

Capítulo

4

Git e GitHub: Desenvolvendo Habilidades Essenciais para Colaboração e Controle de Versões

Ruan Victor Carreiro Gomes, Bruno Lima Pinheiro, João Carlos Pinto Carvalho, Jociel dos Santos Andrade e Anderson dos Reis Barros

Abstract

This book chapter thoroughly delves into the realm of Git and GitHub, providing an in-depth overview of these indispensable tools for developing essential collaboration and version control skills in the software development landscape. Starting with fundamental concepts, proceeding through installation and configuration, and emphasizing the significance of teamwork, the chapter aims to empower readers to effectively apply these skills in their development practices.

Resumo

Este capítulo de livro de minicurso explora de forma abrangente o mundo do Git e do GitHub, oferecendo uma visão detalhada dessas ferramentas cruciais para o desenvolvimento de habilidades essenciais de colaboração e controle de versões no cenário do desenvolvimento de software. Iniciando com conceitos fundamentais, passando pela instalação e configuração, e destacando a relevância da colaboração em equipe, o capítulo visa capacitar os leitores a aplicar eficazmente essas habilidades em suas práticas de desenvolvimento.

4.1. Introdução

Este minicurso estabelece o ponto de partida para uma análise abrangente do cenário do desenvolvimento de software, destacando as características do Git e do GitHub. Nesse contexto, a gestão e supervisão das alterações no código-fonte tornam-se tarefas simplificadas. Nessa situação, colaborar em projetos de equipe é análogo a conduzir habilmente uma orquestra, onde divergências são harmonizadas de forma cuidadosa, e cada etapa do processo é minuciosamente documentada. O poder intrínseco do Git e do GitHub concede aos envolvidos total controle sobre essa ampla gama de funcionalidades.

No decorrer deste capítulo aborda-se uma variedade de tópicos, incluindo a administração de código, a colaboração eficaz e o controle de versões, todos os quais são componentes essenciais que desempenham um papel importante no funcionamento das equipes de desenvolvimento. Os participantes adquirem habilidades em uma das linguagens universais dos desenvolvedores de software à medida que incorporam palavras como "commit", "push" e "pull request" em seu vocabulário.

A estrutura compreende as seguintes seções: a Seção 1.2 destaca a importância dos conceitos relacionados ao controle de versão e à colaboração em comunidades de código aberto. Subsequentemente, a Seção 1.3 explora a introdução ao Git, seguida pela Seção 1.4, que aborda o GitHub, ambas sendo ferramentas fundamentais para desenvolvedores. Por fim, a Seção 1.5 encerra o capítulo com algumas considerações finais, consolidando os princípios discutidos.

4.2. A Importância do Controle de Versão e da Colaboração em Comunidades de Código Aberto

Já foi pensado em como grandes projetos de software são desenvolvidos sem se tornarem caos de códigos confusos? Ou como as equipes de desenvolvimento distribuídas podem trabalhar juntas em projetos complexos? O controle de versão é a ferramenta que possibilita a resposta a essas questões.

Considerando as dimensões dos projetos de software modernos, que geralmente envolvem equipes de desenvolvimento distribuídas geograficamente, milhares de arquivos e milhões de linhas de código, surgem como um desafio significativo. Manter o código organizado, garantir a rastreabilidade das alterações e facilitar a colaboração eficaz nessas circunstâncias pode ser extremamente complicado [7]. Nesse contexto, o controle de versão, representado por sistemas robustos como o Git, desempenha um papel fundamental. Não se trata apenas de uma característica agradável, mas de um componente essencial no desenvolvimento de software contemporâneo [1]. O versionamento permite que os desenvolvedores trabalhem em conjunto em um único código-fonte, independentemente de sua localização geográfica. Além disso, garante o registro de cada alteração realizada, documenta as razões por trás dessas mudanças e, o mais importante, oferece a capacidade de reverter facilmente para qualquer ponto no histórico do projeto [9]. Um sistema de controle de versão robusto, como o Git, é o cerne do desenvolvimento de software moderno. Sem ele, os desenvolvedores teriam que recorrer a métodos mais arcaicos, como a cópia manual de versões de código, a criação de pastas com nomes incrementais ou depender da memória humana para monitorar todas as modificações.

Além de melhorar a eficiência, o controle de versão oferece maior segurança ao funcionar como um registro permanente de todas as mudanças no código. Ele fornece um método valioso de auditoria para identificar quando e por quem uma alteração específica foi implementada [10]. Encontrar erros, corrigir problemas e melhorar a qualidade do software dependem disso.

Todas essas vantagens do controle de versão são complementadas por uma característica ainda mais atraente no mundo do desenvolvimento de software moderno: a colaboração em comunidades de código aberto e a participação em uma "rede social" de inovadores [8]. Isso é uma extensão natural do controle de versão, permitindo conectar-se com outros entusiastas, desenvolvedores talentosos e especialistas de diversas regiões enquanto se gerencia o código. Ao registrar-se no GitHub, a maior

plataforma de colaboração e hospedagem de código-fonte do mundo, é possível juntar-se a uma comunidade de desenvolvedores, compartilhar trabalho, contribuir para projetos e aprender com outros usuários [5]. Essa é uma rede onde pessoas de diferentes locais trabalham juntas e geram novas ideias. Afinal, no GitHub, o código é social.

Ao longo do capítulo, serão discutidas as ferramentas Git e GitHub, essenciais para colaboração e controle de versão no desenvolvimento de software. Esta jornada aumentará o conhecimento, habilidades e auxiliará no gerenciamento de código de forma eficiente e eficaz.

4.3. Iniciando com o Git

Para contextualizar o Git, é necessário percorrer as raízes do desenvolvimento de software e a revolução que ele representou. Embora o Git tenha uma origem humilde, sua história é fascinante. Hoje é considerado uma das ferramentas de controle de versão mais poderosas.

No início do desenvolvimento de software, o gerenciamento de versões do código-fonte era uma das tarefas mais difíceis que os programadores enfrentavam. Antes do Git, os sistemas de controle de versão centralizados eram a norma, em que os desenvolvedores se conectavam a um servidor central para trabalhar em conjunto e o repositório principal era mantido por ele [2]. No entanto, esse método tinha limitações significativas, como pontos de falha e lentidão. Ao usar um sistema centralizado, o servidor era o principal componente do controle de versão. Qualquer problema com o servidor, como falhas de hardware, problemas de rede ou erros humanos, pode causar grandes interrupções no fluxo de trabalho de desenvolvimento. Isso pode resultar na perda de acesso ao código-fonte ou, pior ainda, na corrupção de dados, colocando em risco o projeto como um todo. As operações, como a obtenção de versões anteriores do código ou a integração de alterações, eram frequentemente lentas e sujeitas a atrasos devido ao fato de os desenvolvedores precisarem de uma conexão constante com o servidor central. A produtividade e a agilidade das equipes de desenvolvimento foram prejudicadas por isso, principalmente em projetos grandes ou com membros distribuídos. O criador do kernel Linux, Linus Torvalds, começou a desenvolver o Git por conta dessas restrições. Ele viu uma chance de desenvolver uma ferramenta de controle de versão que desafiaria os padrões estabelecidos e resolvesse esses principais problemas.

Com isso, a história¹ do Git começa no início dos anos 2000, quando Linus Torvalds, enfrentou as limitações de sua comunidade de desenvolvimento. Torvalds viu uma oportunidade para criar uma nova ferramenta de controle de versão que desafiaria as normas estabelecidas porque estava insatisfeito com as soluções existentes. Assim, o Git surgiu. Ele começou como um projeto de código aberto em 2005 e tem velocidade e descentralização como seus pilares. O Git chamou a atenção da comunidade rapidamente e ganhou uma base de usuários fiel. No entanto, como muitas histórias notáveis, o Git

¹<https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git>

teve seu início em meio a uma grande destruição criativa e uma intensa disputa. Com sua amplitude e importância no mundo do código aberto, o projeto do kernel Linux trabalhou por anos distribuindo modificações no código por meio de correções e arquivos. Porém, em 2002, o kernel Linux assumiu um caminho diferente ao usar a BitKeeper, uma DVCS (Distributed Version Control System) proprietária.

O conto teve um giro inesperado em 2005. A ruptura da conexão entre a empresa por trás do BitKeeper e a comunidade de desenvolvedores do Linux levou ao pagamento da ferramenta, obrigando a comunidade a procurar uma alternativa. O próprio arquiteto do Linux, Linus Torvalds, sentiu-se compelido a liderar o desenvolvimento de uma nova ferramenta, aproveitando as lições úteis que aprendeu com o BitKeeper. Velocidade, simplicidade, arquitetura completamente distribuída e suporte robusto para desenvolvimento não-linear com milhares de ramos paralelos eram os princípios fundamentais. Essa nova ferramenta deveria ser capaz de lidar com projetos de grande envergadura, como o núcleo do Linux, com eficiência e destreza.

Com sua rapidez, facilidade e arquitetura distribuída, o Git se estabeleceu como uma ferramenta revolucionária no mundo do desenvolvimento de software. Essencial para projetos de todos os tamanhos, ele permite um controle de versão ágil e adaptável [6]. A influência do Git transcendeu, remodelando a maneira como os desenvolvedores colaboram e gerenciam código. Desde sua fundação em 2005, o Git tem amadurecido e evoluído constantemente, mantendo os princípios que o tornaram notável [8]. Sua velocidade impressionante, eficiência em projetos extensos e capacidade de administrar desenvolvimento não linear, evidenciada por seu sistema de branches, são notáveis. Não é surpresa que, entre os sistemas de controle de versão disponíveis, o Git tenha conquistado uma parcela significativa do mercado, representando uma parte substancial dos sistemas de controle de versão em uso, com perspectivas de crescimento contínuo. Muitas organizações já abandonaram sistemas de controle de versão mais antigos em favor do Git, tornando-o seu VCS (Version Control System) preferido [7]. O Git simboliza a capacidade humana de superar desafios, inovar e, em última análise, criar algo que transforma a maneira como as comunidades de desenvolvedores colaboram e moldam o futuro do software [4].

4.3.1. Instalação e Configuração do Git

Por meio do site oficial² o Git pode ser instalado em ambientes Windows, Linux e MacOS, seguindo as instruções fornecidas pelo assistente de instalação. O objetivo desta seção é discutir em detalhes os passos essenciais necessários para instalar e configurar o Git nos ambientes Windows e Linux, fornecendo uma base sólida para começar o desenvolvimento usando essa poderosa ferramenta de controle de versão.

4.3.1.1. Ambiente Windows

- Através do Site Oficial: Para instalar o Git no ambiente Windows, é necessário acessar o site oficial do Git em <https://git-scm.com/download/win> através de um navegador web;

²<https://git-scm.com/downloads>

- **Baixar o Instalador:** Na página de downloads, será encontrado um link para baixar o instalador do Git para Windows. Ao clicar no link, o arquivo executável (.exe) será baixado para o computador;
- **Executar o Instalador:** Após a conclusão do download, o usuário deve navegar até a pasta onde o arquivo foi salvo e executá-lo. Geralmente, o nome do arquivo segue o formato Git-x.x.x-64-bit.exe, onde "x.x.x" representa a versão atual do Git;
- **Configurações Iniciais Através do Instalador:**
 - **Definir o Caminho do Git:** Ao aceitar os termos e condições do Git, o instalador permitirá escolher o local de instalação. O caminho padrão é recomendado na maioria dos casos, e o usuário pode prosseguir clicando em "Next";
 - **Seleção de Componentes:** Na próxima etapa, o instalador oferecerá a opção de selecionar os componentes a serem instalados. É aconselhável deixar todas as opções selecionadas, uma vez que são essenciais para o funcionamento adequado do Git. Clique em "Next" para continuar;
 - **Selecionar Pasta no Menu Iniciar:** O instalador permitirá escolher a pasta no menu Iniciar onde os atalhos do Git serão criados. A pasta padrão é recomendada, então o usuário pode clicar em "Next";
 - **Escolher Editor Padrão:** Nesta etapa, o usuário poderá selecionar o editor de texto padrão que o Git utilizará para abrir mensagens de commit e outras mensagens. O editor "Vim" é configurado por padrão, mas é possível escolher outro editor, se desejado;
 - **Ajustar o Nome do Branch Inicial:** O instalador perguntará se deseja ajustar o nome do branch inicial de "master" para "main." Essa mudança é uma prática comum para evitar termos historicamente carregados. Deixe a opção selecionada e clique em "Next";
 - **Ambiente PATH:** Nesta etapa, o instalador pergunta sobre a configuração do ambiente PATH. Deixe a opção "Use Git from the Windows Command Prompt" selecionada para que o Git seja acessível a partir do prompt de comando. Clique em "Next";
 - **Escolher Executável SSH:** O usuário poderá escolher o cliente SSH que o Git utilizará. A opção padrão é adequada para a maioria dos casos. Clique em "Next";
 - **Escolher Backend de Transporte HTTPS:** O instalador questionará sobre o backend de transporte HTTPS. Deixe a opção "Use the OpenSSL library" selecionada e clique em "Next";
 - **Terminal a Usar com Git Bash:** Selecione o emulador de terminal padrão a ser usado com o Git Bash. "Use MinTTY" é uma escolha comum. Clique em "Next";
 - **Comportamento Padrão do git pull:** Escolha o comportamento padrão para o comando "git pull." A opção "Default" é a recomendada. Clique em "Next";
- Após concluir todas as etapas, o Git estará instalado e o usuário terá acesso ao prompt do Git Bash no ambiente Windows;

4.3.1.2. Ambiente Linux

- Abrir o Terminal: O Terminal Linux pode ser acessado pressionando simultaneamente as teclas Ctrl + Alt + T ou procurando por "Terminal" no menu de aplicativos, dependendo da distribuição Linux utilizada;
- Atualizar o sistema: Antes de prosseguir com a instalação do Git, é aconselhável atualizar os repositórios de pacotes do sistema para obter as informações mais recentes sobre as versões disponíveis. O comando a ser executado é "sudo apt update";
- Instalar o Git: Com o sistema atualizado, o próximo passo é instalar o Git. Isso pode ser feito através do comando "sudo apt install git";
- Verificar a instalação: Após a conclusão da instalação, é importante verificar se o Git foi instalado corretamente. Isso pode ser feito digitando o comando "git --version";

Após a instalação do Git em um dos ambientes mencionados anteriormente, é necessário configurar o nome de usuário e o endereço de email. Isso pode ser realizado por meio dos seguintes comandos:

- `git config --global user.name "Seu Nome";`
- `git config --global user.email "seu@email.com";`

Essa configuração é crucial para que o Git registre com precisão as contribuições nos projetos em que estiver envolvido. É essencial inserir as informações necessárias para uma identificação adequada. Após concluir a instalação do Git em um dos ambientes e configurar corretamente o nome de usuário e o endereço de email, estará preparado para tirar o máximo proveito dessa ferramenta. Nos próximos tópicos desta seção, será abordado com mais detalhes sobre como utilizar o Git, bem como as melhores práticas que contribuirão para otimizar o trabalho de desenvolvimento.

4.3.2. Fundamentos do Git

Este tópico visa fornecer uma base sólida para entender os conceitos fundamentais do sistema de controle de versão Git. Ao iniciar, explica o que é um repositório Git e como inicializá-lo, enfatizando a importância dessa etapa crucial. Em seguida, discute os conceitos de staging e commit, mostrando como essas técnicas permitem um controle minucioso das mudanças em um projeto. Por fim, apresenta os comandos básicos do Git, como "git add" para fazer alterações, "git commit" para registrar essas mudanças no histórico e outros comandos úteis, como "git status" e "git log".

4.3.2.1. Inicialização de um repositório Git

É fundamental compreender o que realmente é um repositório Git quando se está estudando os fundamentos do Git. Simplesmente, um repositório Git é onde todo o código-fonte de um projeto é armazenado e rastreado. É como um enorme armário virtual que armazena um histórico de todas as mudanças feitas no código. Isso permite que os desenvolvedores naveguem pelo tempo e restaurem os arquivos anteriores quando necessário [8]. Este repositório é o núcleo do controle de versão Git e permite que as equipes de desenvolvimento trabalhem de forma organizada.

Além disso, é importante compreender o procedimento de inicialização de um repositório Git local, que é executado por meio do comando "git init". A Figura 4.1 mostra a criação de um repositório local na pasta "GitFundamentos" através do GitBash. Ao tomar essa iniciativa, seja ao dar início a um novo projeto ou ao desejar começar a rastrear um código já existente com o Git, o desenvolvedor estabelece um repositório local. Com essa ação, o projeto passa a ser supervisionado pelo Git, pronto para registrar com minúcia todas as mudanças e atualizações realizadas. A partir desse momento, o Git assume a responsabilidade de monitorar todas as operações que ocorrem nos arquivos do projeto, o que permite um controle exaustivo de todas as alterações que foram implementadas. A gestão eficaz do código-fonte e o desenvolvimento colaborativo dependem dessa capacidade de monitoramento. Assim, compreender esses princípios essenciais é um passo importante na jornada do domínio do Git e aproveitar suas vantagens no desenvolvimento de software.

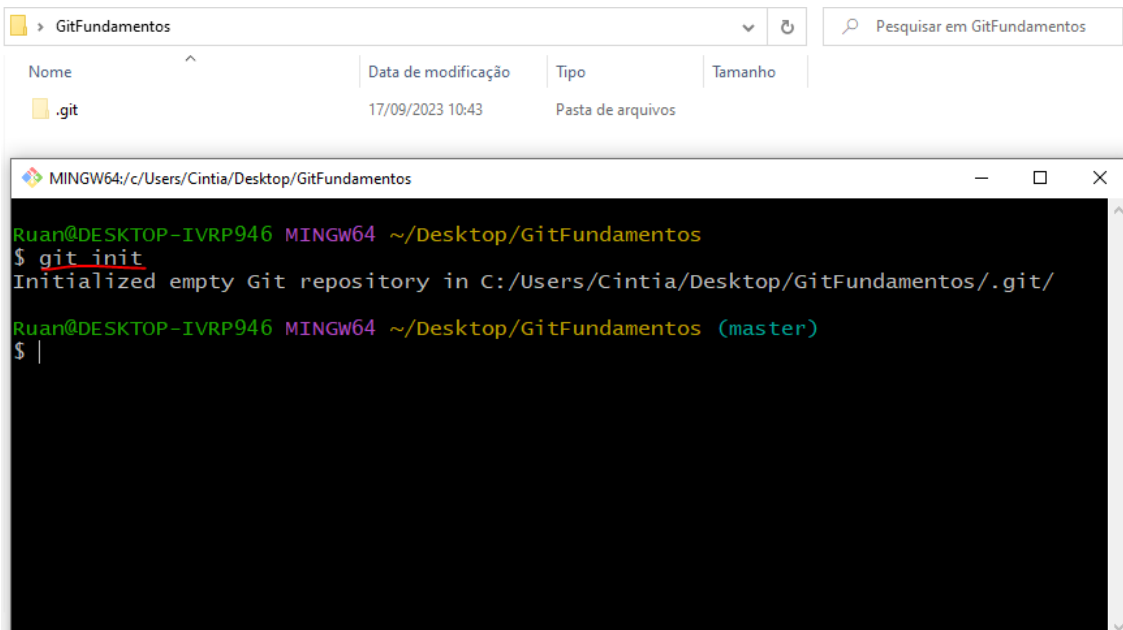


Figure 4.1. Criação de um Repositório Local. Fonte: Elaborada pelos autores

4.3.2.2. O Conceito de Staging Área e Commit

A área de preparação, conhecida como staging area, desempenha um papel fundamental no uso eficaz do Git. Nesse contexto, pode ser comparada a um espaço de seleção, onde as alterações feitas nos arquivos de um projeto são organizadas cuidadosamente antes de serem oficialmente registradas em um commit. Essa prática oferece aos desenvolvedores a oportunidade de revisar todas as modificações realizadas e, com precisão, escolher quais delas serão incorporadas a um commit específico. Essa abordagem não apenas promove uma organização estruturada, mas também se mostra altamente benéfica em cenários complexos de desenvolvimento de software.

Nessa área de preparação, em que as modificações passam por uma triagem e são organizadas antes de se tornarem permanentes no histórico de um projeto, possibilita que as equipes operem com maior eficiência, assegurando que somente as alterações desejadas sejam registradas. Esse aspecto torna-se particularmente valioso em projetos que contam com a participação de várias pessoas e contribuições, onde a clareza e a precisão no controle de versão desempenham um papel crucial para garantir o sucesso de um desenvolvimento colaborativo.

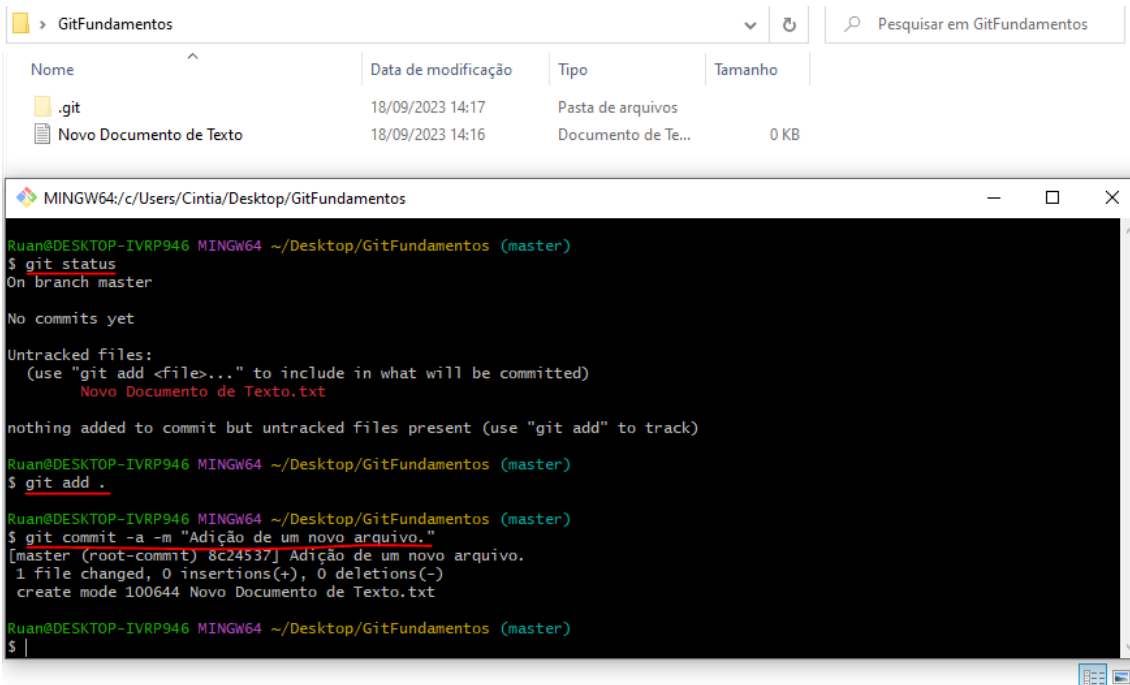
Já o processo de commit é o momento em que as alterações que foram previamente preparadas na área de staging são oficialmente registradas no histórico do repositório Git. Cada commit cria um novo ponto de referência para registrar quaisquer mudanças feitas no projeto até então. Um método recomendado é fornecer mensagens de commit significativas e descritivas que expliquem as alterações de forma simples e compreensível. Dessa forma, todos podem rapidamente entender quais mudanças foram feitas em um commit específico, facilitando a colaboração da equipe.

A Figura 4.2 ilustra visualmente o fluxo do processo, onde as modificações feitas nos arquivos no diretório de trabalho são selecionadas e movidas para a área de preparação (staging area) antes de serem registradas no repositório Git por meio de um commit. Isso cria um histórico claro e rastreável de todas as alterações feitas no projeto, permitindo que os desenvolvedores colaborem de forma mais eficiente e compreensível.



Figure 4.2. Fluxo do processo de um commit. Fonte: Elaborada pelos autores

Além disso, a Figura 4.3 apresenta um exemplo prático desse processo no prompt do Git Bash, demonstrando o uso dos comandos "git status," "git add .," e "git commit -a -m 'Adição de um novo arquivo'" para preparar e registrar alterações em um projeto Git:



```
Ruan@DESKTOP-IVRP946 MINGW64 ~/Desktop/GitFundamentos (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>.." to include in what will be committed)
  Novo Documento de Texto.txt

nothing added to commit but untracked files present (use "git add" to track)
Ruan@DESKTOP-IVRP946 MINGW64 ~/Desktop/GitFundamentos (master)
$ git add .
Ruan@DESKTOP-IVRP946 MINGW64 ~/Desktop/GitFundamentos (master)
$ git commit -a -m "Adição de um novo arquivo."
[master (root-commit) 8c24537] Adição de um novo arquivo.
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 Novo Documento de Texto.txt
Ruan@DESKTOP-IVRP946 MINGW64 ~/Desktop/GitFundamentos (master)
$
```

Figure 4.3. Demonstração do processo no Git Bash. Fonte: Elaborada pelos autores

- O comando **git status** é utilizado para verificar o estado atual do repositório, exibindo quais arquivos foram modificados e se estão prontos para o staging;
- Em seguida, o comando **git add .** é empregado para adicionar todas as mudanças ao staging area, preparando-as para o próximo commit;
- Por fim, o comando **git commit -a -m "Adição de um novo arquivo"** realiza o commit das alterações preparadas no staging area, juntamente com uma mensagem descritiva que documenta o propósito do commit;

4.3.3. Trabalhando com Branches

As branches são como "galhos" de uma grande árvore, permitindo que o projeto se divida em caminhos separados. Essa divisão possibilita que várias linhas de desenvolvimento ocorram simultaneamente, como se estivesse trabalhando em diferentes partes da floresta, sem interferências entre elas. A importância das branches se deve ao fato de que, ao construir software, é comum a necessidade de adicionar novos recursos, corrigir erros ou fazer melhorias. No entanto, fazer todas essas mudanças diretamente no projeto principal pode gerar confusão. É aí que as branches entram em cena.

Com as branches, é possível criar cópias separadas do projeto, cada uma representando uma ideia ou uma tarefa específica. Por exemplo, se deseja adicionar um novo

recurso, pode criar uma branch para isso, e se precisa corrigir um bug ao mesmo tempo, outra branch pode ser criada para a correção. Dessa forma, é possível trabalhar em cada tarefa de forma isolada, sem preocupações com interferências entre elas.

Quando todas essas tarefas estiverem concluídas, é possível reuni-las de volta ao projeto principal, como se estivesse unindo as trilhas de volta à estrada principal da floresta. Isso é conhecido como "merge" (mesclagem) e representa uma parte crucial do controle de versão. O uso adequado das branches torna o desenvolvimento mais organizado, evita confusões e, o mais importante, permite manter um histórico claro e rastreável de todas as mudanças no projeto.

Portanto, as branches funcionam como caminhos que orientam o desenvolvimento de software, garantindo a capacidade de explorar, inovar e melhorar o projeto de forma controlada e eficaz. À medida que este tópico é explorado, são apresentadas informações sobre como criar, gerenciar e utilizar branches para tornar o trabalho mais organizado e produtivo.

Na Figura 4.4, é possível visualizar um cenário típico de controle de versão com o uso de branches. A branch principal, representada como "master", constitui o tronco principal da árvore de desenvolvimento. Os círculos azuis ao longo da branch "master" representam os commits realizados nessa linha de desenvolvimento principal. Além disso, são observadas duas branches secundárias, representadas por círculos verdes e uma bolinha laranja. Cada uma dessas branches secundárias representa uma linha de desenvolvimento separada, onde novos recursos ou correções estão sendo trabalhados de forma independente.

A branch verde escura, que se destaca na figura, representa o processo de "merge" (mesclagem). Esse processo ocorre quando as mudanças efetuadas em uma das branches secundárias são integradas de volta à branch principal "master". O resultado é um único histórico de desenvolvimento que incorpora as alterações feitas nas branches secundárias.

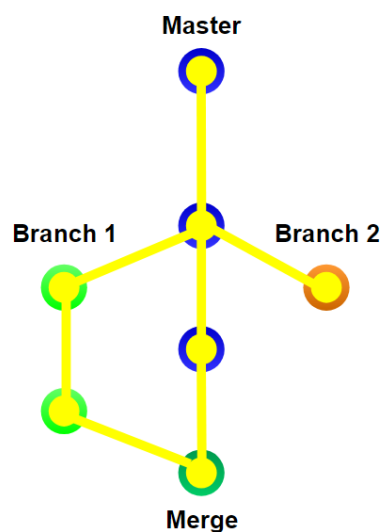
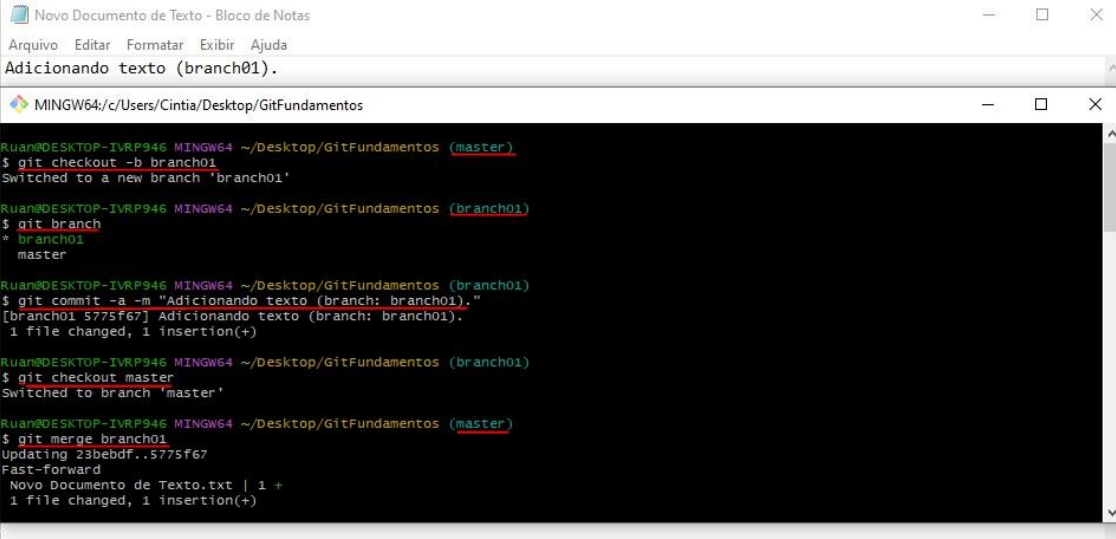


Figure 4.4. Ilustração das branches em um repositório Git. Fonte: Elaborada pelos autores

Essa representação gráfica demonstra como as branches permitem o desenvolvimento simultâneo de diferentes recursos ou correções, mantendo o histórico de cada linha de desenvolvimento de forma clara e organizada. Quando o trabalho em uma branch secundária é concluído e revisado, ele pode ser mesclado de volta à branch principal, garantindo que todas as alterações sejam incorporadas ao projeto principal de maneira controlada e eficiente. Esse método auxilia na preservação da integridade do código e facilita a colaboração entre membros da equipe.

A manipulação de branches no Git é uma parte essencial do controle de versão e do desenvolvimento colaborativo. Existem vários comandos que permitem criar, listar, alternar entre branches e mesclar alterações de diferentes branches. A Figura 4.5 ilustra visualmente, em vermelho, esses comandos no Git Bash seguidos de seus significados:



```
Novo Documento de Texto - Bloco de Notas
Arquivo Editar Formatar Exibir Ajuda
Adicionando texto (branch01).

MINGW64/c/Users/Cintia/Desktop/GitFundamentos

Ruan@DESKTOP-IVRP946 MINGW64 ~/Desktop/GitFundamentos (master)
$ git checkout -b branch01
Switched to a new branch 'branch01'

Ruan@DESKTOP-IVRP946 MINGW64 ~/Desktop/GitFundamentos (branch01)
$ git branch
* branch01
  master

Ruan@DESKTOP-IVRP946 MINGW64 ~/Desktop/GitFundamentos (branch01)
$ git commit -a -m "Adicionando texto (branch: branch01)."
[branch01 5775f67] Adicionando texto (branch: branch01).
 1 file changed, 1 insertion(+)

Ruan@DESKTOP-IVRP946 MINGW64 ~/Desktop/GitFundamentos (branch01)
$ git checkout master
Switched to branch 'master'

Ruan@DESKTOP-IVRP946 MINGW64 ~/Desktop/GitFundamentos (master)
$ git merge branch01
Updating 23bebd..5775f67
Fast-forward
 Novo Documento de Texto.txt | 1 +
 1 file changed, 1 insertion(+)
```

Figure 4.5. Comandos para gerenciar branches no Git Bash. Fonte: Elaborada pelos autores

- Primeiro, o comando **git checkout -b "nome da branch"** é usado para criar uma nova branch e mudar imediatamente para ela. Isso permite que você inicie um novo ramo de desenvolvimento com o nome especificado;
- O comando **git branch** é usado para listar todas as branches disponíveis no repositório. Isso ajuda a visualizar todas as linhas de desenvolvimento em andamento;
- Para confirmar as alterações em uma branch, você utiliza o comando **git commit -a -m "Mensagem de commit"**. Esse comando registra as alterações feitas na branch atual, fornecendo uma mensagem descritiva que explica o objetivo do commit;
- Quando é necessário alternar entre diferentes branches, o comando **git checkout nome-da-branch** é usado. Isso permite que você mude para a branch desejada e continue o trabalho nela;

- Por fim, o comando **git merge** nome-da-branch é utilizado para mesclar as alterações de uma branch em outra. Isso é particularmente útil quando você deseja incorporar o trabalho de uma branch de desenvolvimento de volta à branch principal, como a "master";

Esses comandos são essenciais para a organização do desenvolvimento de software com o Git, pois permitem a confirmação de alterações, a criação de várias linhas de desenvolvimento independentes e a integração de novos recursos ou correções de bugs no projeto principal.

4.3.4. Resolução de Conflitos

A resolução de conflitos no Git é um tópico fundamental para que os desenvolvedores possam gerenciar e trabalhar eficazmente com diferentes ramificações de um projeto. Os parágrafos a seguir apresentam explicações sobre esse tema, com o intuito de torná-lo acessível a iniciantes.

Quando vários desenvolvedores – ou até mesmo um único desenvolvedor – trabalham em partes diferentes de um projeto, é comum que as alterações feitas por um desenvolvedor entrem em conflito com as feitas por outro desenvolvedor. Esses conflitos podem ocorrer quando duas partes alteram uma seção específica de um arquivo ou quando uma parte exclui um arquivo que a outra parte está editando. Embora o Git seja capaz de detectar esses conflitos, é fundamental saber como resolvê-los corretamente.

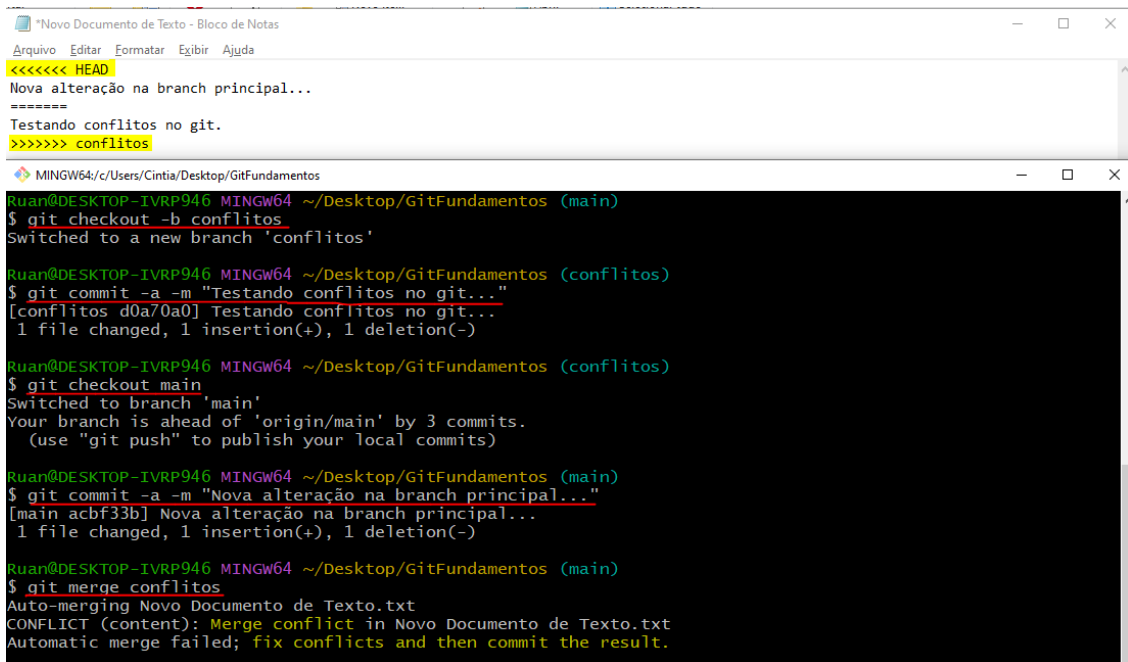
A resolução de conflitos no Git é um processo que envolve examinar as diferenças entre as alterações conflitantes e decidir quais modificações serão mantidas. Esses conflitos nos arquivos com notações especiais, como "««« HEAD" e "»»»»» branch", são marcados pelo Git. Decidir qual versão do código deve ser mantida e qual deve ser descartada é responsabilidade do desenvolvedor. Inicialmente, isso pode parecer um pouco intimidante, mas quando você o faz, ele se torna uma tarefa comum.

A primeira coisa que deve ser feita para resolver conflitos é abrir o arquivo em conflito em um editor de texto. Em seguida, o desenvolvedor verá as partes que estão em conflito marcadas. A escolha de quais partes de cada alteração serão mantidas e como o arquivo deve ficar após a resolução são as principais responsabilidades. Para garantir que a versão final seja funcional e sem erros, pode ser necessário uma compreensão profunda do contexto das alterações e da lógica do código. Após as decisões corretas, as marcações de conflito são removidas e apenas as partes do código que o desenvolvedor deseja manter são mantidas. Para registrar a resolução do conflito, o arquivo alterado é adicionado ao Git e um novo commit é criado. É fundamental fornecer uma explicação detalhada das alterações feitas durante a resolução em uma mensagem de commit. Uma vez que o conflito tenha sido resolvido e documentado, o desenvolvedor pode começar o projeto normalmente.

Na Figura 4.6, é apresentado um exemplo prático que ilustra a resolução de conflitos no Git. Nesse cenário, duas branches estão envolvidas: a branch "main" e a branch "conflitos". Cada passo do processo é acompanhado por comandos Git relevantes e explicações claras.

A branch "main" é a linha principal de desenvolvimento, enquanto a branch "con-

flitos" representa uma linha de desenvolvimento secundária. Cada uma delas tem seus próprios commits e alterações específicas. No exemplo, as alterações conflitantes ocorrem na "Linha 1" de um arquivo de texto, onde ambas as branches fazem modificações diferentes. A resolução de conflitos envolve a análise dessas diferenças, a escolha das modificações a serem mantidas e a criação de um novo commit para registrar a resolução. No exemplo, você pode ver as mensagens de commit associadas a cada passo do processo, tornando a resolução de conflitos uma tarefa clara e compreensível.



The image shows two windows. The top window is a Notepad application titled "Novo Documento de Texto - Bloco de Notas". It contains the following text: <pre><<<<<< HEAD
Nova alteração na branch principal...
=====
Testando conflitos no git.
>>>>>> conflitos</pre>. The bottom window is a terminal window titled "MINGW64/c/Users/Cintia/Desktop/GitFundamentos". It shows the following commands and output: <pre>Ruan@DESKTOP-IVRP946 MINGW64 ~/Desktop/GitFundamentos (main)
\$ git checkout -b conflitos
Switched to a new branch 'conflitos'

Ruan@DESKTOP-IVRP946 MINGW64 ~/Desktop/GitFundamentos (conflitos)
\$ git commit -a -m "Testando conflitos no git..."
[conflitos d0a70a0] Testando conflitos no git...
1 file changed, 1 insertion(+), 1 deletion(-)

Ruan@DESKTOP-IVRP946 MINGW64 ~/Desktop/GitFundamentos (conflitos)
\$ git checkout main
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 3 commits.
(use "git push" to publish your local commits)

Ruan@DESKTOP-IVRP946 MINGW64 ~/Desktop/GitFundamentos (main)
\$ git commit -a -m "Nova alteração na branch principal..."
[main acbf33b] Nova alteração na branch principal...
1 file changed, 1 insertion(+), 1 deletion(-)

Ruan@DESKTOP-IVRP946 MINGW64 ~/Desktop/GitFundamentos (main)
\$ git merge conflitos
Auto-merging Novo Documento de Texto.txt
CONFLICT (content): Merge conflict in Novo Documento de Texto.txt
Automatic merge failed; fix conflicts and then commit the result.</pre>

Figure 4.6. Resolvendo Conflitos. Fonte: Elaborada pelos autores

- **git checkout -b conflitos**: Um novo branch denominado "conflitos" é criado e, em seguida, o Git muda para essa branch. Nesse momento, o usuário está trabalhando na branch "conflitos";
- **git commit -a -m "Testando conflitos no git..."**: Um commit é realizado na branch "conflitos" com uma mensagem descritiva. Esse commit representa uma alteração específica feita nessa branch;
- **git checkout main**: A partir deste comando, o usuário retorna à branch principal "main". Agora, está novamente na branch principal;
- **git commit -a -m "Nova alteração na branch principal..."**: Um novo commit é criado na branch principal "main" com outra mensagem descritiva. Esse commit representa uma alteração diferente feita na branch principal.
- **git merge conflitos**: Neste ponto, o objetivo é mesclar as alterações da branch "conflitos" de volta à branch principal "main". No entanto, devido à existência de alterações conflitantes na mesma parte do arquivo em ambas as branches, isso resultará em um conflito que precisará ser resolvido.

Após a resolução do conflito, o desenvolvimento pode continuar de forma organizada e controlada, assegurando que as alterações sejam incorporadas ao projeto principal de maneira eficaz. Embora a resolução de conflitos possa parecer uma habilidade desafiadora no início, com o tempo, os desenvolvedores ganharão confiança e eficácia ao lidar com essas situações. Isso permite que o projeto prossiga de maneira fluida e com a colaboração de todos os envolvidos.

4.4. GitHub e Colaboração em Equipe

O GitHub³, um serviço de hospedagem de código colaborativo construído sobre o sistema de controle de versão Git, desempenha um papel fundamental no mundo do desenvolvimento de software de código aberto (OSS). Esta plataforma é muito mais do que apenas um repositório para código-fonte; é uma comunidade vibrante que reúne milhões de desenvolvedores e projetos de OSS [3]. O desenvolvimento de software de código aberto sempre foi uma parte vital da indústria de tecnologia, alimentando inúmeras inovações e avanços. Através do GitHub, desenvolvedores de todo o mundo podem colaborar em projetos, contribuir com código, relatar e discutir problemas (bugs) e melhorar continuamente o software que é usado por milhões de pessoas em todo o mundo. [8].

Entretanto, a colaboração distribuída e descentralizada presente no universo do desenvolvimento de software de código aberto (OSS) também traz consigo desafios significativos. À medida que as comunidades de desenvolvimento OSS crescem, é preciso enfrentar a complexa tarefa de coordenar esforços entre os desenvolvedores, monitorar e registrar mudanças em diversos repositórios e manter padrões de qualidade consistentes.

É nesse cenário desafiador que o GitHub se destaca como uma plataforma fundamental para a comunidade OSS. O GitHub oferece uma plataforma robusta que fornece suporte a uma ampla gama de recursos essenciais para o desenvolvimento colaborativo. Entre esses recursos, destacam-se:

- **Controle de Versão:** O GitHub baseia-se no sistema de controle de versão Git, o que significa que oferece uma maneira eficiente e confiável de rastrear e gerenciar alterações no código-fonte. Isso é crucial para manter um histórico claro e rastreável de todas as mudanças realizadas ao longo do tempo;
- **Gerenciamento de Distribuição de Software:** O GitHub não se limita apenas ao armazenamento de código-fonte. Ele também facilita a distribuição de software, permitindo que os desenvolvedores compartilhem seus projetos com facilidade, tornando-os acessíveis a uma ampla audiência;
- **Rastreamento de Bugs e Problemas:** O GitHub oferece um sistema integrado de rastreamento de problemas e bugs. Isso permite que os desenvolvedores relatem problemas, sugiram melhorias e acompanhem o progresso das correções. Essa funcionalidade é vital para manter um software de alta qualidade;
- **Integração Contínua (CI) e Entrega Contínua (CD):** Por meio do GitHub Actions, os desenvolvedores podem automatizar a integração contínua e a entrega contínua

³<https://github.com/>

de seus projetos. Isso significa que as alterações de código são testadas automaticamente e, se aprovadas, podem ser implantadas sem intervenção manual, acelerando o ciclo de desenvolvimento;

Todas essas ferramentas e recursos são essenciais para manter o ritmo acelerado de produção e manutenção de software de alta qualidade em um ambiente de código aberto. O GitHub desempenha um papel crucial ao fornecer uma plataforma unificada que atende às necessidades da comunidade OSS, permitindo que os desenvolvedores colaborem de maneira eficaz, resolvam problemas rapidamente e entreguem software de alta qualidade para usuários em todo o mundo. Portanto, o GitHub não é apenas um repositório de código; é o epicentro da inovação e colaboração no desenvolvimento de software de código aberto.

4.4.1. Entrando no GitHub

Este tópico oferece orientações essenciais sobre como utilizar o GitHub após a criação de uma conta no site oficial⁴. Com o registro concluído, os usuários estão prontos para explorar as funcionalidades da plataforma, hospedar projetos de código e colaborar com outros desenvolvedores. A seguir, destacamos as etapas cruciais relacionadas à criação de repositórios (repos) e ao processo de clonagem de repositórios já existentes:

4.4.1.1. Criando um Repositório

Um repositório, frequentemente chamado de "repo", é onde o código-fonte e os arquivos relacionados a um projeto são armazenados. Criar um repositório no GitHub é o ponto de partida para compartilhar e colaborar com outros desenvolvedores. Aqui estão as etapas para criar um novo repositório:

- Após efetuar o login, o usuário deve acessar o seu perfil, clicando na foto de perfil localizada no canto superior direito e selecionando a opção "Your profile" (Seu perfil);
- Para criar um novo repositório, no perfil do usuário, é necessário clicar no botão "Repositories" (Repositórios) e, em seguida, escolher "New" (Novo) para iniciar o processo de criação do repositório;
- Na página de configuração do repositório, é necessário fornecer os dados essenciais, como o nome do repositório, uma descrição, a escolha entre torná-lo público ou privado, a opção de iniciar com um arquivo README (opcional) e outras configurações pertinentes;
- Após preencher os detalhes, o usuário deve clicar em "Create repository" (Criar repositório). Dessa forma, o novo repositório estará pronto para receber código e arquivos;

⁴<https://github.com/signup>

Após a criação, o repositório estará disponível no GitHub, e o usuário poderá começar a adicionar arquivos, gerenciar o código-fonte e colaborar com outros desenvolvedores. A Figura abaixo mostra visualmente o passo a passo visto anteriormente.

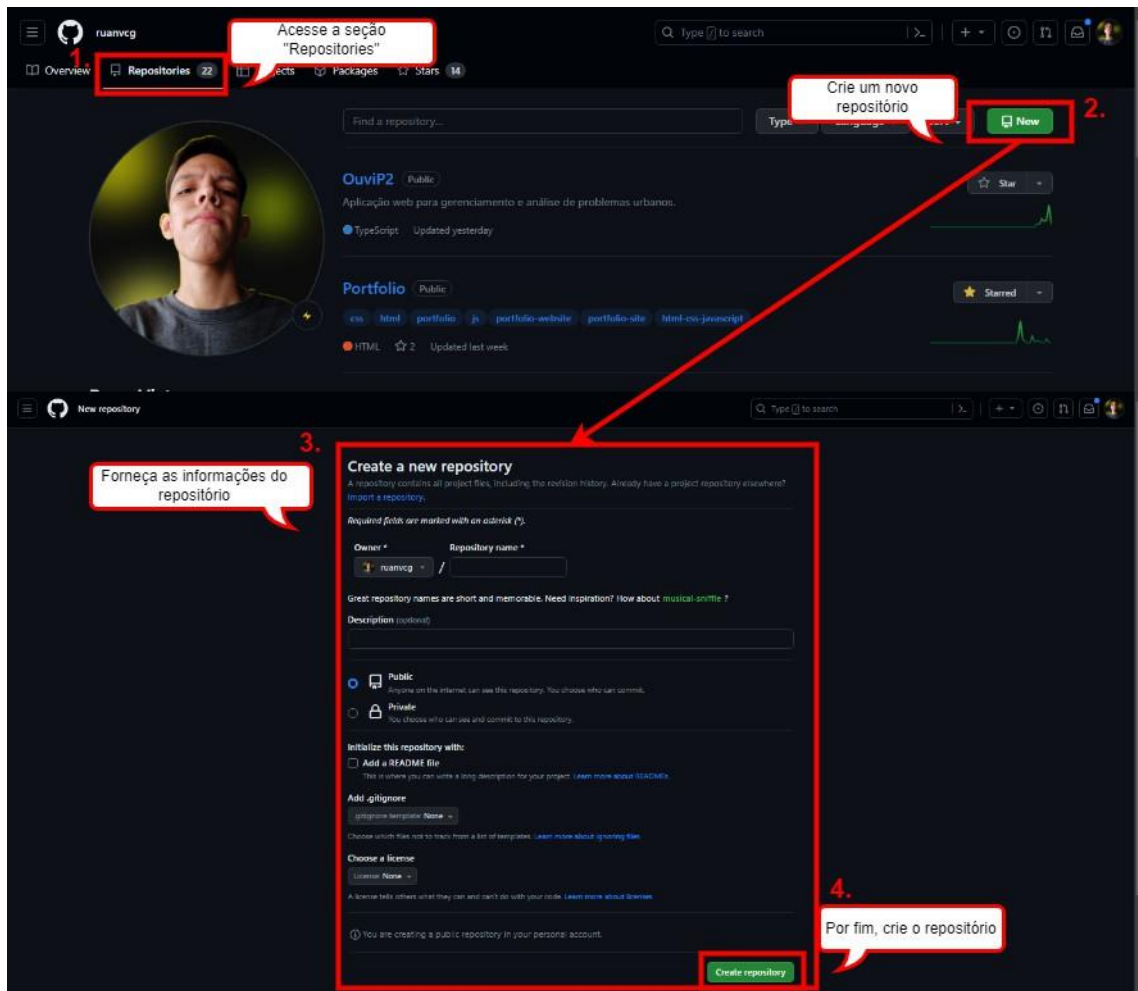


Figure 4.7. Criando um novo repositório. Fonte: Elaborada pelos autores

4.4.1.2. Clonando um Repositório Existente

Clonar um repositório existente no GitHub é um processo fundamental que permite aos usuários criar uma cópia local de um projeto em seu próprio ambiente de desenvolvimento. Isso possibilita que eles trabalhem no código, façam modificações e contribuam para o projeto de maneira eficiente. A seguir, um guia detalhado com todas as etapas necessárias para clonar um repositório existente:

- **Localizar o Repositório:** O usuário deve navegar até o repositório desejado. A barra de pesquisa na parte superior do GitHub pode ser utilizada para encontrar um repositório específico;

- Obter a URL do Repositório: Após clicar em "Code", uma janela pop-up aparecerá, mostrando a URL. É importante certificar-se de que a opção "HTTPS" esteja selecionada, pois essa é a forma mais comum de clonagem. A URL pode ser copiada clicando no ícone de área de transferência ao lado dela, ou é possível optar por "Download ZIP" para baixar o repositório como um arquivo ZIP compactado;
- Abrir o Terminal (Linux/Mac) ou Git Bash (Windows): Agora, o terminal ou Git Bash deve ser aberto no computador. É fundamental garantir que o Git esteja instalado e configurado corretamente no sistema;
- Clonar o Repositório: No terminal ou Git Bash, digite o seguinte comando: **git clone (URL do repositório desejado)**. Dessa forma, o repositório será clonado com sucesso;

Concluindo com êxito, o repositório do GitHub foi clonado para o computador do usuário, e agora ele está preparado para iniciar o desenvolvimento do projeto localmente. É fundamental manter em mente que ao efetuar alterações no código, é possível submeter essas modificações de volta ao repositório original utilizando as operações de "commits" e "pushes", simplificando assim a colaboração em equipe e a gestão de versões dos projetos.

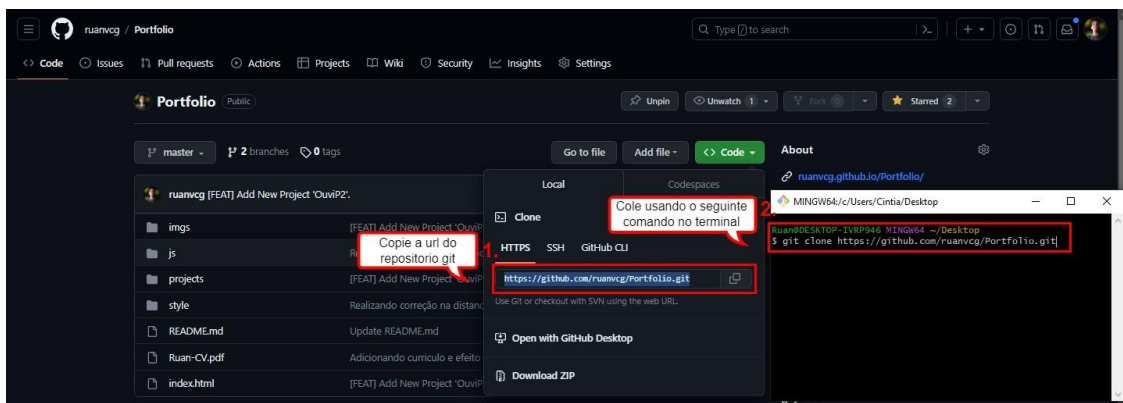


Figure 4.8. Clonando um repositório. Fonte: Elaborada pelos autores

4.4.1.3. Contribuindo com Repositórios

Neste tópico, será abordado detalhadamente como os usuários podem contribuir com repositórios já existentes no GitHub. Contribuir para projetos de código aberto é uma parte fundamental da colaboração na plataforma, permitindo que desenvolvedores de todo o mundo trabalhem juntos em projetos comuns. Abaixo, são apresentados os principais passos para contribuir de maneira eficaz (veja a Figura 4.9 para uma representação visual das principais funções):

- Encontrar um Projeto de Interesse: O primeiro passo para contribuir é encontrar um projeto que desperte interesse. O GitHub possui milhões de repositórios abertos, abrangendo uma ampla variedade de tecnologias e tópicos. É possível utilizar a

barra de pesquisa na parte superior do GitHub para localizar repositórios específicos ou explorar listas de projetos populares;

- Explorar o Repositório: Após identificar um projeto de interesse, é importante explorar o repositório correspondente para compreender melhor seu propósito, estrutura e diretrizes de contribuição. Geralmente, os repositórios contêm um arquivo "README" que oferece informações essenciais sobre o projeto, como sua finalidade, instruções de configuração e orientações para contribuição;
- Criar um Fork do Repositório: Para iniciar a contribuição, é necessário criar um "fork" (cópia) do repositório original. Isso criará uma duplicata do projeto na própria conta do usuário no GitHub, possibilitando trabalhar nele de forma independente;
- Clonar o Fork para o Ambiente Local: Após criar o "fork," o próximo passo é clonar o repositório duplicado para o ambiente de desenvolvimento local utilizando o Git. Essa ação permite que o usuário faça modificações no código e realize testes em seu próprio ambiente;
- Efetuar as Alterações: O desenvolvedor precisa efetuar as alterações requeridas no código, aderindo às melhores práticas de programação e às orientações específicas do projeto;
- Criar uma Solicitação de Mesclagem (Pull Request): Após concluir as alterações e realizar os testes adequados, é hora de criar uma "solicitação de mesclagem" no repositório original. Esse procedimento é realizado por meio da plataforma GitHub e permite que os mantenedores do projeto analisem e revisem as alterações propostas;
- Aguardar a Aprovação: Posteriormente, após a revisão e aprovação das alterações pelos mantenedores do projeto, a contribuição será incorporada ao repositório principal. Com isso, o colaborador terá concluído com sucesso o processo de contribuição;

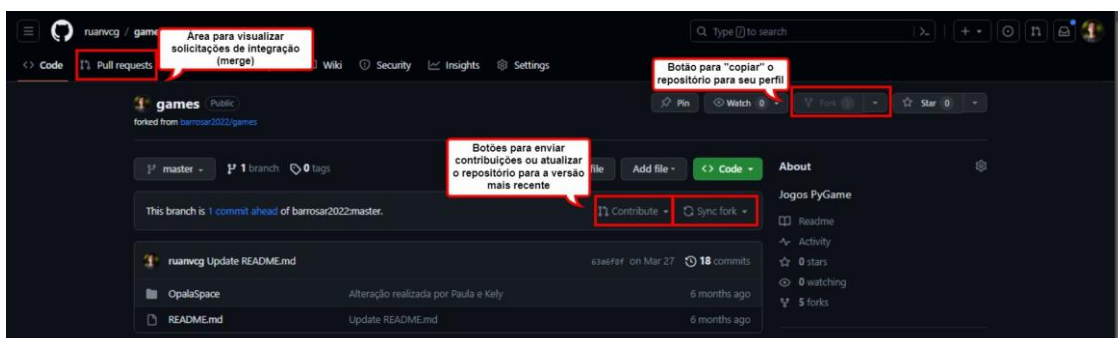


Figure 4.9. Contribuindo em repositórios. Fonte: Elaborada pelos autores

Contribuir para projetos de código aberto no GitHub é uma oportunidade valiosa para desenvolvedores adquirirem experiência significativa, enriquecerem seus conhecimentos ao interagir com outros membros da comunidade e se integrarem a uma rede

global de profissionais do desenvolvimento de software. É crucial manter um comportamento respeitoso, aderir estritamente às diretrizes estabelecidas pelo projeto e demonstrar abertura para receber e incorporar feedback. Essas práticas não apenas facilitam uma colaboração eficaz, mas também enriquecem a experiência, proporcionando um ambiente de aprendizado contínuo e gratificante.

4.5. Considerações Finais

Este minicurso explora profundamente o universo do Git e do GitHub, fornecendo uma visão abrangente dessas ferramentas essenciais para o desenvolvimento de habilidades de colaboração e controle de versões. Inicialmente, estabelece uma base conceitual sólida, esclarecendo os termos e fundamentos relacionados ao Git e ao GitHub. Em seguida, segue uma abordagem prática, detalhando passo a passo como instalar e configurar o Git, ao mesmo tempo em que explora suas funcionalidades centrais.

Ao longo deste minicurso, enfatiza-se a importância da colaboração em equipe e do controle de versões no desenvolvimento de software moderno. Destaca-se como o GitHub se tornou uma plataforma fundamental para desenvolvedores e equipes de software, possibilitando a colaboração global em projetos de pequeno e grande porte. Ao final, o objetivo é inspirar os participantes a aplicarem essas habilidades em suas práticas de desenvolvimento, capacitando-os para contribuir efetivamente em projetos de código aberto e melhorar suas capacidades de colaboração e controle de versões. Promover uma colaboração saudável e eficaz no desenvolvimento de software é uma responsabilidade compartilhada pela comunidade científica, formuladores de políticas e toda a sociedade, e este minicurso representa um passo na direção de alcançar esse objetivo.

Referências

- [1] Blauw, F. F. (2018). The use of git as version control in the south african software engineering classroom. In *2018 IST-Africa Week Conference (IST-Africa)*, pages Page 1 of 8–Page 8 of 8.
- [2] de Alwis, B. and Sillito, J. (2009). Why are software projects moving from centralized to decentralized version control systems? In *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, pages 36–39.
- [3] Decan, A., Mens, T., Mazrae, P. R., and Golzadeh, M. (2022). On the use of github actions in software development repositories. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 235–245.
- [4] Elsen, S. (2013). Visgi: Visualizing git branches. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4.
- [5] Liberty, J. and Galloway, J. (2021).
- [6] Parizi, R. M., Spoletini, P., and Singh, A. (2018). Measuring team members' contributions in software engineering projects using git-driven technology. In *2018 IEEE Frontiers in Education Conference (FIE)*, pages 1–5.
- [7] Singh, V., Singh, A., Aggarwal, A., and Aggarwal, S. (2021). Devops based migration aspects from legacy version control system to advanced distributed vcs for deploying micro-services. In *2021 IEEE International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*, pages 1–5.
- [8] Spinellis, D. (2012). Git. *IEEE Software*, 29(3):100–101.

- [9] Xu, X., Cai, Q., Lin, J., Pan, S., and Ren, L. (2019). Enforcing access control in distributed version control systems. In *2019 IEEE International Conference on Multimedia and Expo (ICME)*, pages 772–777.
- [10] Zhao, Y., Zhou, Y., Hu, C., Zhang, X., and Zhang, Z. (2018). Gitbased version control for beamline control system at the shanghai synchrotron radiation facility. In *2018 5th International Conference on Systems and Informatics (ICSAI)*, pages 134–138.