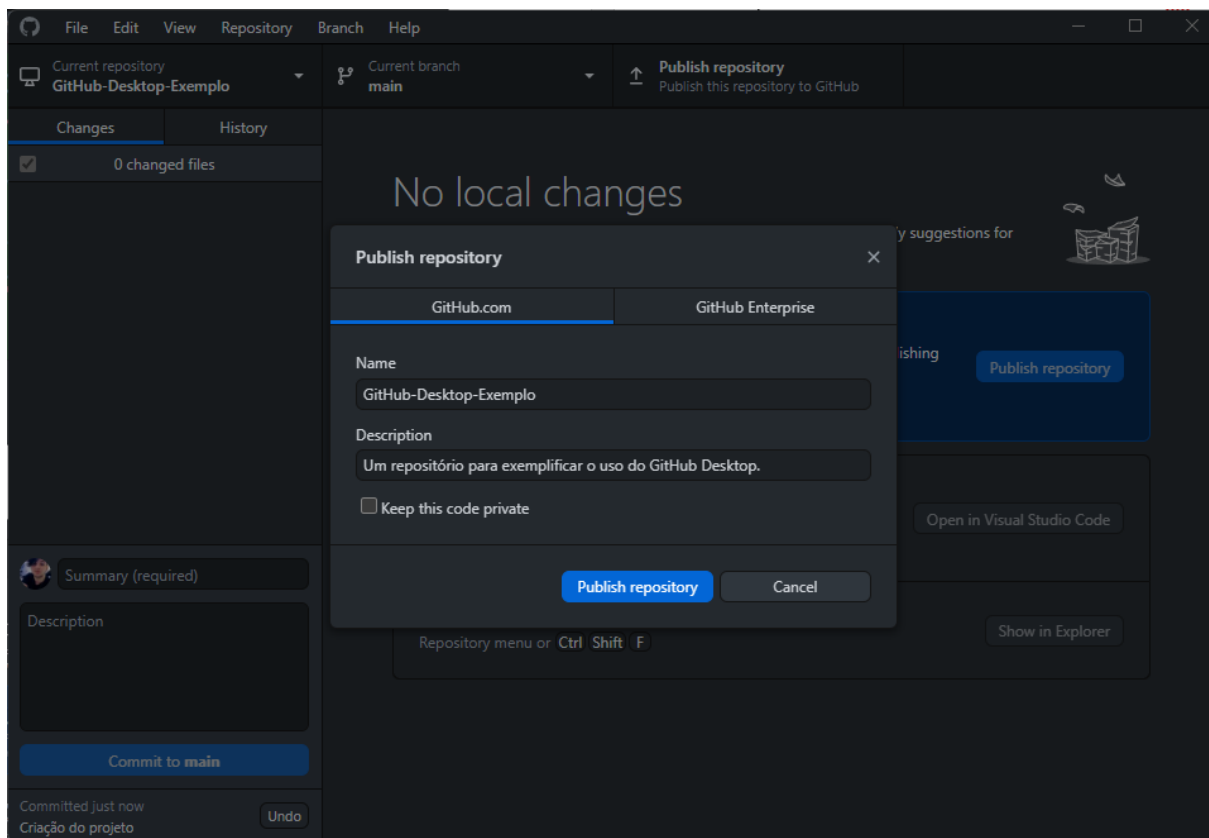


Figura 33 – Tela de publicação de repositório no GitHub do GitHub Desktop

Fonte: GitHub Desktop (2023)

Quando concluir o envio, o botão antes conhecido como **Publish repository** passará a se chamar **Fetch origin** e terá a data e a hora da última vez que houve alguma interação com o repositório remoto. E se você acessar a aba **History**, terá o histórico completo de cada **commit** realizado nesse projeto em uma interface parecida com a do site do GitHub. Se você acessar a página do seu novo repositório, perceberá que as informações estão exatamente iguais.

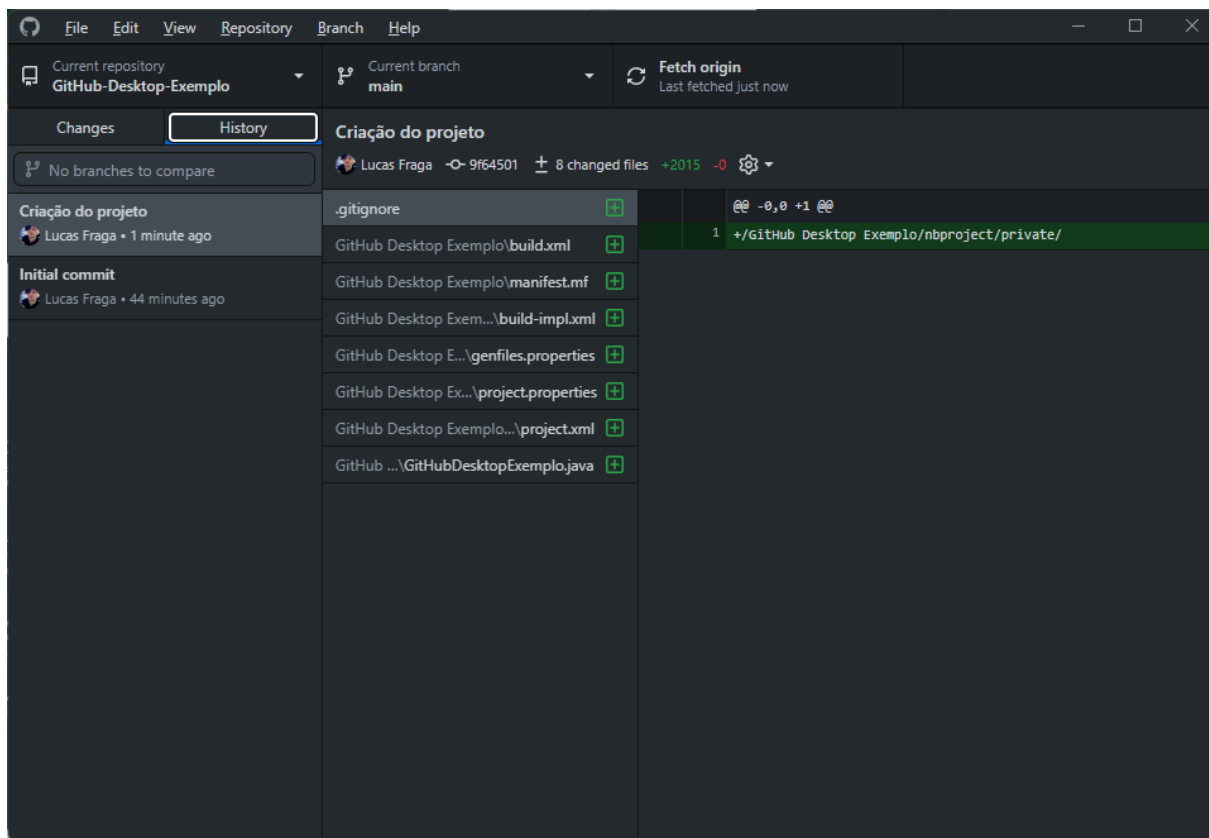


Figura 34 – Tela de histórico no GitHub Desktop

Fonte: GitHub Desktop (2023)

Sincronizando repositórios local e remoto

Agora, volte ao Apache NetBeans IDE e, mais uma vez, altere a sua classe principal, limpe os comentários e escreva um “Hello World”. O código será o mesmo utilizado no projeto anterior. Após essas mudanças, na aba **Changes**, você verá um contador **1** sinalizando que um arquivo sofreu alteração desde o último **commit**. Se você clicar no arquivo na listagem, conseguirá visualizar todas as mudanças que ocorreram nesse arquivo.

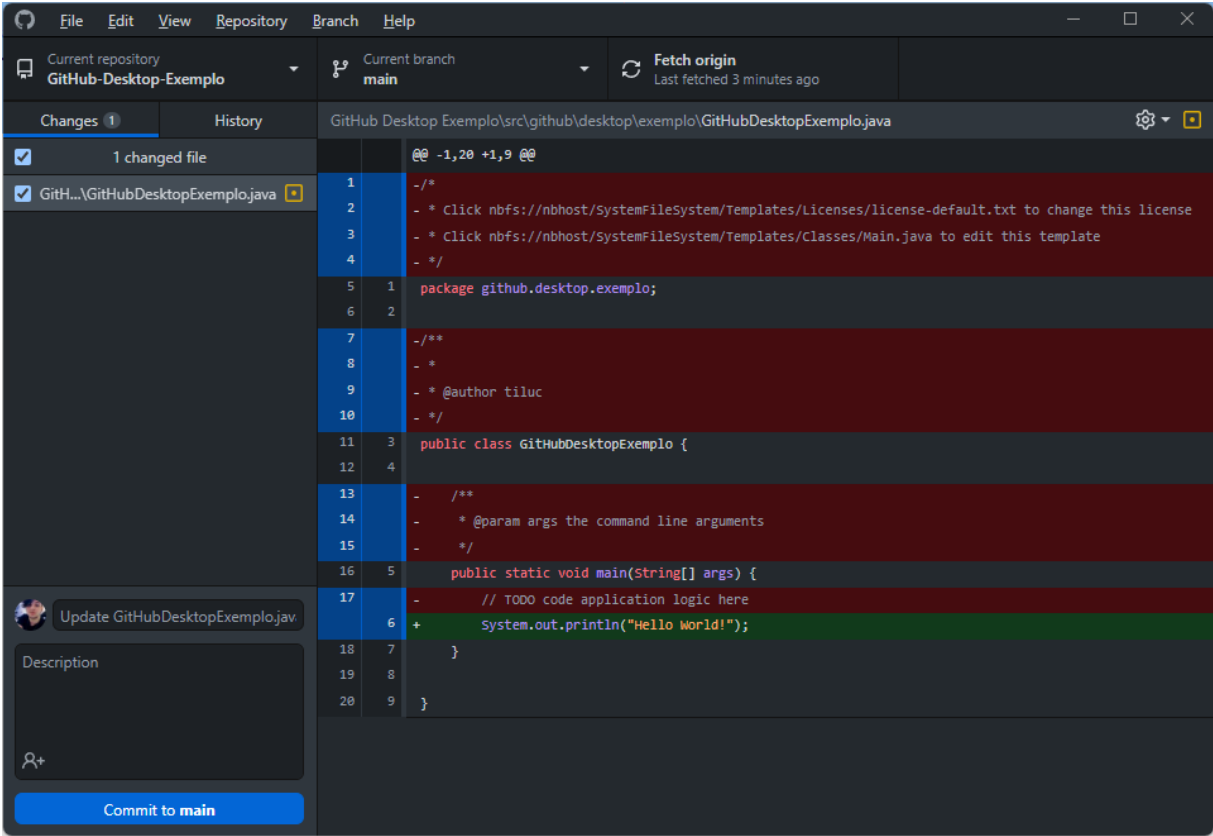


Figura 35 – Tela de histórico no GitHub Desktop
Fonte: GitHub Desktop (2023)

O processo de **commit** será repetido e chamado de “Hello World”.

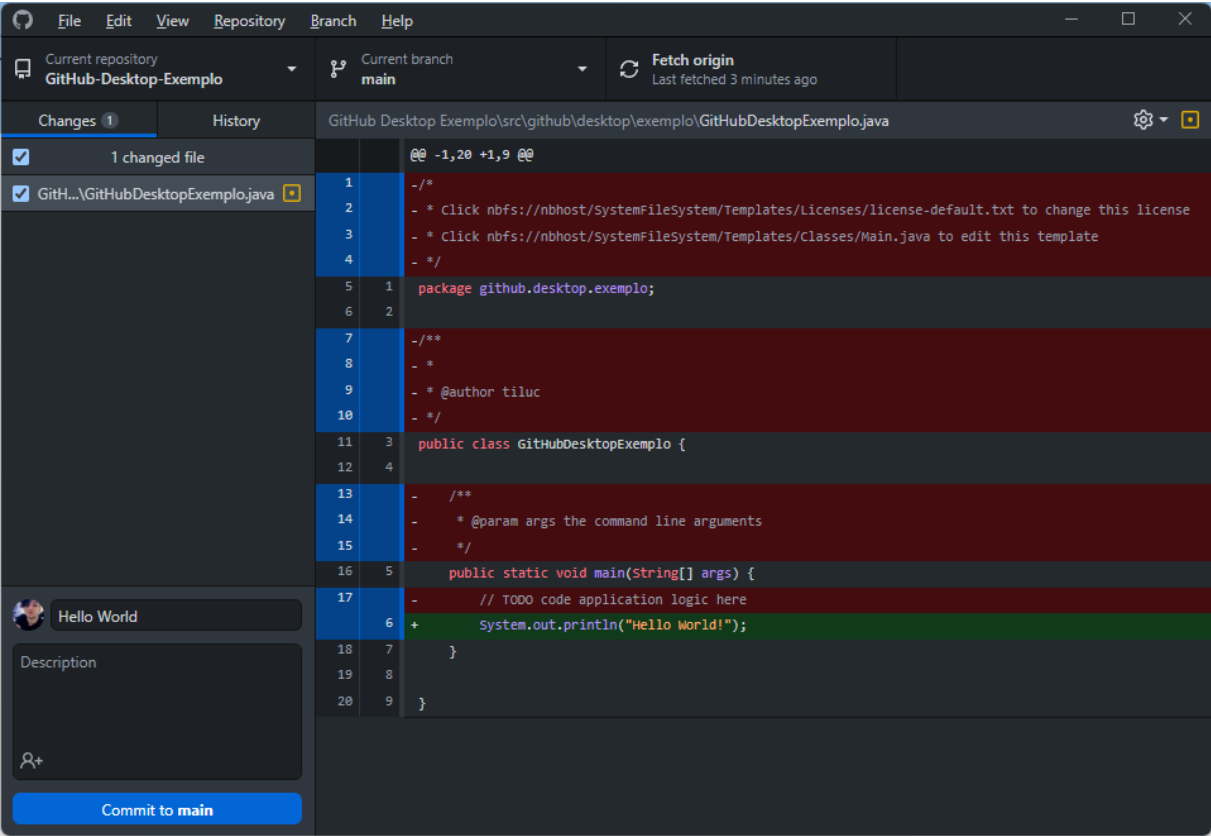


Figura 36 – Tela de histórico no GitHub Desktop

Fonte: GitHub Desktop (2023)

Após concluir o registro de **commit**, acessando a aba de **History**, você verá que, além de o novo **commit** já estar sendo listado, há um ícone de uma seta apontando para cima. Isso significa que há registros no repositório local que ainda não foram sincronizados com o repositório remoto (o mesmo contador aparecerá no botão **Push origin**). Pressione esse botão e, mais uma vez, terá o processo de **git push** acontecendo e enviando as informações para o repositório remoto.

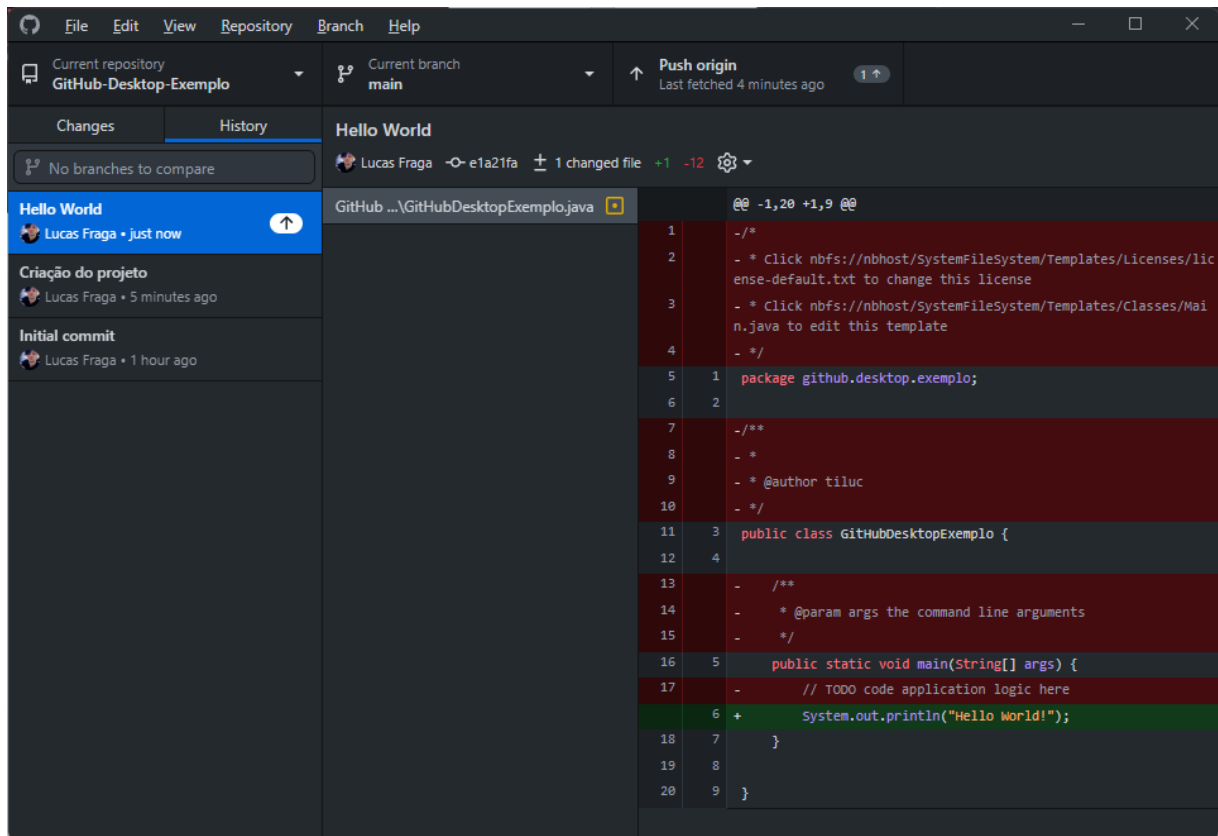


Figura 37 – Tela de histórico no GitHub Desktop

Fonte: GitHub Desktop (2023)

Agora, você fará o processo inverso. Em vez de fazer uma mudança no repositório local, você fará no repositório remoto e depois sincronizará ambos. Para isso, é possível repetir o mesmo procedimento de criação do arquivo **README.md** que você fez na prática com o GitBash.

1. Acesse a página do repositório no GitHub.
2. Clique no botão **Add a README**.
3. Caso desejar, atualize o arquivo.
4. Conclua clicando no botão **Commit new file**.

Depois, retorne ao GitHub Desktop e clique no botão **Fetch origin** para revelar uma nova ação. Como agora há mudanças no repositório remoto em comparação ao repositório local, é preciso “puxar” essas informações. Você verá que a opção **Pull**

origin e um contador de **commits** aparecerão no topo da tela.

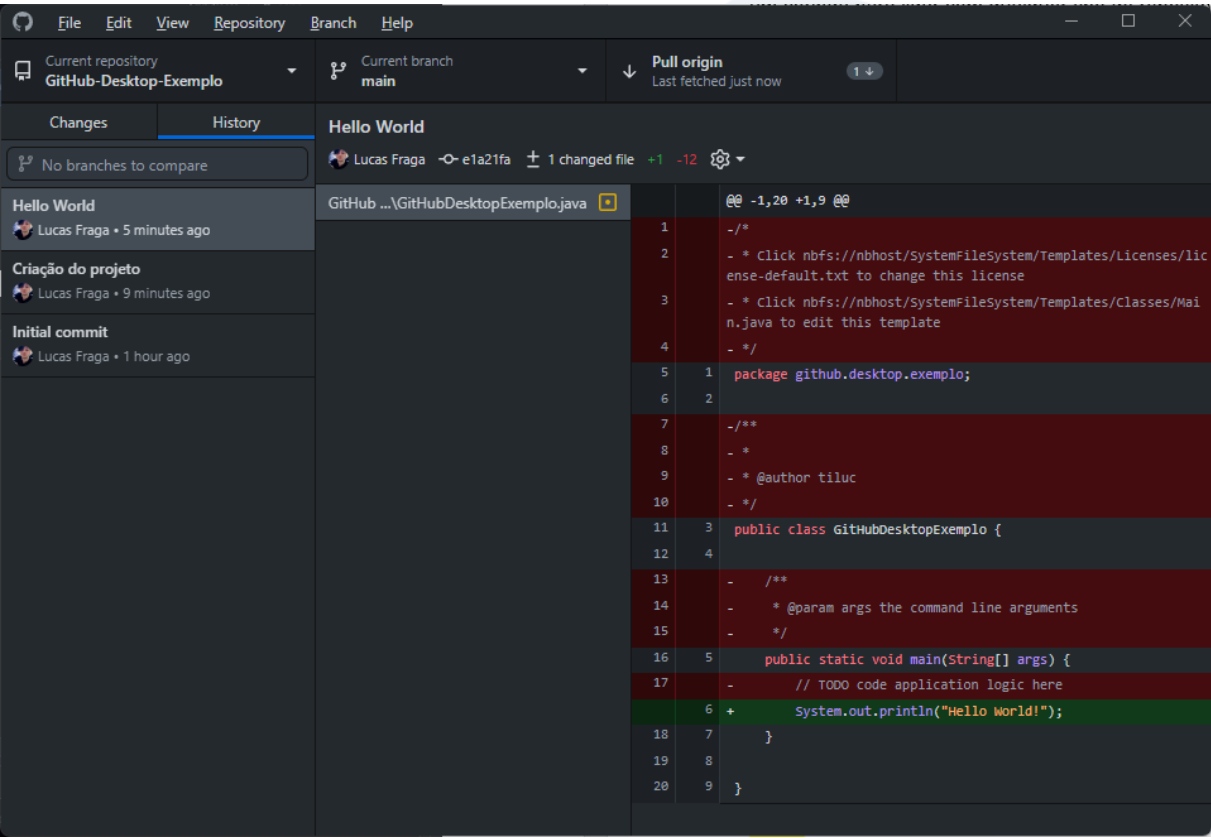


Figura 38 – Tela de histórico no GitHub Desktop

Fonte: GitHub Desktop (2023)

O GitHub Desktop iniciará o processo de **git pull** assim que você clicar nessa ação. Após concluí-lo, você poderá conferir na tela de **History** o novo **commit** adicionado e o arquivo **README.md** na pasta do diretório.

Figura 39 – Tela de histórico no GitHub Desktop

Fonte: GitHub Desktop (2023)

Revertendo um commit

Você deve se lembrar que, para reverter o último **commit** no GitBash, utilizou o comando **git revert HEAD**. Se quisesse reverter um **commit** mais antigo, precisaria fazer uma contagem do **commit** em questão em relação ao último. Bom, aqui no

GitHub Desktop, o processo ficou muito mais fácil graças à interface visual do aplicativo. Em **History**, basta você pressionar o botão direito do *mouse* sobre o **commit** que deseja reverter para revelar um menu de opções. Nele, a primeira opção será **Revert changes in commit**. Nesse exemplo, selecione o **commit** “Hello World” para reverter.

Figura 40 – Tela de histórico no GitHub Desktop

Fonte: GitHub Desktop (2023)

Após isso, um novo **commit** será adicionado com a mensagem “Revert Hello World”. Isso significa que as mudanças registradas no **commit** “Hello World” já foram desfeitas. Agora, tudo o que você precisa é sincronizar isso com o repositório remoto. Então, mais uma vez, clique na ação **Push origin**.

Figura 41 – Tela de histórico no GitHub Desktop

Fonte: GitHub Desktop (2023)

Agora que você já aprendeu os principais comandos e processos de versionamento, utilize o GitHub Desktop para clonar o repositório “Git Exemplo”, criado na prática anterior.

Encerramento

Este conteúdo começou tratando dos principais comandos utilizados no processo de versionamento de *software* com Git. Essa base servirá a você de curinga para lidar com qualquer tipo de repositório versionado em Git e em qualquer sistema operacional que você esteja operando, seja Windows, seja Linux, seja Mac.

É importante manter o registro de versionamento frequente para evitar problemas no desenvolvimento de projetos de *software*. Então, sempre que implementar alguma funcionalidade nova, após fazer os devidos testes, procure

executar, logo em seguida, os comandos **git add .** e **git commit -m “Resumo sobre o que foi implementado”**.

Você usará muitos esses comandos no dia a dia como programador. Então, para criar esse hábito, procure anotar os principais comandos em um local onde você tenha fácil acesso para consultá-los sempre que necessário e faça o versionamento de todos os seus projetos.

A ideia é criar o hábito para que o uso dos comandos se torne mais natural no seu dia a dia. Uma dica é que essas anotações estejam sempre por perto. Para isso, você pode usar *post-its*, blocos de notas ou mesmo o *software* de notas adesivas do Windows. Com o tempo, você deve tornar-se mais independente dessas ferramentas, mas tudo dependerá da sua dedicação e das práticas no dia a dia.

Após finalizar um longo dia de trabalho, não esqueça de garantir que as informações tenham sido copiadas para o repositório remoto com o comando **git push -u origin master**. Caso esteja lidando com repositórios remotos do GitHub, você ainda pode recorrer ao uso de ferramentas com interfaces gráficas, como o GitHub Desktop para realizar o processo com alguns cliques.

Para finalizar este conteúdo, veja uma última dica: aproveite os projetos de *software* que você desenvolveu até aqui e faça o versionamento deles com o GitHub. Assim, você começará a construir o seu portfólio como desenvolvedor e terá guardado todos os seus projetos em um local seguro, caso precise algum dia. Como bônus, você já pode ir praticando tudo o que aprendeu neste conteúdo.

Em caso de dúvidas, consulte seu tutor!