Ministério da Educação – MEC
Secretaria de Educação Superior – SESu
Programa de Educação Tutorial – PET

Universidade Federal Fluminense – UFF
Escola de Engenharia – TCE
Grupo PET do Curso de Eng. de Telecomunicações – PET-Tele

Tutoriais PET-Tele

Introdução ao sistema de controle de versão Git

(Versão: A2025M01D07)

Autores: João Guilherme Coutinho Beltrão

Tutor: Alexandre Santos de la Vega

Niterói – RJ
Janeiro / 2025

Lista de Figuras

Sumário

Capítulo 1

Introdução

O Programa de Educação Tutorial (PET) [Proa], do Ministério da Educação (MEC), exige que os grupos PET desenvolvam atividades que contemplem, de forma indissociável, itens de Pesquisa, de Ensino e de Extensão. Além disso, os grupos devem estimular uma evolução positiva dos seus integrantes, dos demais alunos do seu curso de graduação, do próprio curso e da sua instituição.

Nesse sentido, o PET-Tele [Gru21] procura desenvolver atividades e/ou atender a demandas que cumpram tais exigências.

A seguir, são apresentadas as motivações e o objetivo para o trabalho em questão.

1.1 Motivações

Dentro do grupo PET-Tele, muitos projetos são feitos em grandes grupos, durante longos períodos de tempo. Com isso, naturalmente aparece a necessidade de organizar vários documentos e, às vezes, comparar versões dos mesmos arquivos, em momentos diferentes. Para atender a essa necessidade, o grupo organizou um estudo sobre métodos de versionamento, em especial o Git, que pode ser considerado o mais popular deles, no momento.

1.2 Objetivo

O objetivo a ser alcançado com esse tutorial é mostrar o uso de sistemas para armazenamento, compartilhamento e versionamento de *software*, É dada ênfase ao sistema Git, abordando-se sua história e importância. Além disso, também serão estudados ambientes de hospedagem de código como o GitHub e o GitLab.

Capítulo 2

Versionamento de *software*

2.1 Definição

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

2.2 Benefícios

Com o versionamento, o usuário consegue ter controle total de seu projeto e organizá-lo da melhor forma possível. Além disso, versionar arquivos protege o usuário, quando eles são perdidos ou corrompidos, bem como facilita a procura por *bugs* (defeitos de projeto) em programas. Algumas outras vantagens de se versionar um projeto são:

- Análise de ameaças.
- Reparação de *bugs*.
- Alterações de arquitetura.
- Fortalecimento do trabalho colaborativo.
- Atualizações estéticas.

Porém, versionar um projeto manualmente tem muitas desvantagens. Primeiramente, salvar todos os estados de um código é muito trabalhoso. Algumas versões podem ser perdidas, o que não é o ideal. Outro problema ocorre quando se trabalha em equipes, já que cada pessoa pode salvar versões diferentes do seu próprio código, podendo causar conflitos no momento da união de todos os códigos da equipe. Para resolver esses problemas foram criados os *softwares* de controle de versão [Gua20], o que será abordado a seguir.

Capítulo 3

Sistema de controle de versão (SCV)

Um sistema de controle de versão, também chamado de SCV, é um *software* que auxilia o programador a fazer o versionamento de seu projeto. Nele, cada pequena mudança do código será salva, sem o risco de perder o arquivo, além de facilitar o processo de armazenamento e compartilhamento dos arquivos. Foram criados dois tipos de *software* de controle de versão: centralizado e distribuído.

3.1 Vantagens de um SCV

Algumas das vantagens de se utilizar um SCV são as seguintes:

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

3.2 SCV centralizado

No SCV centralizado, quando o programador salva um estado do código, o SCV envia o arquivo para um repositório local, conectado ao mesmo servidor que o computador de trabalho está usando. Dessa forma, um grupo de programadores, que estejam trabalhando no mesmo projeto, conseguirão salvar todos os arquivos em um único lugar e compará-los, além de poderem acessar todas as versões anteriores desses arquivos. A desvantagem desse tipo de versionamento é a dependência ao servidor do repositório. É necessário estar conectado constantemente para acessar os arquivos ou criar arquivos novos. O tipo de SCV distribuído, citado a seguir, não apresenta esse problema.

3.3 SCV distribuído

No SCV distribuído, o armazenamento do estado do código pelo programador pode ser feito em um repositório local dentro do próprio computador. Posteriormente, esse arquivo será direcionado a um repositório remoto, onde será feito o armazenamento de todos os documentos, sendo facilmente acessado pelo grupo de programadores.

Capítulo 4

Sistema de controle de versão Git

4.1 Apresentação

O Git [Git20] é um sistema de controle de versão distribuído, criado, em 2005, por Linus Torvalds (criador do sistema operacional Linux). Ele é, possivelmente, o SCV mais usado no mercado. O Git ficou popular por alguns motivos, entre os quais:

- É um *software* Open Source [Ini], diferentemente de seu principal concorrente da época (o BitKeeper [Bit]).
- Possui um desempenho muito superior aos outros SCV da época.
- É distribuído, o que é incontestavelmente melhor do que os SCV centralizados.

4.2 Configuração

A seguir, é explicado como fazer o *download* e a instalação do Git, bem como a vinculação de usuário.

4.2.1 *Download*

Por ser um *software* Open Source, o Git é totalmente gratuito. Para baixá-lo, basta entrar no *website* do Git [Git20] e iniciar o *download*, copiar um dos *URLs* seguintes, de acordo com seu sistema operacional, ou clicar no nome do sistema operacional escolhido, para ir direto ao *website*.

Lista de URLs para *download*:

- *URL* de *download* para Linux:

  ```
  https://git-scm.com/download/linux
  ```

- *URL* de *download* para Windows

  ```
  https://git-scm.com/download/win
  ```

- *URL* de *download* para MAC

  ```
  https://git-scm.com/download/mac
  ```

4.2.2 Instalação

Após o *download*, é necessário realizar a instalação. Isso é explicado a seguir, para algumas plataformas diferentes.

- **Linux**: para a instalação no Linux, se for desejado instalar através de um instalador binário, pode-se geralmente fazê-lo através da ferramenta básica de gerenciamento de pacotes que vem com a distribuição usada. No Fedora, por exemplo, pode-se executar o seguinte comando: *$ sudo yum install git-all*. Instruções para instalação em outros sistemas Unix podem ser encontradas pelo *URL* de *download* do Linux citado acima.
- **Windows**: para a instalação no Windows, deve-se somente executar o arquivo baixado.
- **MAC**: para a instalação no MAC, deve-se simplesmente rodar o Git a partir do terminal pela primeira vez. Se o Git não estiver instalado, o sistema pedirá para instalar.

4.2.3 Vinculação de usuário

Após a instalação, deve-se abrir o terminal (ver Seção 4.9) e digitar os seguintes comandos: *git config –global user.name "nome"* e *git config –global user.e-mail "endereço"*. Isso termina a configuração, vinculando o nome e a conta de *e-mail* do usuário à conta do Git.

4.3 Terminologia e comandos

Com o Git instalado, é preciso conhecer a sua terminologia e alguns dos seus comandos, para começar a utilizá-lo [Sek20].

Alguns termos e comandos básicos são os seguintes:

1. **Repositório**: é o nome do lugar onde os arquivos serão salvos. Se for um repositório local, será localizado na própria máquina. Se for remoto, será hospedado em uma plataforma *on-line*.
2. ***Init***: usando o comando *git init*, o usuário cria ou reinicializa um repositório local para o arquivo que estiver sendo acessado no momento. Isso é mostrado na Figura 4.1.
3. ***Status***: com o comando *git status*, o Git irá mostrar se ocorreu alguma alteração no repositório e o que pode se fazer com elas. Isso aparece na Figura 4.2.
4. ***Add***: com o comando *git add "nome do arquivo"*, o programador adiciona uma das modificações mostradas pelo *status* na fila de *commits*, como ilustrado na Figura 4.3.
5. ***Commit***: com o comando *git commit*, o Git faz o *"commit"* das mudanças na fila de *commits*. Isso significa enviá-los para o repositório local. Porém, para fazer o *commit* é necessário enviar um comentário junto a ele. Existem duas formas de fazer isso. Uma delas é usando o comando mostrado acima, onde o Git levará a uma outra tela, onde deverá ser adicionada a mensagem e, em seguida, selecionar *"esc"* e digitar *":wq"* para sair. Isso pode ser visualizado na Figura 4.4. A outra forma é usar *git commit -m "comentario sobre a mudança"*, onde o Git já enviará o comentário junto ao *commit*, como mostra a Figura 4.5.

6. ***Log***: usando o comando *git log*, o usuário consegue visualizar o histórico de todos os *commits* feitos no repositório acessado. Isso é ilustrado na Figura 4.6.

7. ***Diff***: com o comando *git diff*, o usuário consegue ver os estados das mudanças feitas no repositório inteiro ou em um arquivo específico (colocando o nome do arquivo depois do comando) no momento e após o último *commit*.

8. ***Push***: usando o comando *git push*, o Git envia o conteúdo do repositório local para o repositório remoto, dependendo de qual tipo de repositório foi configurado. Isso é mostrado na Figura 4.7.

9. ***Pull***: com o comando *git pull*, o Git busca e mescla o repositório remoto com o repositório local, deixando-os iguais. Isso pode ser visto na Figura 4.8.

10. ***Help***: com o comando *git help "comando"*, o programador é redirecionado para uma página de manual sobre o comando escolhido. Pode-se usar também o comando *git help*. Dessa forma, o Git dará a lista de todos os comandos e uma pequena frase sobre o que cada um deles faz.

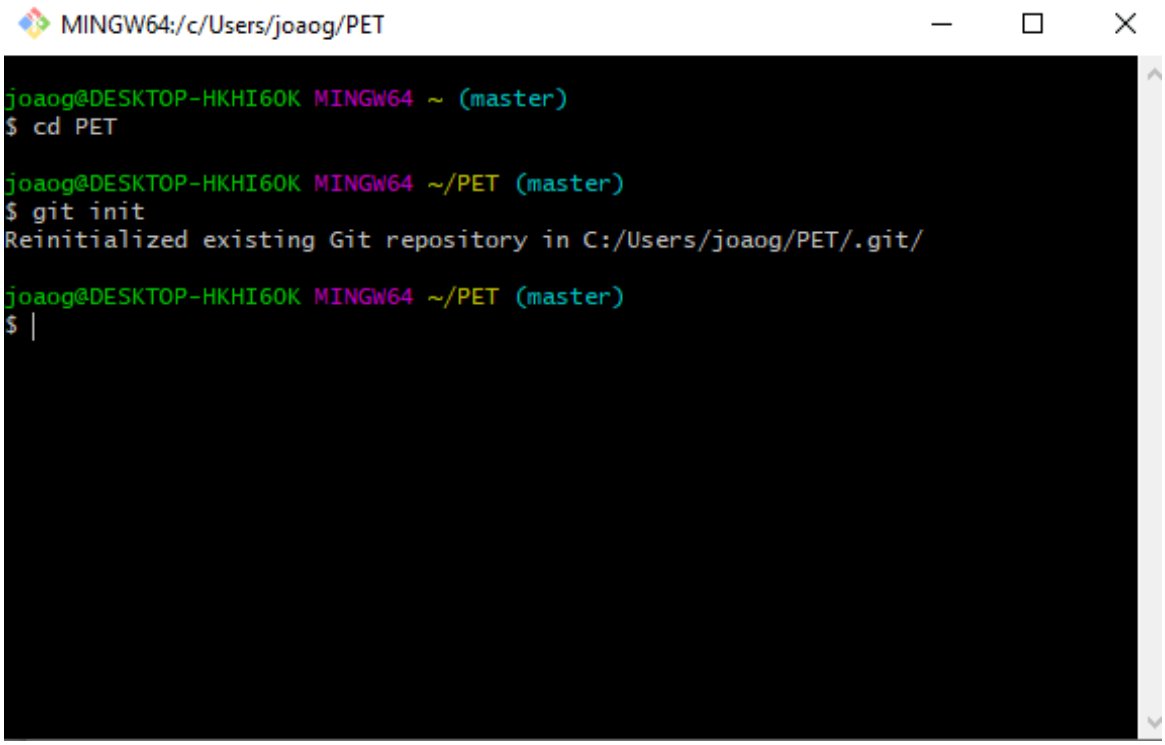

Figura 4.1: Comando *git init*.

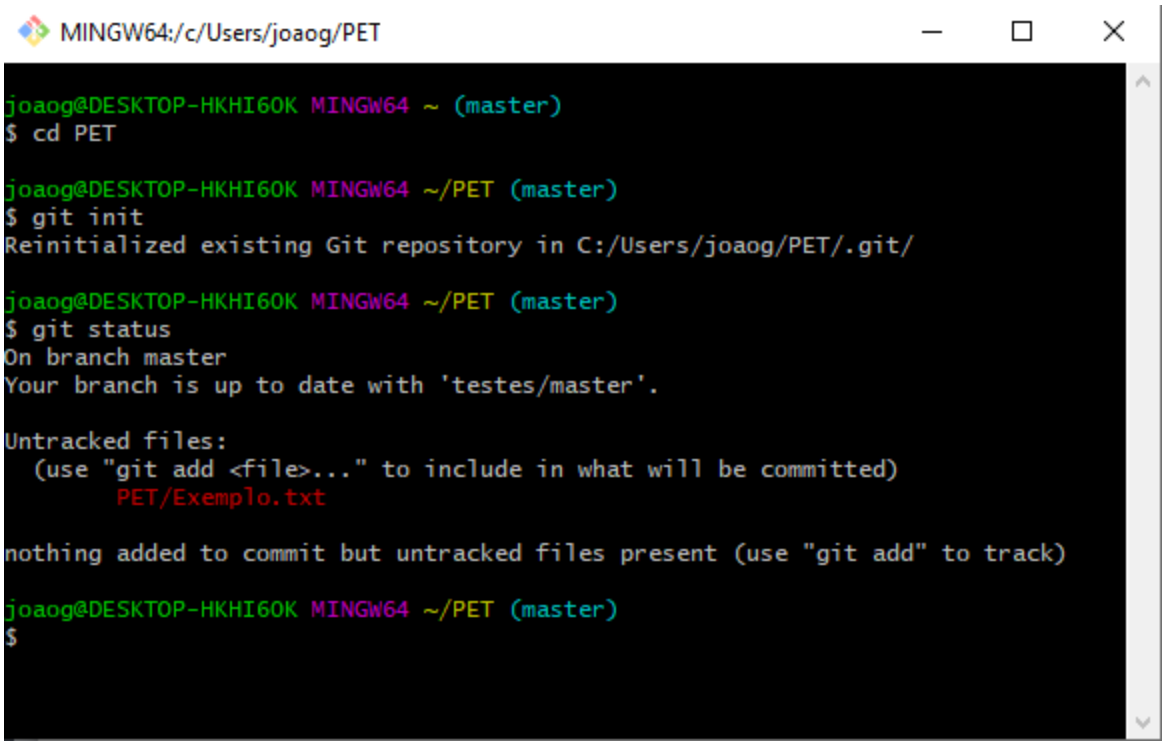

Figura 4.2: Comando *git status*.

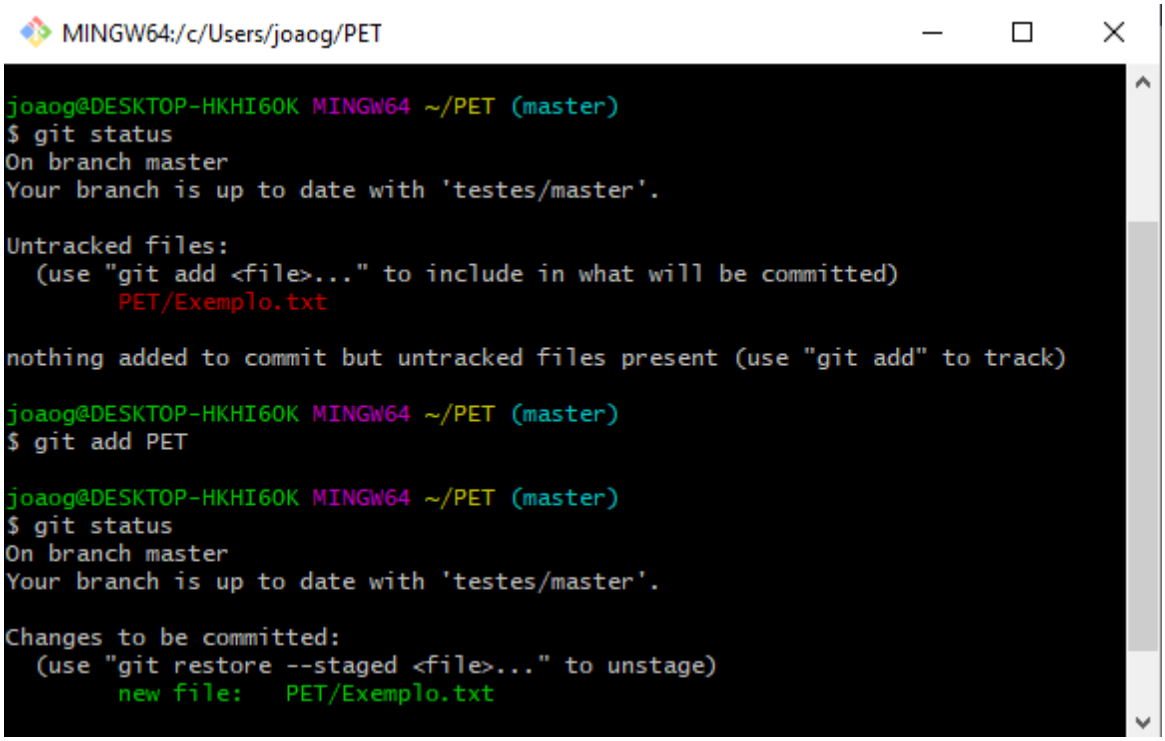

Figura 4.3: Comando *git add*.

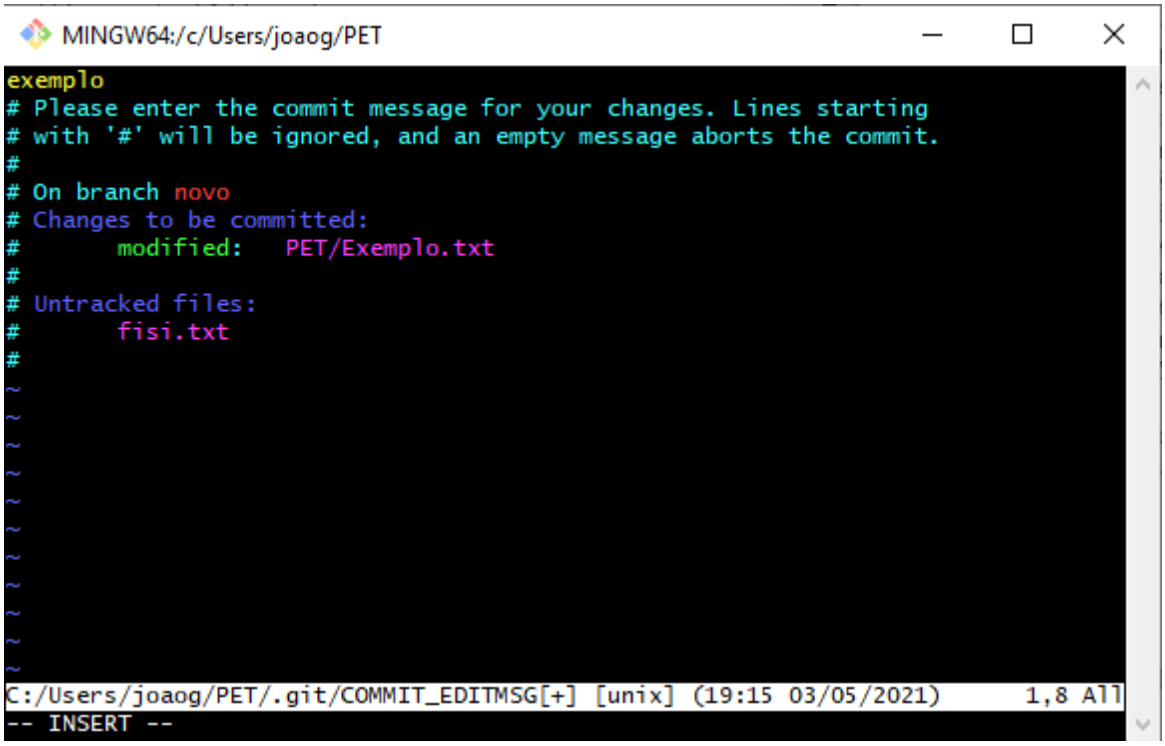

Figura 4.4: Tela para inserção de comentário.

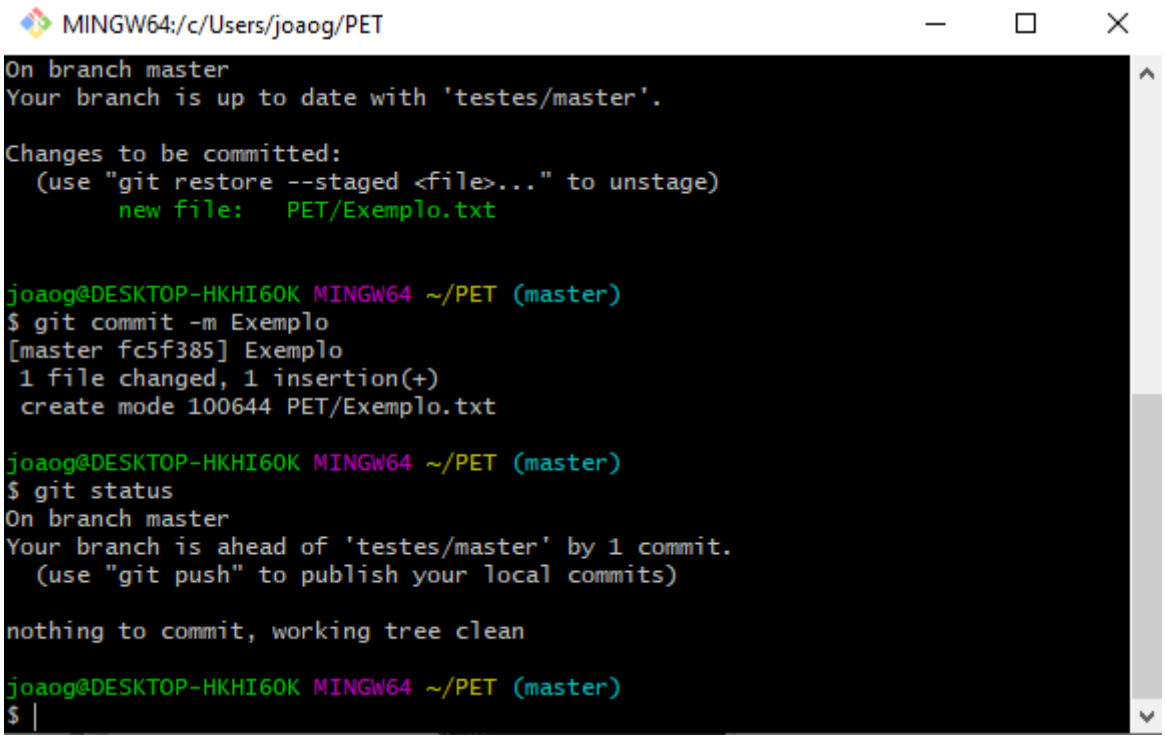

Figura 4.5: Comando *git commit* com opção *-m*.

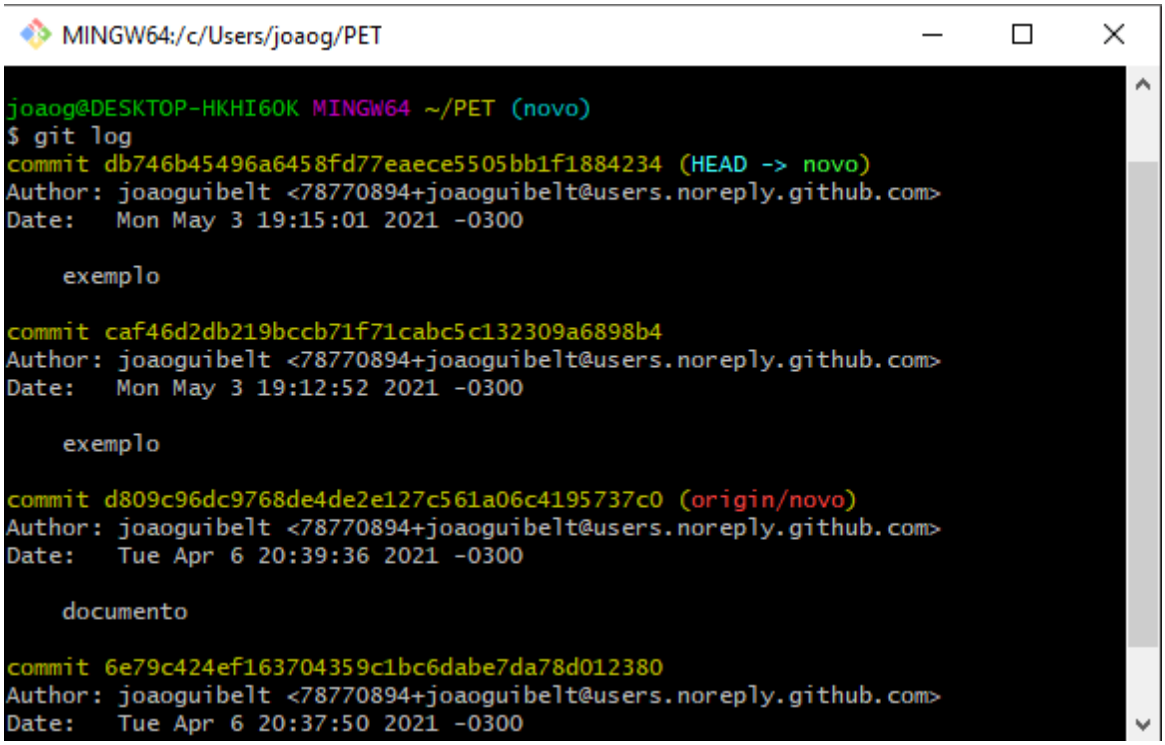

Figura 4.6: Comando *git log*.

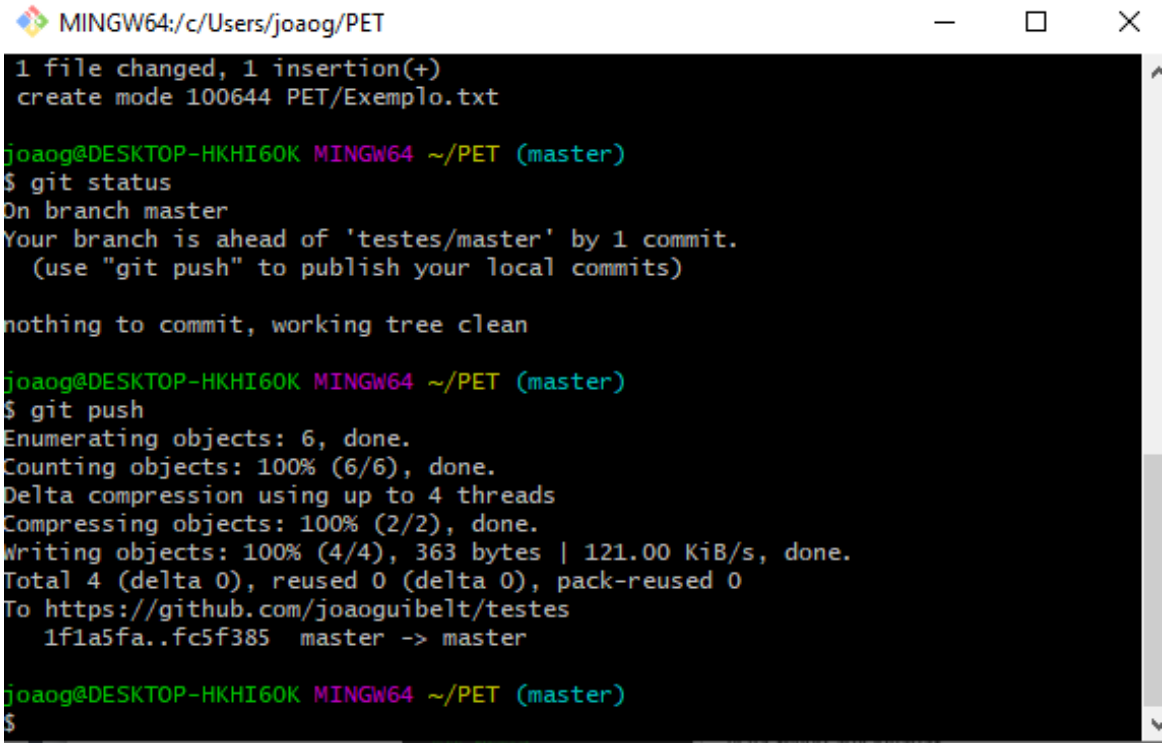

Figura 4.7: Comando *git push*.

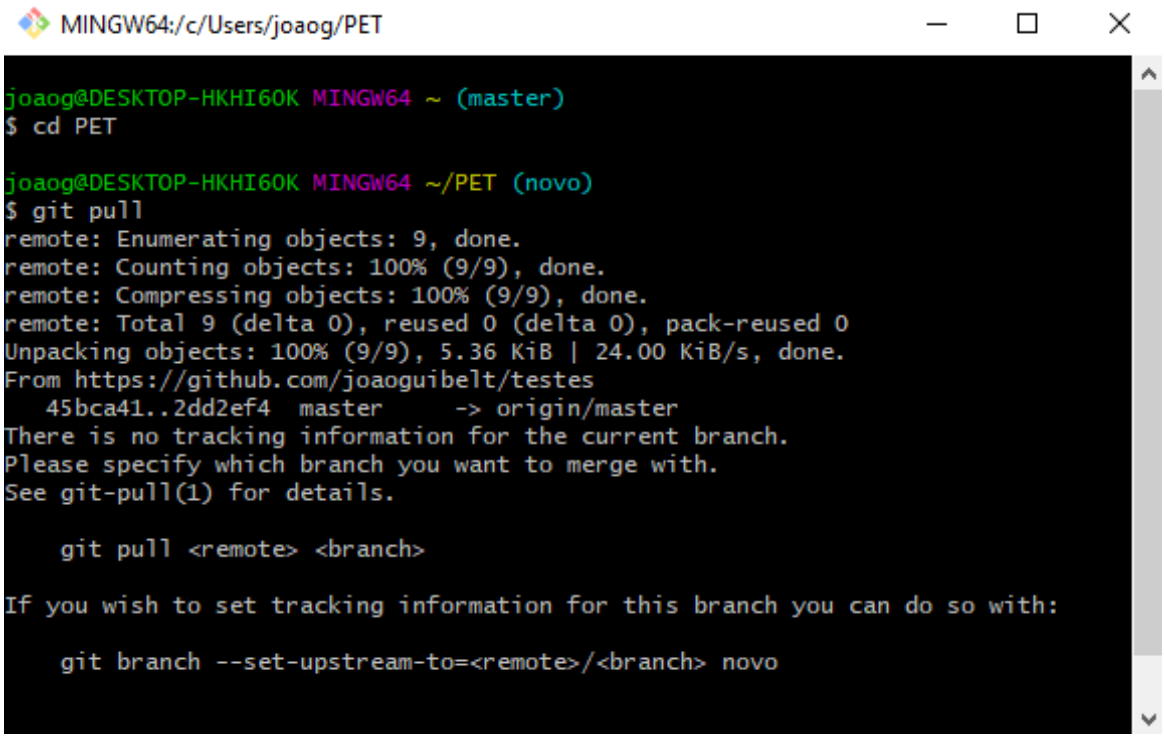

Figura 4.8: Comando *git pull*.

4.4 *Branch*

Outro conceito muito importante para otimizar o uso do Git são as *branchs*. Elas representam ambientes de desenvolvimento diferentes para o mesmo projeto, onde um usuário pode trabalhar no mesmo código que outro programador, mas sem que ambos interfiram um no trabalho do outro. Para usá-las deve-se conhecer alguns comandos.

Para criar uma nova *branch* pelo Git, é só usar o comando *git branch "nome da branch"*.

Para checar as *branchs* ativas, deve-se usar o comando *git branch*.

Quando as *branchs* forem checadas, aparecerá um asterisco ao lado daquela que estiver sincronizada para receber os códigos. Para trocar isso, deve-se usar o comando *git checkout "nome da branch a sincronizar"*. Assim que o programador fizer o *git push*, os *commits* serão enviados para essa *branch* [Fer19]. Isso é exemplificado na Figura 4.9.

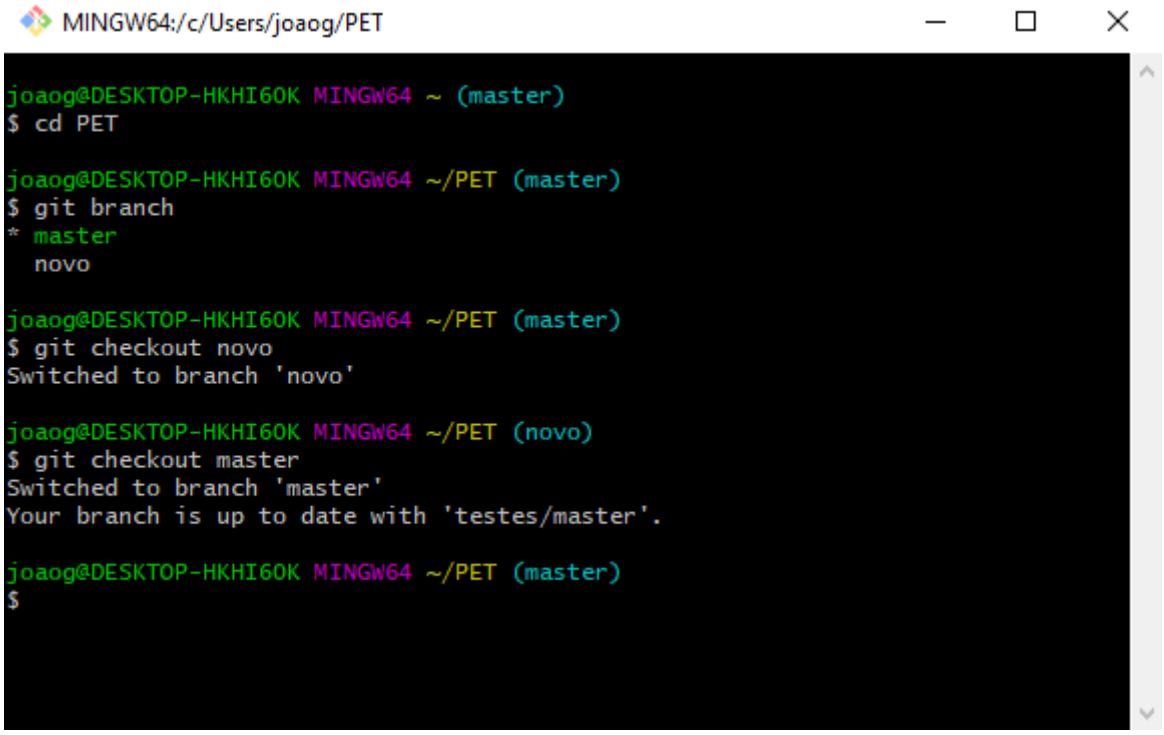

Figura 4.9: Comandos sobre as *branchs*.

Porém, para que uma nova *branch* seja levada para o repositório remoto, é necessário que ela tenha permissão para fazer isso (o que se aplica para a *branch* principal também). Para isso, na hora de usar o comando *push*, deve-se usar *git push origin "nome da nova branch"*. Assim, nas próximas vezes que for desejado fazer o *push* nessa *branch*, será necessário apenas usar o *git push* (igual à *branch* principal).

4.5 *Merge*

Sabendo usar as *branchs*, é preciso, agora, aprender a mesclá-las. Para isso, usa-se o comando *git merge*, onde a *branch* atual será fundida com a *branch* principal.

4.6 *Pull requests*

Um conceito importante, quando se está trabalhando com o Git em equipe, é o *pull request*. Os *pull requests* são pedidos que o usuário pode fazer, para conseguir fazer alterações em repositórios remotos, quando ele não possui permissão para tal. Fazendo um *pull request*, o usuário deixa uma sugestão de modificação para um repositório. Em seguida, um dos contribuintes desse repositório pode aceitar essa mudança e implementá-la.

4.7 *Reset*, *restore* e *revert*

Outros comandos importantes de se conhecer são os seguintes: *reset*, *restore* e *revert*. Com eles, o usuário consegue desfazer alterações no repositório. Porém, os comandos são usados em diferentes situações [Atl], tais como:

- ***Reset***: o comando *git reset* é usado em arquivos que foram adicionados pelo comando *git add*, para desfazer o *add* e tirá-los da linha de *commits*.
- ***Restore***: o comando *git restore "nome do arquivo"* é usado para restaurar um arquivo para sua versão anterior, descartando qualquer mudança feita nele que não foi adicionada para a linha de *commits*.
- ***Revert***: o comando *git revert "código do commit"* reverte o *commit* referenciado, cancelando qualquer alteração feita nele. O código do *commit* pode ser conseguido usando o comando *git log*.

4.8 Encontrando *bugs*

Em projetos muito grandes, com equipes numerosas, trabalhando no mesmo código, é normal que *bugs* (erros de projeto) aconteçam entre os *commits* feitos. Logo, é importante saber encontrá-los. Para isso, o Git disponibiliza um comando que ajuda nesse processo, que é o *"bisect"*.

O comando *"git bisect"* é usado para descobrir em qual *commit* ocorreu o erro indesejável no código. Para usá-lo é necessário seguir alguns passos. Primeiramente, deve-se usar o comando *git bisect start*, para o Git começar o processo, como mostra a Figura 4.10.

Com o processo iniciado, é preciso informar ao Git que o *commit* atual possui um erro, usando o comando *git bisect bad*, conforme a Figura 4.11.

Em seguida, é necessário fornecer um parâmetro que permita ao Git encontrar o *commit* mais antigo com erro. Isso é equivalente a indicar um *commit* que esteja sem o erro. Para tal, usa-se o comando *git bisect good "código do commit"*, como na Figura 4.12. Com isso, o programa irá mostrar o código de um *commit* para ser testado se possui o erro, o que é localizado na parte sublinhada da Figura 4.13.

Depois de testar o código, deve-se notificar o Git, por meio do comando *git bisect bad*, se o *commit* possuir erro, ou do comando *git bisect good*, se ele não possuir, como exemplificado na Figura 4.14.

Por fim, basta repetir o processo, até o programa informar qual o primeiro *commit* que apresenta o erro e, assim, achar o *bug*, conforme a Figura 4.15.

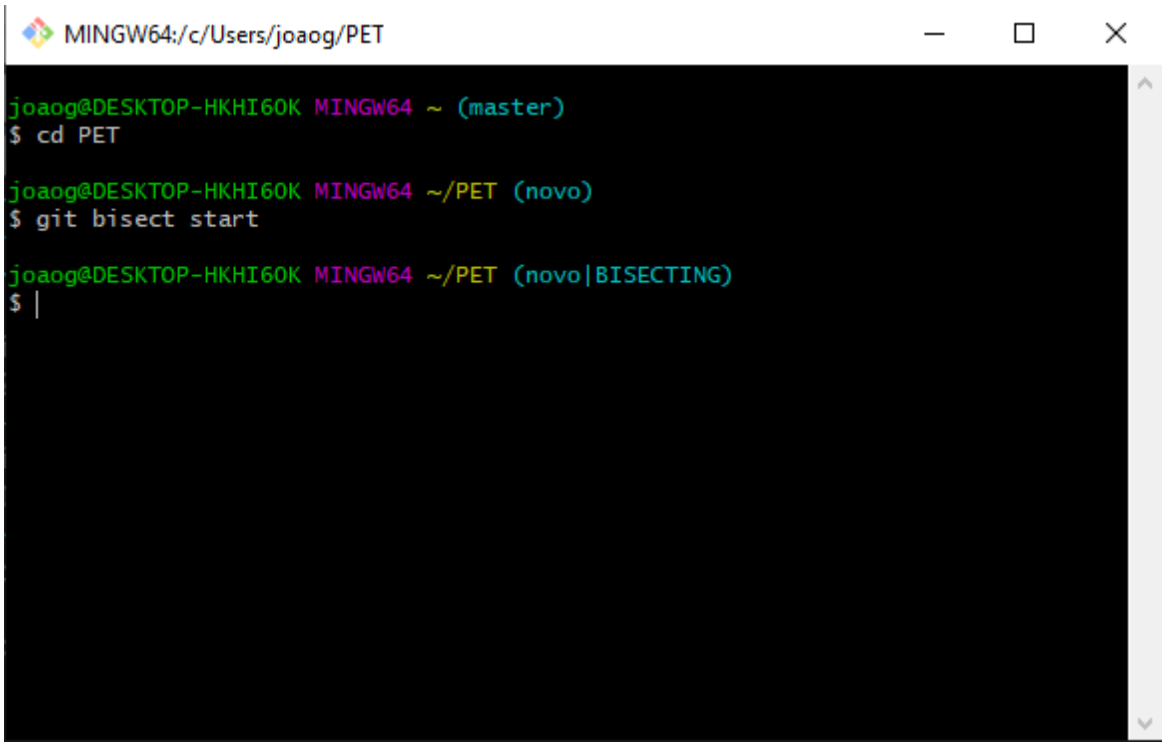

Figura 4.10: Iniciando o processo de *bisect*.

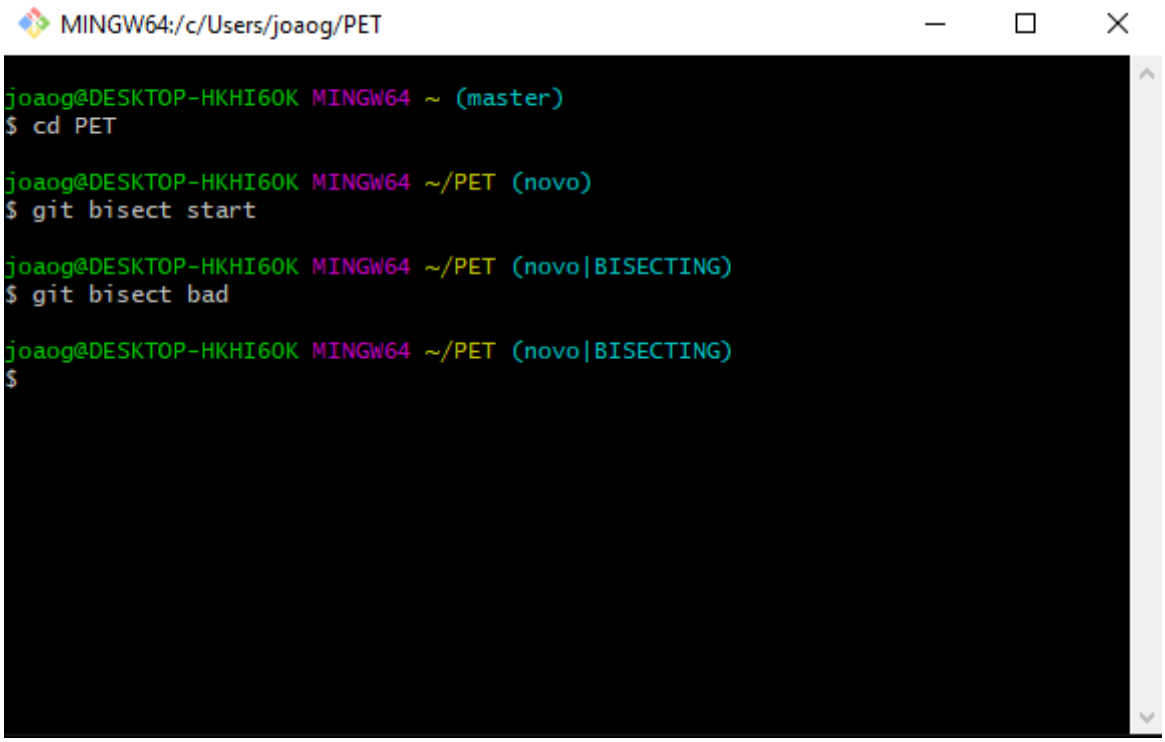

Figura 4.11: Informando o erro.

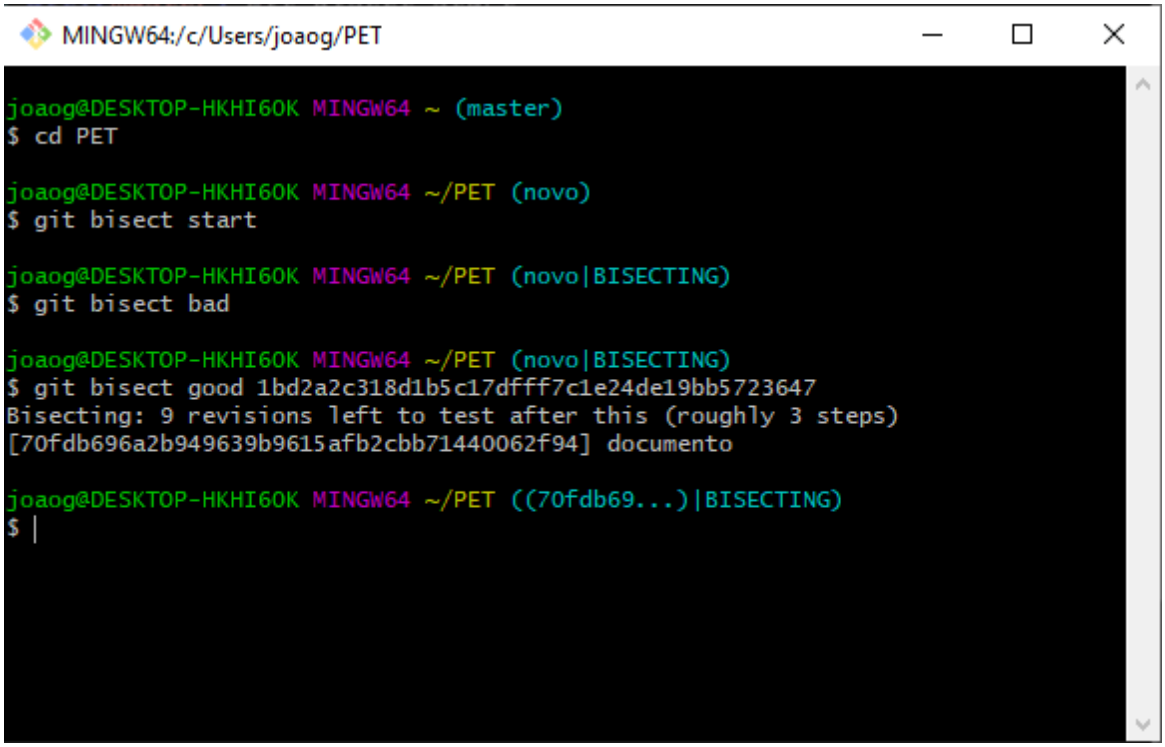

Figura 4.12: Informando o *commit* sem erro.

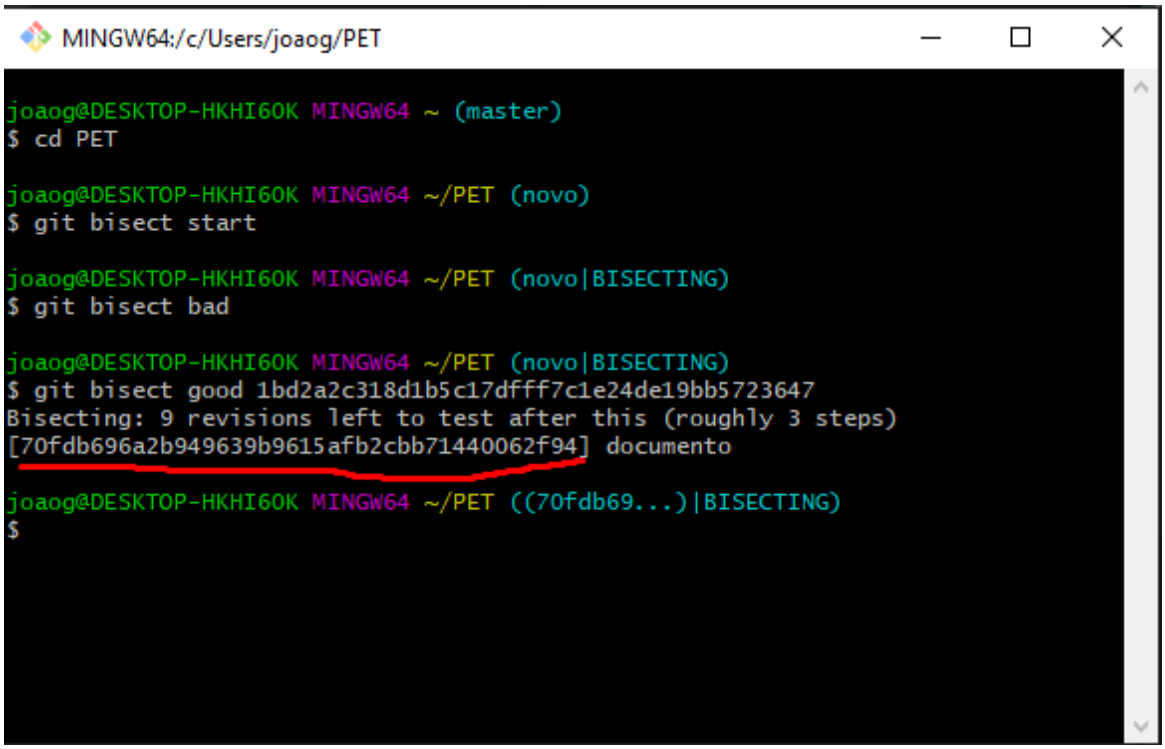

Figura 4.13: Código de commit.

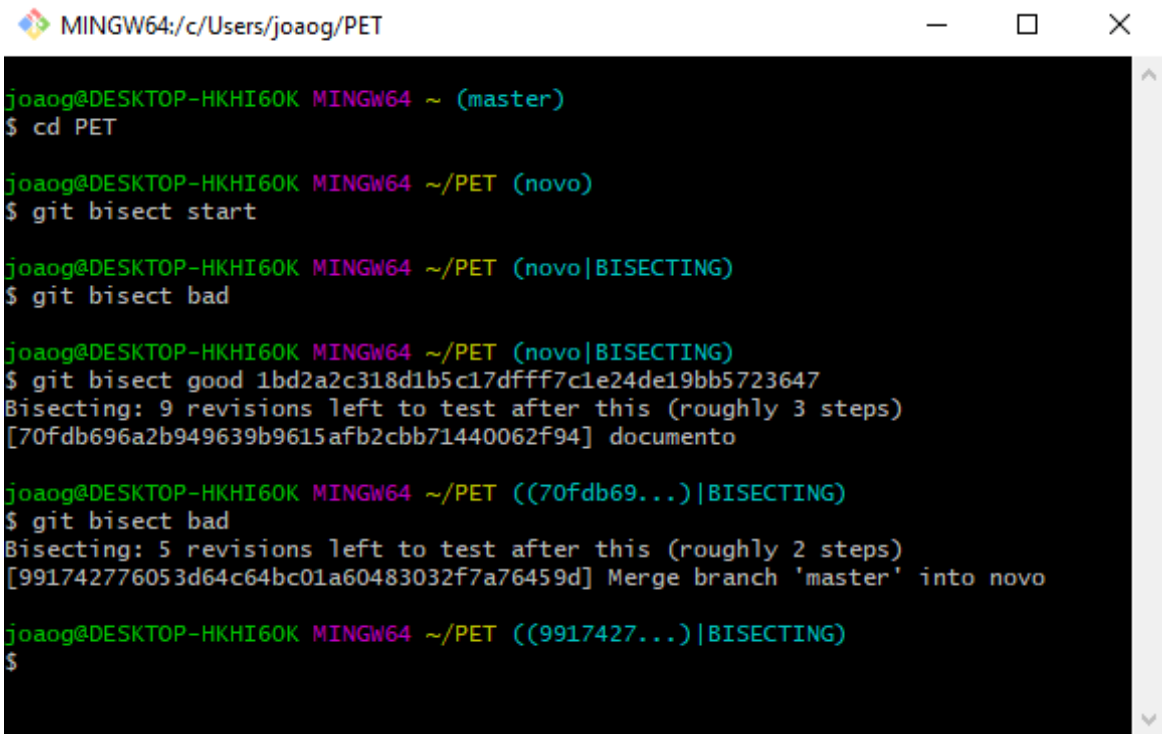

Figura 4.14: Notificação do *commit*.

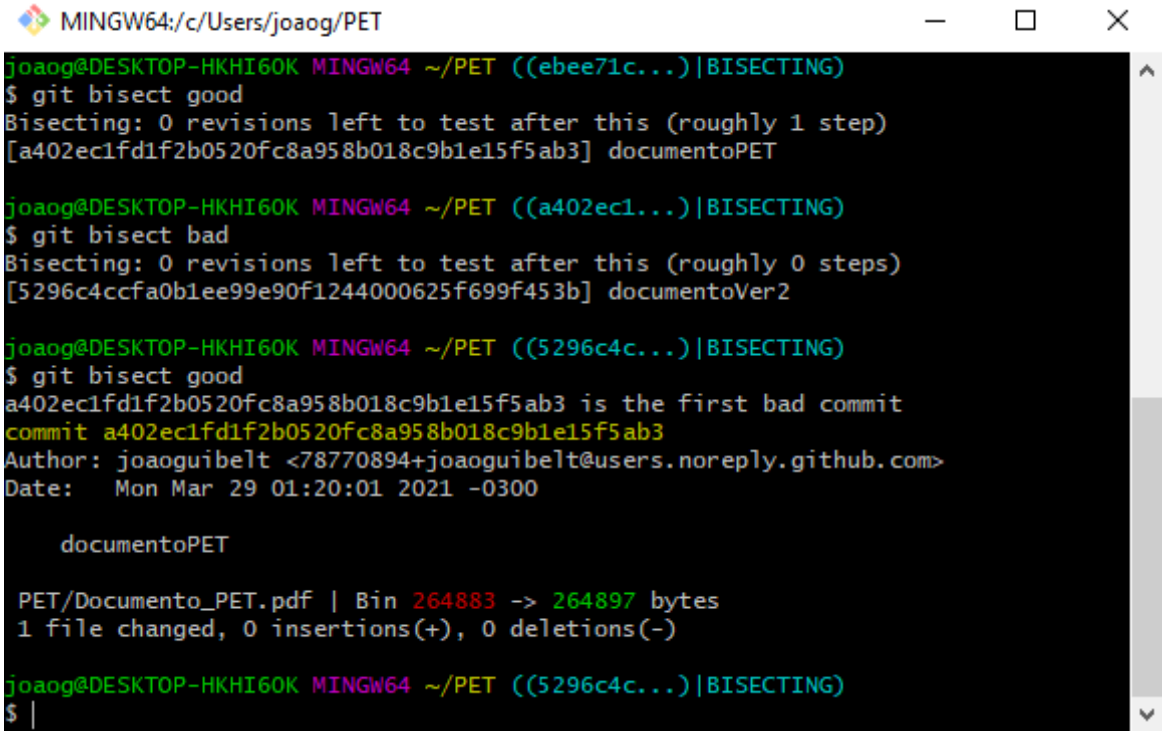

Figura 4.15: Final do processo de *bisect*.

4.9 Terminal

Conhecendo os comandos, agora é necessário saber onde usá-los. A principal forma de usar o Git é pelo terminal.

Nos sistemas operacionais Linux e MAC, recomenda-se o uso do próprio terminal do sistema. Porém, no Windows, ao baixar o Git, é instalado também o programa *git bash*, que é um aplicativo que oferece a camada de emulação para o trabalho em linha de comando.

Com o terminal já aberto, utiliza-se o comando *cd "nome da pasta"* para acessar a pasta escolhida a se tornar um repositório Git local. Isso é mostrado na Figura 4.16.

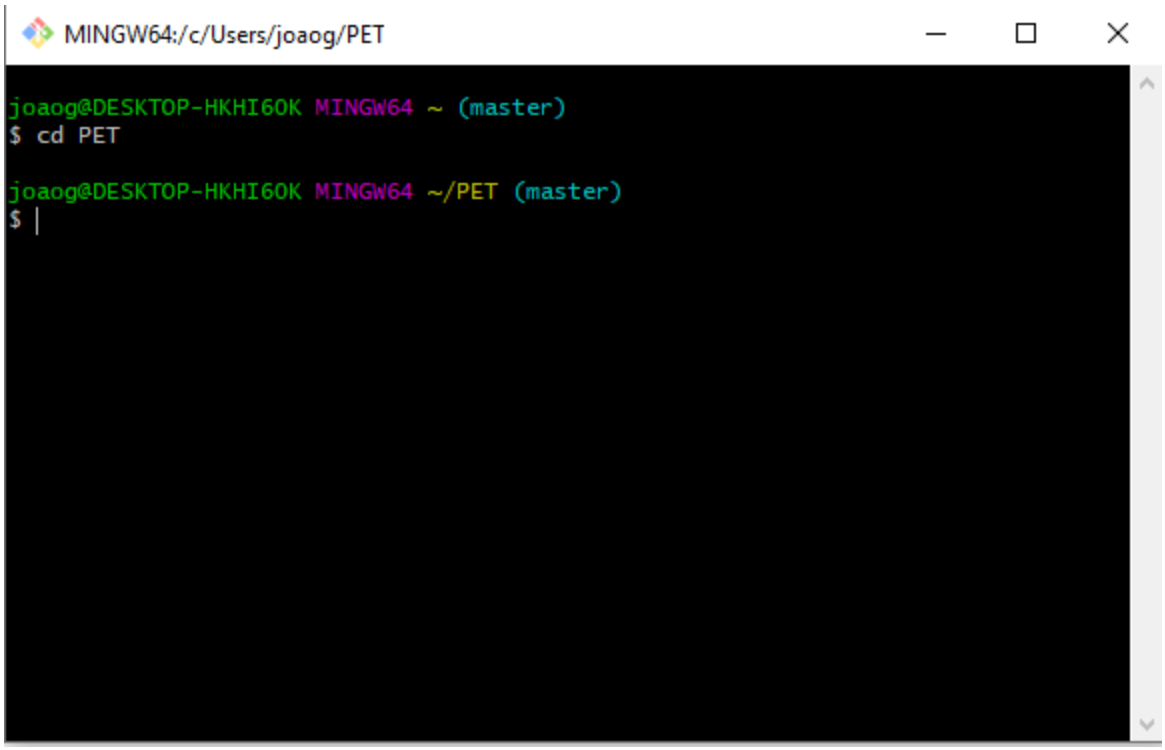

Figura 4.16: Acesso a uma pasta pelo terminal.

4.10 Interface de usuário GUI

Além do terminal, o Git tem compatibilidade com uma variedade de interfaces gráficas, como o *sourcetree* e o *GitKraken*, e até algumas interfaces específicas para certas plataformas de hospedagem, como o *GitHubDesktop*, ou a própria interface do Git, que é o *git GUI*. Nelas, o programador consegue, de forma mais simples, fazer os *"commits"* e visualizar mais facilmente o que o Git está fazendo. Porém, cada um desses *softwares* funciona de uma forma diferente, bem como apresenta suas próprias vantagens e desvantagens.

Capítulo 5

GitHub

Antes de começar a usar o Git para versionar os projetos, é preciso dominar o conceito de repositórios remotos. Como já foi dito, um SCV distribuído versiona os códigos enviando-os primeiramente para um repositório local e, em seguida, para o remoto. Nesse caso, o Git desempenha o papel do repositório local. Porém, para o remoto é preciso usar outra plataforma. As principais plataformas que funcionam como repositório remoto são o GitHub e o GitLab. Por ser a mais empregada, apenas o GitHub é discutido a seguir.

5.1 GitHub vs GitLab

O GitHub e o GitLab são plataformas de hospedagem de código-fonte. Elas permitem que os desenvolvedores contribuam em projetos privados ou abertos. Ambas fazem o controle de versão dos projetos hospedados utilizando o Git [Ber19].

A principal diferença entre elas é que o GitLab foca na integração, além de proporcionar, nativamente, ferramentas de integração e de entrega contínua. Já o GitHub foca em eficácia e desempenho de infraestrutura, e, assim, configura-se como a melhor opção para projetos com muitos programadores.

5.2 Conexão do Git com o GitHub

Uma vez que o GitHub é considerado a plataforma mais empregada, será discutida a conexão do Git com o GitHub, a fim de utilizar o versionamento de código por completo. Será assumido que a conexão entre ambos se dará por meio de uma conexão segura, do tipo SSH, como explicado a seguir.

5.2.1 Protocolo e chave SSH

O *Secure Shell* (SSH) [prob] é um protocolo que garante trocas seguras de informação por meio de chaves geradas virtualmente. Com essas chaves, é possível fazer a comunicação entre os repositórios locais e os remotos [Git].

Geração de uma nova chave

Para fazer a conexão é necessário gerar uma nova chave SSH. Para isso, deve-se abrir o terminal e digitar o comando *ssh-keygen -t ed25519 -C "e-mail da conta do github"*. Como resposta, deverá ser criada uma chave usando o endereço de *e-mail* fornecido como parâmetro,

bem como deverá aparecer uma mensagem contendo o texto *"Enter a file in which to save the key"*, que significa o pedido do nome de arquivo onde a chave será armazenada. Logo, deve-se fornecer o nome de arquivo desejado e apertar a tecla *ENTER*. Se somente for apertado a tecla *ENTER*, a chave será salva em seu endereço padrão. Para finalizar, será pedido para criar uma palavra-chave e, depois, confirmar a mesma. Se todo o processo fluir sem erros, a chave SSH deverá ter sido gerada e será possível ligá-la ao *ssh-agent*, conforme descrito a seguir.

Ligação da chave com um agente

Para que não seja necessário informar a senha toda vez que a chave for usada, é preciso ligá-la a um agente. Primeiramente, deve-se iniciar o *ssh-agent*, digitando o comando *eval "ssh-agent -s"*. Em seguida, deve-se adicionar a chave a esse agente, com o comando *ssh-add ∼/.ssh/id_ed25519*. Se a chave não foi salva no local padrão, deve-se substituir *"id_ed25519"* pelo nome do local onde encontra-se a chave. Por fim, é necessário adicionar a chave ao GitHub, o que é explicado a seguir.

Adição da chave ao GitHub

Como último passo, deve-se ligar a chave ao repositório remoto. Para isso, é preciso copiar o código da chave. Isso é feito digitando-se o comando *clip < ∼/.ssh/id_ed25519.pub* e copiando-se o código que será retornado. Como anteriormente, se a chave não foi salva no local padrão, deve-se substituir *"id_ed25519"*, pelo local onde ela está armazenada.

Em seguida, no GitHub, é necessário acessar o perfil e selecionar a opção *"settings"*. Depois, em *"SSH and GPC keys"*, selecionar *"new SSH key"*, adicionar um nome e colar o código salvo na caixa *key*. Para finalizar, deve-se selecionar *"add SSH key"* e inserir a senha do GitHub, caso seja solicitado. Se todo o processo fluir sem erro, o Git estará conectado com o repositório no GitHub.

Capítulo 6

Conclusão

Esse trabalho apresentou um resumo sobre controle de versões de documentos e plataformas de hospedagem.

O emprego do controle de versões e da hospedagem de documentos são essenciais para a organização e o controle de projetos contemporâneos, que envolvem uma grande quantidade de documentos e um trabalho em equipe.

Foram abordados o sistema de controle de versões Git e a plataforma de hospedagem de documentos GitHub.

Referências bibliográficas

[Atl] Atlassian. ***Resetting, Checking Out & Reverting.*** Disponível em: "https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting.". Acesso em: 07/01/2025.

[Ber19] Fernanda Bertola. **GitHub e GitLab: o que são e principais diferenças.** Disponível em: "https://www.zup.com.br/blog/git-github-e-gitlab", 2019. Acesso em: 07/01/2025.

[Bit] BitKeeper. ***Why use BitKeeper when there are lots of great alternatives?*** Disponível em: "http://www.bitkeeper.org/why.html". Acesso em: 07/01/2025.

[Fer19] Maurício Fernandes. **Git e GitHub: Por que e como usar? Parte 2.** Disponível em: "https://medium.com/nstech/git-e-github-por-que-e-como-usar-parte-2-d00c3b248822", 2019. Acesso em: 07/01/2025.

[Git] GitHub. **Conectar-se ao GitHub com SSH**. Disponível em: "https://docs.github.com/pt/authentication/connecting-to-github-with-ssh/about-ssh". Acesso em: 07/01/2025.

[Git20] Git. ***Website* oficial do Git.** Disponível em: "https://git-scm.com/", 2020. Acesso em: 07/01/2025.

[Gru21] Grupo PET-Tele. ***Website* do grupo.** Disponível em: "http://www.telecom.uff.br/pet", 2021. Acesso em: 07/01/2025.

[Gua20] Gustavo Guanabara. **Curso Grátis Git e GitHub.** Disponível em: "https://www.youtube.com/watch?v=xEKo29OWILE", 2020. Acesso em: 07/01/2025.

[Ini] Open Source Initiative. ***About the Open Source Initiative***. Disponível em: "https://opensource.org/about". Acesso em: 07/01/2025.

[Proa] Programa de Educação Tutorial - PET. ***Website* do programa.** Disponível em: "http://portal.mec.gov.br/index.php?option=com_conent&view=article&id=12223&ativo=481&Itemid=480". Acesso em: 07/01/2025.

[prob] SSH protocol. **SSH *Protocol* – *Secure Remote Login and File Transfer***. Disponível em: "https://www.ssh.com/academy/ssh/protocol". Acesso em: 07/01/2025.

[Sek20] Harsh Seksaria. **Terminologias básicas do Git.** Disponível em: "https://ichi.pro/pt/terminologias-basicas-do-git-147845933986850", 2020. Acesso em: 07/01/2025.