



INTRODUCTION TO GIT & GITHUB

by Bobby Ilier



Table of Contents

Sobre o livro	6
Sobre o autor	7
Patrocinadores	8
Ferramenta de geração do PDF do Ebook	9
Ferramenta de geração do ePub do Ebook	10
Capa do Livro	11
Licença	12
 Introdução ao Git	 13
 Controle de Versão	 14
 Instalando o Git	 16
 Comandos Básicos do Shell	 18
 Configuração do Git	 22
 Introdução ao GitHub	 26
Estrelas do GitHub	29
 Inicializando um projeto Git	 30
 Git Status	 32
 Git Add	 34

Git Commit	36
Assinando Commits	38
Git Diff	43
Git Log	45
Gitignore	47
Chaves SSH	56
Git Push	59
Criando e vinculando um repositório remoto	60
Enviando commits	61
Verificando o repositório remoto	62
Git Pull	63
Branches do Git	66
Git Merge	72
Revertendo alterações	78
Resetando alterações (⚠ Resetar é perigoso ⚠)	79
Git Clone	83
Fork no Git	85

Fluxo de Trabalho no Git	87
Pull Requests	89
Git e VS Code	92
Instalando o VS Code	93
Clonando um repositório no VS Code	94
Criar um branch	95
Configurar um template de mensagem de commit	96
Conclusão	97
Fontes adicionais:	98
GitHub CLI	99
Instalação do GitHub CLI	100
Autenticação	101
Comandos úteis do GitHub CLI	104
Git Stash	107
Guardando seu trabalho	108
Restaurando as alterações guardadas	110
Lidando com múltiplas cópias guardadas do seu trabalho	111
Git Alias	112
Git Rebase	113
Git Switch	119
Guia Rápido de Markdown do GitHub	121

Crie seu perfil no GitHub	129
Guia Rápido de Comandos Git	135
Conclusão	141

Sobre o livro

- **Esta versão foi publicada em 30 de outubro de 2023**

Este é um guia introdutório de código aberto sobre Git e GitHub que irá ajudá-lo a aprender o básico do controle de versão e começar a usar o Git em seus projetos de SysOps, DevOps e desenvolvimento. Não importa se você é engenheiro DevOps/SysOps, desenvolvedor ou apenas um entusiasta de Linux, você pode usar o Git para rastrear alterações no código e colaborar com outros membros da equipe ou mantenedores de projetos open source.

O guia é adequado para qualquer pessoa que trabalhe como desenvolvedor, administrador de sistemas ou engenheiro DevOps e queira aprender o básico de Git e GitHub.

Sobre o autor

Meu nome é Bobby Iliev e trabalho como Engenheiro DevOps Linux desde 2014. Sou um amante do Linux e defensor da filosofia do movimento open source. Estou sempre fazendo aquilo que não sei fazer para aprender como se faz, e acredito em compartilhar conhecimento.

Acredito que é essencial manter sempre o profissionalismo, cercar-se de boas pessoas, trabalhar duro e ser gentil com todos. Você precisa ter um desempenho consistentemente superior ao dos outros. Esse é o verdadeiro profissional.

Para mais informações, visite meu blog em <https://bobbyiliev.com>, siga-me no Twitter [@bobbyiliev_](#) e no [YouTube](#).

Patrocinadores

Este livro só foi possível graças a estas empresas fantásticas!

DigitalOcean

A DigitalOcean é uma plataforma de serviços em nuvem que oferece a simplicidade que os desenvolvedores adoram e a confiança que as empresas precisam para executar aplicações em produção em escala.

Ela fornece soluções de computação, armazenamento e rede altamente disponíveis, seguras e escaláveis, que ajudam desenvolvedores a criar ótimos softwares mais rapidamente.

Fundada em 2012, com escritórios em Nova York e Cambridge, MA, a DigitalOcean oferece preços transparentes e acessíveis, uma interface elegante e uma das maiores bibliotecas de recursos open source disponíveis.

Para mais informações, visite <https://www.digitalocean.com> ou siga [@digitalocean](#) no Twitter.

Se você é novo na DigitalOcean, pode obter um crédito gratuito de \$200 e criar seus próprios servidores através deste link de indicação:

[Crédito gratuito de \\$200 na DigitalOcean](#)

DevDojo

O DevDojo é um recurso para aprender tudo sobre desenvolvimento web e design. Aprenda no seu intervalo de almoço ou acorde e aproveite uma xícara de café conosco para aprender algo novo.

Junte-se a esta comunidade de desenvolvedores, e todos podemos aprender juntos, construir juntos e crescer juntos.

[Junte-se ao DevDojo](#)

Para mais informações, visite <https://www.devdojo.com> ou siga [@thedevedojo](#) no Twitter.

Ferramenta de geração do PDF do Ebook

Este ebook foi gerado por [Ibis](#), desenvolvido por [Mohamed Said](#).

Ibis é uma ferramenta PHP que ajuda você a escrever eBooks em markdown.

Ferramenta de geração do ePub do Ebook

A versão ePub foi gerada por [Pandoc](#).

Capa do Livro

A capa deste ebook foi criada com [Canva.com](https://www.canva.com).

Se você precisar criar um gráfico, pôster, convite, logotipo, apresentação – ou qualquer coisa que precise ficar bonita — experimente o Canva.

Licença

Licença MIT

Copyright (©) 2020 Bobby Iliev

Permissão é concedida, gratuitamente, a qualquer pessoa que obtenha uma cópia deste software e dos arquivos de documentação associados (o "Software"), para lidar com o Software sem restrições, incluindo, sem limitação, os direitos de usar, copiar, modificar, mesclar, publicar, distribuir, sublicenciar e/ou vender cópias do Software, e permitir que pessoas a quem o Software é fornecido o façam, sujeito às seguintes condições:

O aviso de copyright acima e esta permissão deverão ser incluídos em todas as cópias ou partes substanciais do Software.

O SOFTWARE É FORNECIDO "COMO ESTÁ", SEM GARANTIA DE QUALQUER TIPO, EXPRESSA OU IMPLÍCITA, INCLUINDO MAS NÃO SE LIMITANDO ÀS GARANTIAS DE COMERCIALIZAÇÃO, ADEQUAÇÃO A UM DETERMINADO FIM E NÃO VIOLAÇÃO. EM NENHUM CASO OS AUTORES OU DETENTORES DOS DIREITOS AUTORAIS SERÃO RESPONSÁVEIS POR QUALQUER REIVINDICAÇÃO, DANO OU OUTRA RESPONSABILIDADE, SEJA EM UMA AÇÃO DE CONTRATO, DELITO OU DE OUTRA FORMA, DECORRENTE DE, OU EM CONEXÃO COM O SOFTWARE OU O USO OU OUTRAS NEGOCIAÇÕES NO SOFTWARE.

Introdução ao Git

Bem-vindo a este guia de treinamento básico sobre Git e GitHub! Neste **curso rápido de Git**, você aprenderá o **básico do Git** para poder usar o Git para rastrear alterações no seu código e colaborar com outros membros da sua equipe ou mantenedores de projetos open source.

Seja você iniciante ou experiente em programação, é fundamental saber como usar o Git. A maioria dos projetos desenvolvidos por pequenos ou grandes grupos de desenvolvedores são feitos através do GitHub ou GitLab.

Trabalhar com outros desenvolvedores se torna muito mais empolgante e agradável, apenas criando um novo branch, adicionando todas as suas ideias brilhantes ao código que podem ajudar o projeto, fazendo o commit e depois enviando para o GitHub ou GitLab. Então, após abrir o PR (pull request), ele é revisado e mesclado, você pode voltar ao seu código e continuar adicionando mais coisas incríveis. Claro, depois de puxar as alterações do branch main/master.

Se o que você acabou de ler não faz sentido para você, não se preocupe. Tudo será explicado neste eBook!

Este eBook mostrará o básico de como começar a usar o Git e tentará ajudar você a se sentir mais confortável com ele.

Pode parecer um pouco assustador no começo, mas não se preocupe. Não é tão difícil quanto parece e, espero que, após ler este eBook, você se sinta mais confortável com o Git.

Aprender Git é essencial para todo programador. Até mesmo algumas das maiores empresas usam o GitHub para seus projetos. Lembre-se: quanto mais você usar, mais acostumado ficará.

[Git](#) é, sem dúvida, o sistema de controle de versão open source mais popular para rastrear alterações em código fonte.

O autor original do Git é [Linus Torvalds](#), que também é o criador do **Linux**.

O Git foi projetado para ajudar programadores a coordenar o trabalho entre si. Seus objetivos incluem velocidade, integridade dos dados e suporte a fluxos de trabalho distribuídos.

Controle de Versão

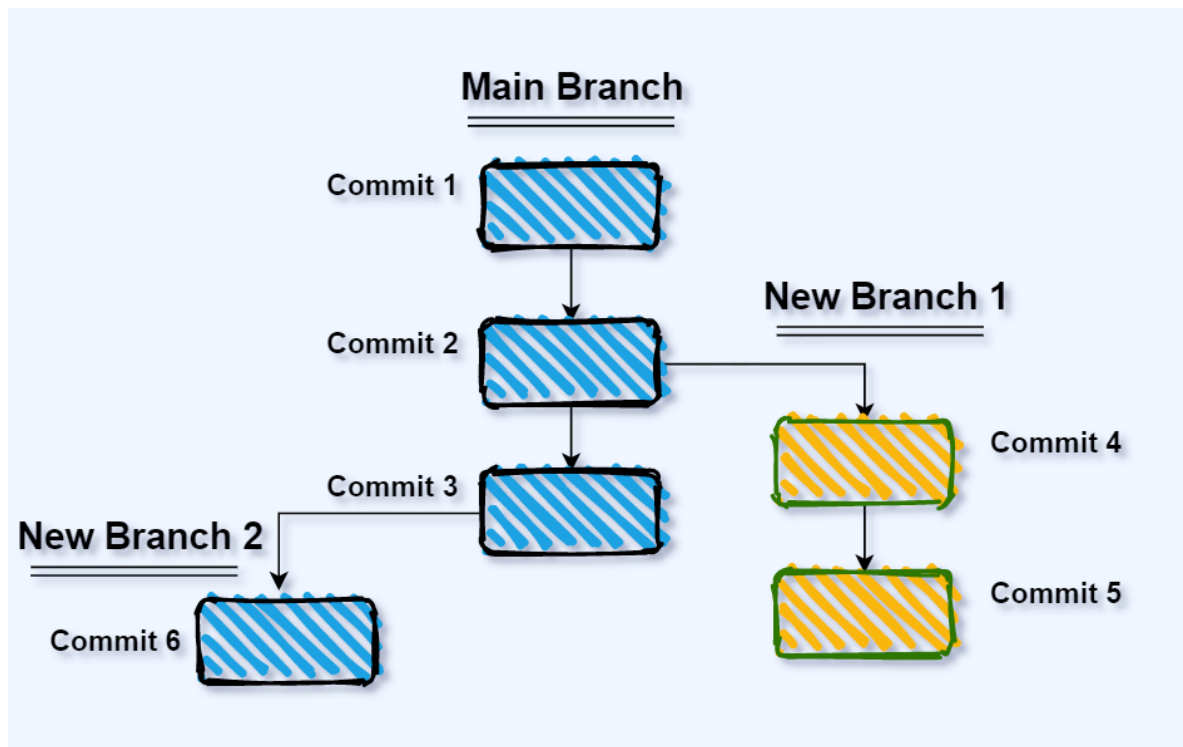
Controle de versão, também chamado de *controle de fonte*, permite que você rastreie e gerencie todas as alterações no seu código.

Por que usar controle de versão?

- Várias pessoas podem trabalhar no mesmo projeto simultaneamente.
- Serve simultaneamente como repositório, narrativa do projeto, meio de comunicação e ferramenta de gestão de equipe e produto.
- Registra todas as alterações em um log.
- Permite que membros da equipe trabalhem de forma concorrente e oferece a possibilidade de mesclar esse trabalho.
- Rastreia cada alteração feita no software.
- Os dados são transitórios e podem ser facilmente perdidos.

O que é um Sistema de Controle de Versão?

Também conhecido como gerenciador de código-fonte (SCM) ou sistema de controle de revisão (RCS), é um sistema que mantém o controle das alterações em um arquivo ou conjunto de arquivos e, em caso de problemas, permite voltar no tempo, comparar mudanças ao longo do tempo e reverter facilmente para um estado funcional do seu código-fonte. SVN, Mercurial e o extremamente popular Git são sistemas de controle de versão usados por desenvolvedores. Todos são gratuitos e open-source.



Com sistemas de controle de versão distribuídos como o Git, você terá seu código-fonte armazenado em um repositório remoto como o GitHub e também em um repositório local no seu computador.

Você aprenderá mais sobre repositórios remotos e locais nos próximos capítulos. Mas, por enquanto, um dos principais pontos é que seu código-fonte ficará armazenado em um repositório remoto, então, caso algo dê errado com seu laptop, você não perderá todas as suas alterações, pois elas estarão armazenadas com segurança no GitHub.

Instalando o Git

Para que você possa usar o Git em sua máquina local, será necessário instalá-lo.

Dependendo do sistema operacional que você está usando, siga os passos abaixo.

Instalar o Git no Linux

Na maioria das distribuições Linux, a ferramenta de linha de comando do Git já vem instalada por padrão.

Se esse não for o seu caso, você pode instalar o Git com o seguinte comando:

- No RHEL Linux:

```
sudo dnf install git-all
```

- Em distribuições baseadas no Debian, incluindo Ubuntu:

```
sudo apt install git-all
```

Instalar o Git no Mac

Se você estiver usando Mac, o Git também deve estar disponível por padrão. No entanto, se não estiver, há 2 principais formas de instalar o Git no seu Mac:

- Usando o Homebrew: caso você utilize o Homebrew, abra o terminal e execute:

```
brew install git
```

- Instalador do Git: alternativamente, você pode usar o seguinte instalador:

[git-osx-installer](#)

Pessoalmente, recomendo usar o Homebrew.

Instalar o Git no Windows

Se você tem um PC com Windows, siga os passos para instalar o Git no Windows aqui:

[Instalar Git no Windows](#)

Durante a instalação, certifique-se de escolher a opção Git Bash, pois ela fornecerá um terminal Git Bash que você usará ao acompanhar o conteúdo.

Verificar a versão do Git

Depois de instalar o Git, para verificar a versão instalada em sua máquina, utilize o seguinte comando:

```
git --version
```

Exemplo de saída:

```
git version 2.25.1
```

No meu caso, tenho o Git 2.25.1 instalado no meu laptop.

Comandos Básicos do Shell

Como ao longo deste eBook usaremos principalmente o Git via linha de comando, é importante conhecer comandos básicos do shell para que você possa se orientar no terminal.

Então, antes de começarmos, vamos revisar alguns comandos básicos do shell!

O comando `ls`

O comando `ls` permite listar o conteúdo de uma pasta/diretório. Para executar o comando, basta abrir o terminal e rodar o seguinte:

```
ls
```

A saída mostrará todos os arquivos e pastas que estão localizados no seu diretório atual. No meu caso, começando pelo diretório raiz deste projeto, a saída é a seguinte:

```
CONTRIBUTING.md LICENSE README.md ebook index.html
```

Pela saída, podemos ver que `CONTRIBUTING.md`, `LICENSE`, `README.md`, `index.html` são arquivos, enquanto `ebook` é um subdiretório/subpasta.

Para mais informações sobre o comando `ls`, confira esta página [aqui](#).

Nota: Isso funciona em sistemas Linux/UNIX. Se você estiver no Windows e usando o CMD padrão, deve usar o comando `dir`.

O comando `cd`

O comando `cd` significa `Change Directory` (Mudar Diretório) e permite navegar pelo sistema de arquivos do seu computador ou servidor. Suponha que eu queira entrar no diretório `ebook` da saída acima. Basta rodar o comando `cd` seguido do diretório que quero acessar:

```
cd ebook
```

Se eu quiser voltar um nível, uso o comando `cd ..`. Assim, volto um nível acima do diretório `ebook`, retornando ao diretório raiz do projeto.

O comando `pwd`

O comando `pwd` significa **Print Working Directory** (Imprimir Diretório Atual), ou seja, ao rodar o comando, ele mostra o diretório em que você está.

No exemplo acima, se eu rodar o comando `pwd`, terei o caminho completo da pasta em que estou:

```
pwd
```

Saída:

```
/home/bobby/introduction-to-git
```

Depois posso usar o comando `cd` para acessar o diretório `ebook`:

```
cd ebook
```

E, se rodar o comando `pwd` novamente, verei a seguinte saída:

```
/home/bobby/introduction-to-git/ebook
```

Ou seja, graças ao comando `pwd`, pude ver que estava no diretório `/home/bobby/introduction-to-git` e, após acessar o diretório `ebook`, usando novamente o `pwd`, vi que meu novo diretório atual é `/home/bobby/introduction-to-git/ebook`.

O comando **rm**

O comando **rm** significa **remove** (remover) e permite excluir arquivos e pastas. Suponha que eu queira excluir o arquivo **README.md**, basta rodar:

```
rm README.md
```

Caso precise excluir uma pasta/diretório, é necessário especificar a flag **-r**:

```
rm -r ebook
```

Nota: lembre-se que o comando **rm** exclui completamente os arquivos e pastas, e a ação é irreversível, ou seja, não tem como recuperá-los.

O comando **mkdir**

O comando **mkdir** significa **make directory** (criar diretório) e é usado para criar um ou mais novos diretórios. Para criar um novo diretório, basta abrir o terminal, navegar até o local desejado com **cd** e rodar:

```
mkdir Minha_Nova_Pasta
```

O comando acima criará um novo diretório vazio chamado **Minha_Nova_Pasta**.

Você também pode criar vários diretórios de uma vez, colocando os nomes desejados um após o outro:

```
mkdir Minha_Nova_Pasta Minha_Outra_Nova_Pasta
```

O comando **touch**

O comando **touch** é usado para atualizar o timestamp dos arquivos. Uma funcionalidade útil do comando **touch** é que ele cria um arquivo vazio. Isso é útil se você quiser criar um arquivo no diretório que ainda não existe:

```
touch README.md
```

O comando acima criará um novo arquivo vazio chamado README.md.

Uma coisa importante é que todos os comandos do shell são sensíveis a maiúsculas e minúsculas, então se você digitar **LS** não irá funcionar.

Com isso, agora você conhece alguns comandos básicos do shell que serão úteis no seu dia a dia.

Configuração do Git

A primeira vez que você configurar o Git na sua máquina, será necessário fazer algumas configurações iniciais.

Há algumas coisas principais que você precisará configurar:

- Seus dados: como nome e endereço de e-mail
- Seu editor do Git
- O nome padrão do branch: aprenderemos mais sobre branches mais adiante

Podemos alterar todas essas configurações usando o comando `git config`.

Vamos começar com a configuração inicial!

O comando `git config`

Para configurar seus dados do Git, como nome de usuário e endereço de e-mail, use o seguinte comando:

- Configurando seu nome de usuário do Git:

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

- Configurando seu endereço de e-mail do Git:

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

Normalmente, é bom ter o mesmo nome de usuário e e-mail na configuração local do Git e no seu perfil do GitHub.

- Configurando o editor padrão do Git

Em alguns casos, ao executar comandos do Git pelo terminal, um editor será aberto para você digitar, por exemplo, uma mensagem de commit. Para especificar seu editor

padrão, execute:

```
git config --global core.editor nano
```

Você pode trocar o editor **nano** por outro, como **vim** ou **emacs**, conforme sua preferência.

- Configurando o nome padrão do branch

Ao criar um novo repositório na sua máquina local, ele é inicializado com um nome de branch específico, que pode ser diferente do padrão do GitHub. Para garantir que o nome do branch local seja igual ao padrão do GitHub, use:

```
git config --global init.defaultBranch main
```

Por fim, após todas as alterações, você pode verificar sua configuração atual do Git com o comando:

```
git config --list
```

Exemplo de saída:

```
user.name=Bobby Iliev
user.email=bobby@bobbyiliev.com
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
```

O arquivo **~/.gitconfig**

Como usamos a opção **--global** nos comandos, todas essas configurações globais do Git ficam armazenadas em um arquivo **.gitconfig** dentro do seu diretório pessoal.

Podemos usar o comando **cat** para ver o conteúdo do arquivo:

```
cat ~/.gitconfig
```

Exemplo de saída:

```
[user]
  name = Bobby Iliev
  email = bobby@bobbyiliev.com
```

Você pode até editar o arquivo manualmente com seu editor favorito, mas eu prefiro usar o comando `git config` para evitar problemas de sintaxe.

Configurações específicas de repositório

Até agora, usamos a opção `--global` em todas as alterações de configuração do git, o que faz com que as mudanças sejam aplicadas a todos os repositórios. No entanto, você pode querer alterar a configuração apenas para um repositório específico. Para isso, basta executar os mesmos comandos do git config mencionados anteriormente, mas sem a opção `--global`. Assim, as alterações serão salvas apenas para o repositório em que você está, mantendo as configurações globais inalteradas.

O diretório `.git`

Sempre que você inicializa um novo projeto ou clona um do GitHub, ele terá um diretório `.git` onde todos os commits do Git serão registrados e também um arquivo `config` onde as configurações do projeto específico serão armazenadas.

Você pode usar o comando `ls` para ver o conteúdo da pasta `.git`:

```
ls .git
```

Saída:

```
COMMIT_EDITMSG  HEAD  branches  config  description  hooks
index  info  logs  objects  refs
```


Nota: Antes de executar o comando, você precisa estar dentro do diretório do seu projeto. Aprenderemos sobre isso nos próximos capítulos, quando estudarmos o comando `git init` e como clonar um repositório existente do GitHub com o comando `git clone`.

Introdução ao GitHub

Antes de mergulharmos em todos os diversos comandos do Git, vamos rapidamente nos familiarizar com o GitHub.

O Git é essencialmente a ferramenta que você usa para rastrear as alterações do seu código, e o GitHub, por outro lado, é um site onde você pode enviar seus projetos locais.

Isso é necessário pois funciona como um hub central onde você armazena seus projetos e todos os seus colegas de equipe ou outras pessoas trabalhando no mesmo projeto podem enviar suas alterações.

Cadastro no GitHub

Antes de começar, você precisa criar uma conta no GitHub. Você pode fazer isso através deste link:

- [Cadastre-se no GitHub](#)

Você será direcionado para uma página onde deverá adicionar os detalhes da sua nova conta:



Perfil do GitHub

Depois de se cadastrar, você pode acessar https://github.com/SEU_NOME_DE_USUÁRIO e verá seu perfil público, onde pode adicionar algumas informações sobre você. Aqui está um exemplo de perfil que você pode conferir: [Perfil do GitHub](#)



Criando um novo repositório

Se você não está familiarizado com a palavra repositório, pode pensar nele como um projeto. Ele irá conter todos os arquivos da sua aplicação ou site que você está

construindo. As pessoas geralmente chamam o repositório de repo.

Para criar um novo repositório no GitHub, clique no sinal de + no canto superior direito ou no botão verde **NEW** no canto superior esquerdo onde estão os repositórios e depois clique em **New Repository**:



Depois disso, você será direcionado para uma página onde pode especificar as informações do seu novo repositório, como:

- O nome do projeto: aqui, certifique-se de usar algo descritivo
- Uma descrição geral sobre o projeto e do que se trata
- Escolha se deseja que o repositório seja Público ou Privado



Depois de adicionar as informações necessárias e clicar no botão de criar, você será direcionado para uma página com instruções de como enviar seu projeto local para o GitHub:



Vamos abordar esses passos com mais detalhes nos próximos capítulos.

Repositórios Públicos vs. Privados

Dependendo do projeto e se ele é open source ou não, você pode definir seu repositório como **público** ou **privado**.

A principal diferença é que, com um repositório público, qualquer pessoa na internet pode ver esse repositório. Mesmo que possam ver e ler o código, você será o mantenedor do projeto e escolherá quem pode fazer commits.

Já um repositório privado só será acessível a você e às pessoas que você convidar.

Repositórios públicos são usados para projetos open source.

Adicionar colaboradores aos seus projetos

Colaboradores são pessoas que trabalham ativamente no projeto. Por exemplo, se uma empresa tem um projeto para o qual algumas pessoas devem trabalhar, essas pessoas são adicionadas como colaboradoras pela empresa.

Selecione um repositório do GitHub e navegue até a aba de configurações. No menu

lateral esquerdo, há a opção **Manage access**, onde você pode adicionar colaboradores ao seu projeto.

O arquivo README.md

O arquivo **README.md** é uma parte essencial de cada projeto. A extensão **.md** significa Markdown.

Você pode pensar no arquivo **README.md** como a introdução ao seu repositório. Ele é útil porque, ao olhar o repositório de alguém, você pode simplesmente rolar até o README e ver do que se trata o projeto.

É fundamental que seu projeto seja bem apresentado. Se o projeto não for bem introduzido, ninguém vai gastar tempo ajudando a melhorá-lo ou desenvolvê-lo.

Por isso, ter um bom arquivo README é necessário e não deve ser negligenciado. Você deve dedicar um tempo considerável a ele.

Neste post, vou compartilhar algumas dicas sobre como melhorar seu README, e espero que isso ajude nos seus repositórios.

Para mais informações, confira este post sobre [como escrever um bom arquivo README.md](#).

Estrelas do GitHub

Vamos começar respondendo à pergunta: por que damos estrela a um repositório?

Primeiramente, as pessoas dão estrela a um repositório para uso futuro ou apenas para acompanhar. Eu daria estrela a um repositório porque posso precisar dele depois.

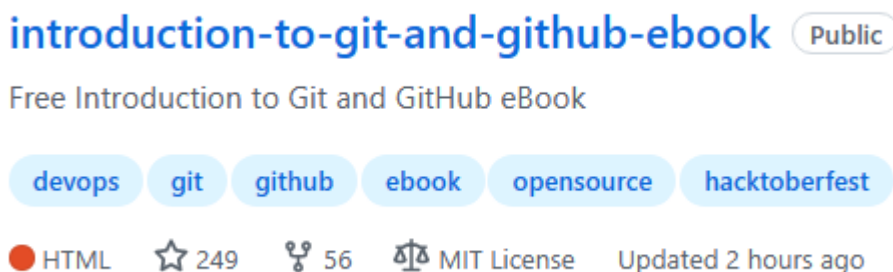
Por exemplo, o repositório [introduction-to-git-and-github-ebook](https://github.com/bobbyiliev/introduction-to-git-and-github-ebook) é essencial porque você pode ficar com dúvidas sobre Git como iniciante e pode simplesmente consultá-lo facilmente.

Em segundo lugar, é uma forma de mostrar apoio ao criador e aos mantenedores do repositório.

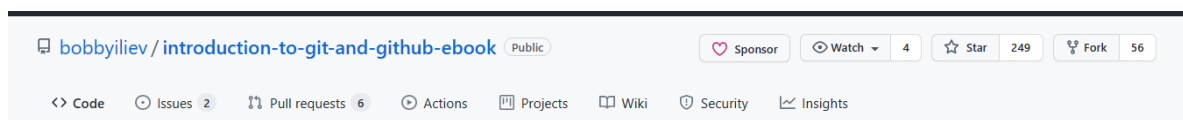
Então, vamos usar este eBook [introduction-to-git-and-github-ebook](https://github.com/bobbyiliev/introduction-to-git-and-github-ebook) como exemplo para dar estrela a um repositório.

Você deve acessar o GitHub e encontrar o repositório [introduction-to-git-and-github-ebook](https://github.com/bobbyiliev/introduction-to-git-and-github-ebook) pela busca, ou acessar diretamente em:

<https://github.com/bobbyiliev/introduction-to-git-and-github-ebook>



Agora, enquanto estiver na página do repositório no GitHub, no topo da página onde está **USERNAME/NOME_DO_REPOSITÓRIO**, você verá alguns ícones:



Clique no ícone de estrela e você terá dado estrela ao projeto com sucesso.

Sempre que gostar de um projeto e quiser apoiar o criador, não esqueça de clicar na estrela do repositório! Assim como você curte um vídeo no YouTube.

Inicializando um projeto Git

Se você está começando um novo projeto ou se tem um projeto existente que gostaria de adicionar ao Git e depois enviar para o GitHub, é necessário inicializar um novo projeto Git com o comando `git init`.

Para manter as coisas simples, vamos supor que queremos começar a construir um projeto totalmente novo. A primeira coisa que normalmente faço é criar uma nova pasta onde vou armazenar os arquivos do projeto. Para isso, posso usar o comando `mkdir` seguido do nome da pasta, o que criará um novo diretório/pasta vazio:

```
mkdir novo-projeto
```

O comando acima criará uma pasta chamada `novo-projeto`. Então, como aprendemos no capítulo 4, podemos usar o comando `cd` para acessar o diretório:

```
cd novo-projeto
```

Depois disso, usando o comando `ls`, poderemos verificar que o diretório está completamente vazio:

```
ls -lah
```

Com isso, estamos prontos para inicializar um novo projeto Git:

```
git init
```

Você verá a seguinte saída:

```
Initialized empty Git repository in /home/devdojo/novo-  
projeto/.git/
```

Como você pode ver, o que o comando `git init` faz é criar uma nova pasta `.git`, que já discutimos no capítulo 5.

Com isso, você criou com sucesso um novo projeto Git vazio! Vamos para o próximo capítulo, onde você aprenderá a usar o comando `git status` para verificar o status atual do seu repositório.

Git Status

Sempre que você fizer alterações no seu projeto Git, é importante verificar o que mudou antes de fazer um commit ou antes de enviar suas alterações para o GitHub, por exemplo.

Para checar o status atual do seu projeto, você pode usar o comando `git status`. Se você executar o comando `git status` no mesmo diretório onde inicializou seu projeto Git no capítulo anterior, verá a seguinte saída:

```
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Como este é um repositório novo, ainda não há commits nem alterações. Então, vamos criar um arquivo `README.md` com algum conteúdo genérico. Podemos executar o seguinte comando para isso:

```
echo "# Projeto Demo" >> README.md
```

Esse comando irá escrever `# Projeto Demo` e armazenar no arquivo `README.md`.

Se você rodar `git status` novamente, verá a seguinte saída:

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
      README.md
nothing added to commit but untracked files present (use "git add" to track)
```

Como você pode ver, o Git está detectando que há 1 novo arquivo que não está sendo

rastreado no momento, chamado `README.md`, que acabamos de criar. E o Git já está sugerindo que usemos o comando `git add` para começar a rastrear o arquivo. Vamos aprender mais sobre o comando `git add` no próximo capítulo!

Vamos usar bastante o comando `git status` nos próximos capítulos! Ele é especialmente útil quando você modificou vários arquivos e quer verificar o status atual e ver todos os arquivos modificados, atualizados ou excluídos.

Git Add

Por padrão, quando você cria um novo arquivo dentro do seu projeto Git, ele não está sendo rastreado pelo Git. Então, para informar ao git que ele deve começar a rastrear o arquivo, você precisa usar o comando `git add`.

A sintaxe é a seguinte:

```
git add NOME_DO_ARQUIVO
```

No nosso caso, temos apenas 1 arquivo dentro do nosso projeto chamado `README.md`, então para adicionar esse arquivo ao Git, podemos usar o seguinte comando:

```
git add README.md
```

Se você executar `git status` novamente, verá uma saída diferente:

```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   README.md
```

Aqui você verá que agora há algumas alterações preparadas e prontas para serem commitadas. Além disso, o Git informa que o `README.md` é um novo arquivo que acabou de ser preparado e não foi rastreado antes.

Caso você tenha vários arquivos, pode listá-los todos separados por espaço após o comando `git add` para prepará-los todos de uma vez, em vez de executar `git add` várias vezes para cada arquivo individual:

```
git add arquivo1.html arquivo2.html arquivo3.html
```

Com o comando acima, adicionamos os 3 arquivos executando `git add` apenas uma vez, porém em alguns casos, você pode ter muitos arquivos novos, e adicioná-los um

por um pode ser muito demorado.

Então existe uma forma de preparar absolutamente todos os arquivos do seu projeto atual, especificando um ponto após o comando `git add` assim:

```
git add .
```

Nota: Você precisa ter cuidado com isso, pois em alguns casos pode haver arquivos que você não deseja adicionar ao Git.

Com isso, estamos prontos para avançar e aprender sobre o comando `git commit`.

Git Commit

Depois de adicionar/preparar seus arquivos, o próximo passo é realmente fazer o commit das alterações. Então, se você executar `git status` novamente, verá que o Git informa que há alterações para serem commitadas:

```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   README.md
```

Neste caso, é apenas o arquivo `README.md` que será commitado. Para isso, podemos executar o seguinte comando:

```
git commit -m "Sua mensagem de commit aqui"
```

Explicação do comando:

- `git commit`: aqui, estamos dizendo ao git que queremos fazer o commit das alterações que preparamos com o comando `git add`
- `-m`: esta flag indica que vamos especificar nossa mensagem de commit diretamente após ela
- Por fim, entre aspas está nossa mensagem de commit, é importante escrever mensagens curtas e descritivas

No nosso caso, podemos definir nossa mensagem de commit como `"Commit inicial"` ou `"Adicionar README.md"`, por exemplo.

Se você não especificar a flag `-m`, o Git abrirá o editor de texto padrão que configuramos no capítulo 5, onde você poderá digitar a mensagem de commit diretamente.


Commitando diretamente sem preparar arquivos:

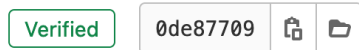
Se você ainda não preparou suas alterações usando o comando `git add`, pode ainda assim fazer o commit de todas as suas alterações diretamente usando o seguinte comando:

```
git commit -a -m "Sua mensagem de commit aqui"
```

A flag **-a** irá automaticamente preparar todas as alterações e fazer o commit delas.

Assinando Commits

O Git permite que você assine seus commits. Commits assinados com uma assinatura verificada no GitHub e GitLab exibem um selo de verificado como mostrado abaixo. 



Para assinar commits, primeiro você precisa:

- Certificar-se de que tem o GNU GPG instalado em sua máquina.
- Gerar um par de chaves GPG para assinatura, se ainda não tiver:

```
gpg --full-generate-key
```

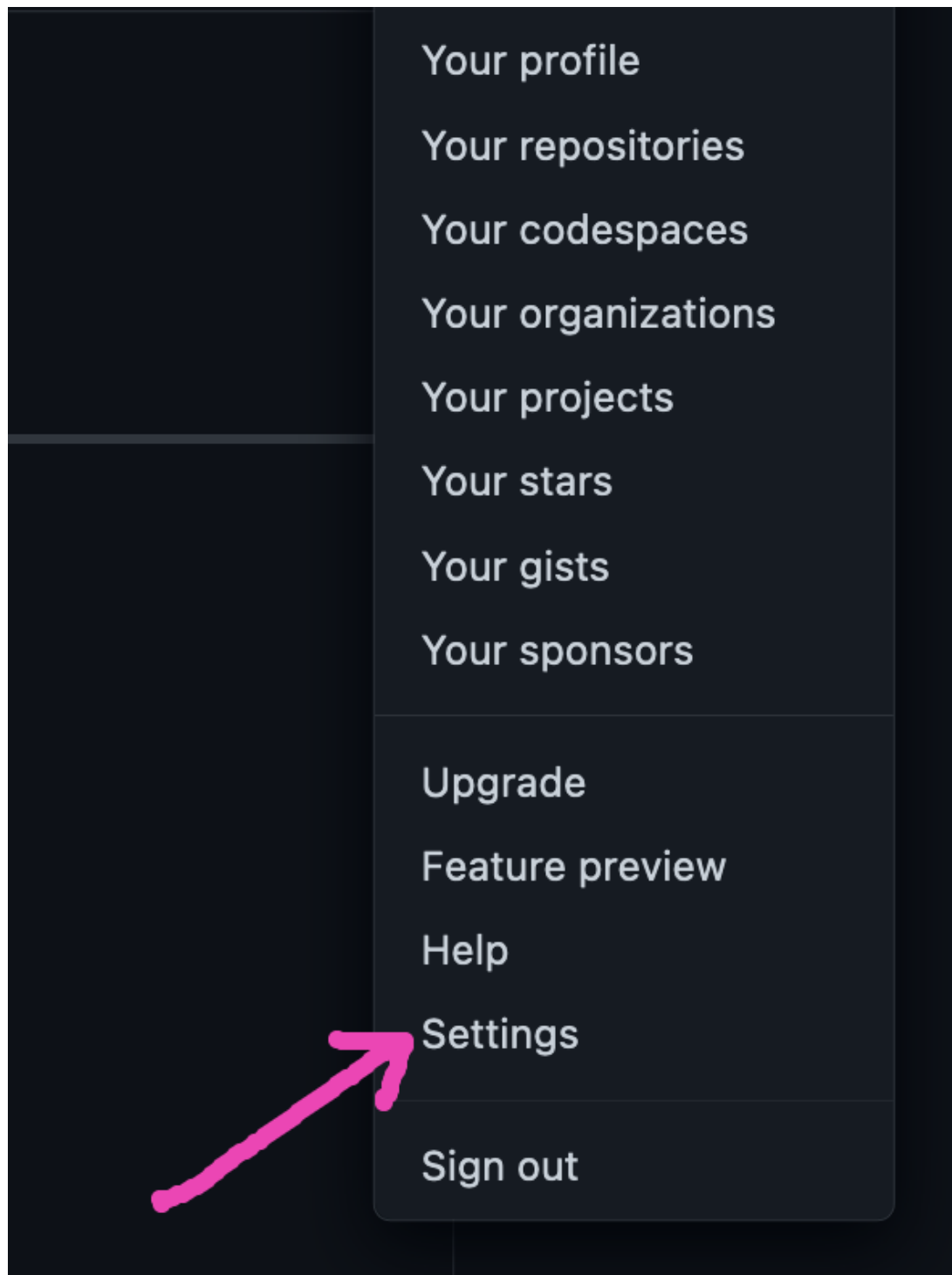
- Use o comando `gpg --list-secret-keys --keyid-format=long` para listar a forma longa das chaves GPG

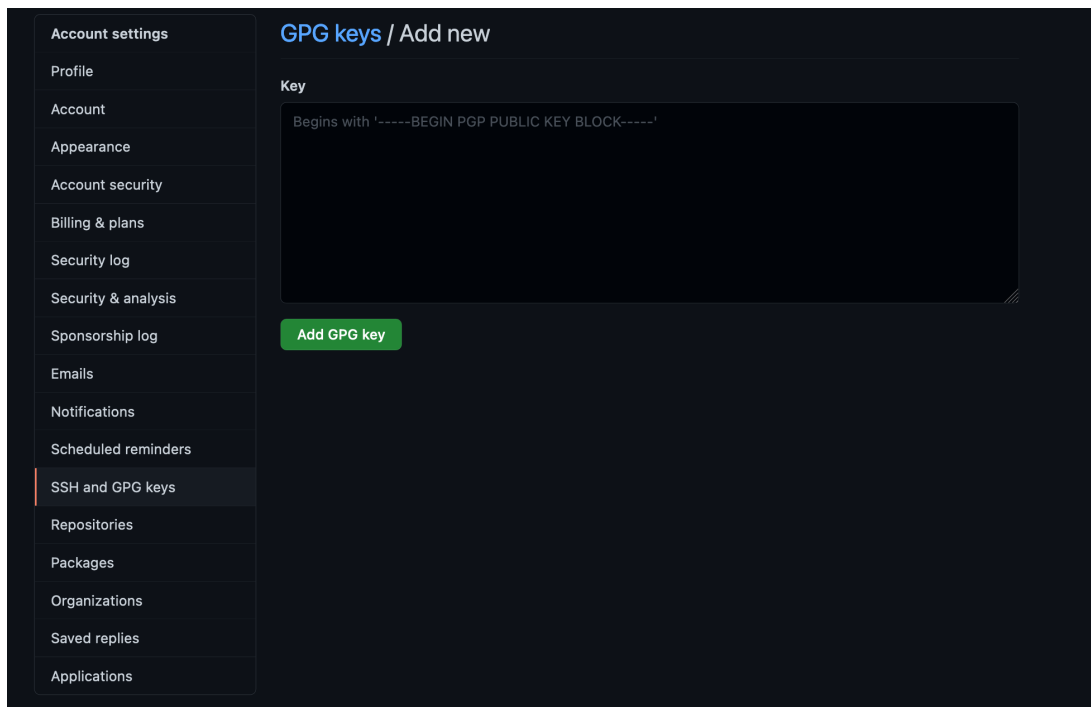
```
gpg --list-secret-keys --keyid-format=long
/Users/bobby/.gnupg/pubring.kbx
-----
sec  rsa4096/E630A0A00CAA7AAA 2021-10-01 [SC] [expires:
2026-10-01]
      5F1F417F8A043C888888888888E630F6D35CFA7ECD
uid                               [ultimate] Bobby Illiev (For signing git
commits) <bobby@bobbyiliev.com>
ssb  rsa4096/46EE4AA180001AA6 2021-10-01 [E] [expires:
2026-10-01]
```

- Copie o ID longo da chave GPG que deseja usar. Neste exemplo, o ID da chave GPG é **E630A0A00CAA7AAA**.
- Exporte a chave pública:

```
gpg --armor --export E630A0A00CAA7AAA
```

- Copie sua chave GPG, começando com `-----BEGIN PGP PUBLIC KEY BLOCK-----` e terminando com `-----END PGP PUBLIC KEY BLOCK-----`.
- Faça login no GitHub ou GitLab e adicione uma nova chave GPG nas configurações:





- Defina sua chave de assinatura GPG no Git: (Se quiser adicionar a chave por repositório, omita a flag `--global`)

```
git config --global user.signingkey E630A0A00CAA7AAA
```

- Habilite a assinatura automática para todos os commits:

```
git config --global commit.gpgsign true
```

- Ou assine por commit usando a opção `-S` no `git commit`:

```
git commit -S -m "sua mensagem de commit"
```

Após executar o comando `git commit`, podemos usar o comando `git status` novamente para verificar o status atual:

```
git status
```

Saída:


```
On branch main
nothing to commit, working tree clean
```

Como você pode ver, o Git está dizendo que não há alterações para serem commitadas, pois já fizemos o commit delas.

Vamos fazer outra alteração no arquivo `README.md`. Você pode abrir o arquivo com seu editor favorito e fazer a alteração diretamente, ou pode executar o seguinte comando:

```
echo "Git é incrível!" >> README.md
```

O comando acima adicionará uma nova linha ao final do arquivo `README.md`. Se executarmos `git status` novamente, veremos a seguinte saída:

```
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
  directory)
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -
a")
```

Como você pode ver, o Git detectou que o arquivo `README.md` foi modificado e também está sugerindo que usemos o comando que aprendemos para primeiro preparar/adicionar o arquivo!

Caso você queira alterar sua última mensagem de commit, pode executar o comando `git commit --amend`. Isso abrirá o editor padrão onde você pode alterar sua mensagem de commit. Também permite alterar as alterações do commit.

O comando `git status` nos dá uma ótima visão geral dos arquivos que foram alterados, mas não mostra quais alterações foram feitas. No próximo capítulo, vamos aprender como verificar as diferenças entre o último commit e as alterações atuais.

Para verificar os commits que alteraram um determinado arquivo, você pode usar a flag `--follow`:

```
git log --follow [arquivo]
```

O comando acima mostra os commits que alteraram o arquivo, mesmo em renomeações.

Git Diff

Como mencionado no capítulo anterior, o comando `git status` nos dá uma ótima visão geral dos arquivos que foram alterados, mas não mostra quais alterações foram feitas.

Você pode verificar as alterações reais feitas com o comando `git diff`. Se executarmos o comando em nosso repositório, veremos a seguinte saída:

```
diff --git a/README.md b/README.md
index 9366068..2b14655 100644
--- a/README.md
+++ b/README.md
@@ -1,2 @@
 # Projeto Demo
+Git é incrível
```

Como só alteramos o arquivo `README.md`, o Git está mostrando o seguinte:

- `diff --git a/README.md b/README.md`: aqui o git indica que está mostrando as alterações feitas no arquivo `README.md` desde o último commit em comparação com a versão atual do arquivo.
- `@@ -1,2 @@`: aqui o git indica que 1 nova linha foi adicionada
- `+Git é incrível`: aqui, o importante é o `+`, que indica que esta é uma nova linha que foi adicionada. Caso você remova uma linha, verá um sinal de `-` em vez disso.

No nosso caso, como só adicionamos 1 nova linha ao arquivo, o Git indica que apenas 1 arquivo foi alterado e que apenas 1 nova linha foi adicionada.

Em seguida, vamos preparar essa alteração e fazer o commit com os comandos que aprendemos nos capítulos anteriores!

- Prepare o arquivo alterado:

```
git add README.md
```

- Execute novamente o `git status` para verificar o status atual:

```
git status
```

A saída será parecida com esta, indicando que há 1 arquivo modificado:

```
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   README.md
```

- Faça o commit das alterações:

```
git commit -m "Atualizar README.md"
```

Por fim, se você executar `git status` novamente verá que não há alterações para serem commitadas.

Eu sempre executo `git status` e `git diff` antes de fazer qualquer commit, só para ter certeza do que foi alterado.

Nota 1: `git diff --staged` mostrará apenas as alterações dos arquivos na área de "staged".

Nota 2: `git diff HEAD` mostrará todas as alterações em arquivos rastreados (arquivos do último snapshot). Se todas as alterações estiverem preparadas para commit, ambos os comandos darão a mesma saída.

Em alguns casos, você pode querer ver uma lista dos commits anteriores. Vamos aprender como fazer isso no próximo capítulo.

Git Log

Para listar todos os commits anteriores, você pode usar o seguinte comando:

```
git log
```

Isso fornecerá o histórico de commits, e a saída será parecida com esta:

```
commit da46ce39a3fd663ff802d013f834431d4b4159a5 (HEAD -> main)
Author: Bobby Iliev <bobby@bobbyiliev.com>
Date:   Fri Mar 12 17:14:02 2021 +0000

    Atualizar README.md

commit fa583473b4be2807b45f35b755aa84ac78922259
Author: Bobby Iliev <bobby@bobbyiliev.com>
Date:   Fri Mar 12 17:01:17 2021 +0000

    Commit inicial
```

As entradas são listadas, em ordem, da mais recente para a mais antiga.

Resumo da saída:

- **commit da46ce39a3fd663ff802d013f834431d4b4159a5**: Aqui você pode ver o ID específico do commit
- **Author: Bobby Iliev...**: Depois você vê quem criou as alterações
- **Date: Fri Mar 12...**: Em seguida, você tem a data e hora exata em que o commit foi criado
- Por fim, você tem a mensagem do commit. Por isso é importante escrever mensagens curtas e descritivas, para que depois você possa saber quais alterações foram introduzidas por cada commit.

Se quiser verificar as diferenças entre o estado atual do seu repositório e um commit específico, basta usar o comando **git diff** seguido do ID do commit:

```
git diff fa583473b4be2807b45f35b755aa84ac78922259
```

No meu caso, a saída será a seguinte:

```
diff --git a/README.md b/README.md
index 9366068..2b14655 100644
--- a/README.md
+++ b/README.md
@@ -1,2 @@
- # Projeto Demo
+Git é incrível
```

Portanto, a diferença entre esse commit específico e o estado atual do repositório é a alteração no arquivo **README.md**.

Caso queira ver apenas os IDs dos commits e as mensagens em uma linha, você pode adicionar o argumento **--oneline**:

```
git log --oneline
```

Saída:

```
* da46ce3 (HEAD -> main) Atualizar README.md
* fa58347 Commit inicial
```

Com isso, agora você sabe como verificar o histórico de commits! No próximo capítulo, vamos aprender como excluir arquivos específicos do Git!

Gitignore

Ao trabalhar em um repositório Git, muitas vezes você terá arquivos e diretórios que não deseja commitar, para que não fiquem disponíveis para outras pessoas que usam o repositório.

Em alguns casos, você pode não querer commitar alguns arquivos no Git por motivos de segurança.

Por exemplo, se você tem um arquivo de configuração que armazena todas as credenciais do banco de dados e outras informações sensíveis, nunca deve adicioná-lo ao Git e enviá-lo para o GitHub, pois outras pessoas poderão acessar essas informações confidenciais.

Outro caso em que você pode não querer commitar um arquivo ou diretório é quando esses arquivos são gerados automaticamente e não contêm código-fonte, para não poluir seu repositório. Também faz sentido não commitar arquivos que contêm informações de ambiente, para que outras pessoas possam usar seu código com seus próprios arquivos de ambiente.

Para evitar que esses tipos de arquivos sejam commitados, você pode criar um arquivo **gitignore** que inclui uma lista de todos os arquivos e diretórios que devem ser excluídos do seu repositório Git. Neste capítulo, você aprenderá como fazer isso!

Ignorando um arquivo específico

Veja o exemplo a seguir: se você tem um projeto em **PHP** e um arquivo chamado **config.php**, que armazena os detalhes da conexão com o banco de dados, como usuário, senha, host, etc.

Para excluir esse arquivo do seu projeto git, você pode criar um arquivo chamado **.gitignore** dentro do diretório do seu projeto:

```
touch .gitignore
```

Dentro desse arquivo, basta adicionar o nome do arquivo que deseja ignorar, então o conteúdo do **.gitignore** ficaria assim:

```
config.php
```

Assim, da próxima vez que você rodar `git add .` e depois `git commit` e `git push`, o arquivo `config.php` será ignorado e não será adicionado nem enviado para o seu repositório no Github.

Dessa forma, você mantém suas credenciais do banco de dados seguras!

Ignorando um diretório inteiro

Em alguns casos, você pode querer ignorar uma pasta inteira. Por exemplo, se você tem uma pasta enorme chamada `node_modules`, não há necessidade de adicioná-la e commitá-la ao seu projeto Git, pois esse diretório é gerado automaticamente sempre que você executa `npm install`.

O mesmo vale para a pasta `vendor` no Laravel. Você não deve adicionar a pasta `vendor` ao seu projeto Git, pois todo o conteúdo dessa pasta é gerado automaticamente quando você executa `composer install`.

Para ignorar as pastas `vendor` e `node_modules`, basta adicioná-las ao seu arquivo `.gitignore`:

```
# Pastas ignoradas
/vendor/
node_modules/
```

Ignorando um diretório inteiro, exceto um arquivo específico

Às vezes, você quer ignorar um diretório, exceto um ou alguns arquivos dentro dele. Pode ser que o diretório seja necessário para sua aplicação rodar, mas os arquivos criados não devem ser enviados ao repositório remoto, ou talvez você queira ter um arquivo `README.md` dentro do diretório por algum motivo. Para isso, seu arquivo `.gitignore` deve ficar assim:


```
data/*
!data/README.md
```

A primeira linha indica que você quer ignorar o diretório **data** e todos os arquivos dentro dele. Porém, a segunda linha instrui que o **README.md** é uma exceção.

Observe que a ordem é importante nesse caso. Caso contrário, não funcionará.

Obtendo um arquivo gitignore para Laravel

Para obter um arquivo **gitignore** para Laravel, você pode pegar o arquivo do [repositório oficial do Laravel no Github](#).

O arquivo seria parecido com isso:

```
/node_modules
/public/hot
/public/storage
/storage/*.key
/vendor
.env
.env.backup
.phpunit.result.cache
Homestead.json
Homestead.yaml
npm-debug.log
yarn-error.log
```

Ele inclui essencialmente todos os arquivos e pastas que não são necessários para rodar a aplicação.

Usando o gitignore.io

Como o número de frameworks e aplicações cresce a cada dia, pode ser difícil manter seus arquivos **.gitignore** atualizados ou pode ser complicado se você precisar procurar o arquivo **.gitignore** correto para cada framework específico que usar.

Recentemente descobri um projeto open-source chamado [gitignore.io](#). É um site e uma ferramenta de linha de comando com uma enorme lista de arquivos **gitignore** pré-

definidos para diferentes frameworks.

Tudo o que você precisa fazer é visitar o site e buscar pelo framework específico que está usando.

Por exemplo, vamos buscar um arquivo **.gitignore** para Node.js:



Depois é só clicar no **Create button** e você terá instantaneamente um arquivo **.gitignore** bem documentado para seu projeto Node.js, que ficará assim:

```
# Criado por
https://www.toptal.com/developers/gitignore/api/node
# Editar em
https://www.toptal.com/developers/gitignore?templates=node

### Node ###
# Logs
logs
*.log
npm-debug.log*
yarn-debug.log*
yarn-error.log*
lerna-debug.log*

# Relatórios de diagnóstico
(https://nodejs.org/api/report.html)
report.[0-9]*.[0-9]*.[0-9]*.[0-9]*.json

# Dados de runtime
pids
*.pid
*.seed
*.pid.lock

# Diretório para libs instrumentadas geradas por
jscoverage/JSCover
lib-cov

# Diretório de cobertura usado por ferramentas como istanbul
coverage
*.lcov

# Cobertura de teste nyc
```

```
.nyc_output

# Armazenamento intermediário do Grunt
(https://gruntjs.com/creating-plugins#storing-task-files)
.grunt

# Diretório de dependências do Bower (https://bower.io/)
bower_components

# Configuração node-waf
.lock-wscript

# Addons binários compilados
(https://nodejs.org/api/addons.html)
build/Release

# Diretórios de dependências
node_modules/
jspm_packages/

# Arquivos de declaração TypeScript v1
typings/

# Cache do TypeScript
*.tsbuildinfo

# Diretório opcional de cache npm
.npm

# Cache opcional do eslint
.eslintcache

# Cache Microbundle
.rpt2_cache/
.rts2_cache_cjs/
.rts2_cache_es/
.rts2_cache_umd/

# Histórico opcional do REPL
.node_repl_history

# Saída do 'npm pack'
*.tgz

# Arquivo de integridade do Yarn
.yarn-integrity
```

```
# Arquivo de variáveis de ambiente dotenv
.env
.env.test

# Cache parcel-bundler (https://parceljs.org/)
.cache

# Saída de build do Next.js
.next

# Saída de build / generate do Nuxt.js
.nuxt
dist

# Arquivos do Gatsby
.cache/
# Comente a linha public se seu projeto usa Gatsby e não
Next.js
# https://nextjs.org/blog/next-9-1#public-directory-support
# public

# Saída de build do vuepress
.vuepress/dist

# Diretórios Serverless
.serverless/

# Cache FuseBox
.fusebox/

# Arquivos DynamoDB Local
.dynamodb/

# Arquivo de porta TernJS
.tern-port

# Armazena versões do VSCode usadas para testar extensões
VSCode
.vscode-test

# Fim de https://www.toptal.com/developers/gitignore/api/node
```

Usando o gitignore.io CLI

Se você gosta de linha de comando, o projeto gitignore.io oferece uma versão CLI também.

Para instalar no Linux, basta rodar o seguinte comando:

```
git config --global alias.ignore \  
'!gi() { curl -sL \  
https://www.toptal.com/developers/gitignore/api/$@ ;}; gi'
```

Se estiver usando outro sistema operacional, recomendo conferir a documentação [aqui](#) para saber como instalar para seu Shell ou OS específico.

Depois de instalar o comando **gi**, você pode listar todos os arquivos **.gitignore** disponíveis do gitignore.io rodando:

```
gi list
```

Por exemplo, se você precisar rapidamente de um arquivo **.gitignore** para Laravel, basta rodar:

```
gi laravel
```

E você receberá um arquivo **.gitignore** bem documentado para Laravel:

```
# Criado por
https://www.toptal.com/developers/gitignore/api/laravel
# Editar em
https://www.toptal.com/developers/gitignore?templates=laravel

### Laravel ###
/vendor/
node_modules/
npm-debug.log
yarn-error.log

# Específico do Laravel 4
bootstrap/compiled.php
app/storage/

# Específico do Laravel 5 & Lumen
public/storage
public/hot

# Específico do Laravel 5 & Lumen com caminho público alterado
public_html/storage
public_html/hot

storage/*.key
.env
Homestead.yaml
Homestead.json
/.vagrant
.phpunit.result.cache

# Laravel IDE helper
*.meta.*
_ide_*

# Fim de
https://www.toptal.com/developers/gitignore/api/laravel
```

Conclusão

Ter um arquivo **gitignore** é essencial, e é ótimo poder usar uma ferramenta como o gitignore.io para gerar seu arquivo **gitignore** automaticamente, dependendo do seu projeto!

Se você gostou do projeto gitignore.io, não deixe de conferir e contribuir com o projeto [aqui](#).

Chaves SSH

Existem algumas maneiras de autenticar no GitHub. Essencialmente, você precisa disso para poder enviar suas alterações locais do seu laptop para o seu repositório no GitHub.

Você pode usar um dos seguintes métodos:

- HTTPS: Essencialmente, isso exigirá seu nome de usuário e senha do GitHub toda vez que tentar enviar suas alterações
- SSH: Com SSH, você pode gerar um par de chaves SSH e adicionar sua chave pública ao GitHub. Assim, você não será solicitado a informar seu nome de usuário e senha toda vez que enviar suas alterações para o GitHub.



Uma coisa que você precisa ter em mente é que a URL do repositório GitHub é diferente dependendo se você está usando SSH ou HTTPS:

- HTTPS: <https://github.com/bobbyiliev/demo-repo.git>
- SSH: <git@github.com:bobbyiliev/demo-repo.git>

Note que ao escolher SSH, o [https://](https://github.com/bobbyiliev/demo-repo.git) é substituído por [git@](git@github.com:bobbyiliev/demo-repo.git), e você tem [:](https://github.com/bobbyiliev/demo-repo.git) após [github.com](https://github.com/bobbyiliev/demo-repo.git) em vez de [/](https://github.com/bobbyiliev/demo-repo.git). Isso é importante pois define como você irá autenticar cada vez.

Gerando chaves SSH

Para gerar um novo par de chaves SSH caso ainda não tenha uma, você pode rodar o seguinte comando:

```
ssh-keygen
```

Por questões de segurança, você pode especificar uma senha, que será a senha da sua chave SSH privada.

O comando acima irá gerar 2 arquivos:

- 1 chave SSH **privada** e 1 chave SSH pública. A chave privada deve sempre ser armazenada com segurança no seu laptop e você não deve compartilhá-la com ninguém.
- 1 chave SSH **pública**, que você precisa enviar para o GitHub.

Os dois arquivos serão gerados automaticamente na seguinte pasta:

```
~/ .ssh
```

Você pode usar o comando **cd** para acessar a pasta:

```
cd ~/ .ssh
```

Depois, com o comando **ls**, pode verificar o conteúdo:

```
ls
```

Saída:

```
id_rsa  id_rsa.pub
```

O **id_rsa** é sua chave privada, e novamente, você não deve compartilhá-la com ninguém.

O **id_rsa.pub** é a chave pública que precisa ser enviada para o GitHub.

Adicionando a chave SSH pública ao GitHub

Depois de criar suas chaves SSH, você precisa enviar a **chave pública** para sua conta do GitHub. Para isso, primeiro você precisa obter o conteúdo do arquivo.

Para obter o conteúdo do arquivo, use o comando **cat**:

```
cat ~/ .ssh/id_rsa.pub
```

A saída será parecida com isto:

```
ssh-rsa AAB3NzaC1yc2EAAAADAQAB..... seu_usuario@seu_host
```

Copie tudo e então acesse o [GitHub](#) e siga estes passos:

- Clique na sua foto de perfil no canto superior direito
- Depois clique em configurações



- À esquerda, clique em **SSH and GPG Keys**:



- Depois disso, clique no botão **New SSH Key**
- Especifique um título para a chave SSH, deve ser algo descritivo, por exemplo: **Chave SSH do Laptop de Trabalho**. E na área **Key**, cole sua chave SSH pública:



- Por fim, clique no botão **Add SSH Key** na parte inferior da página

Conclusão

Com isso, agora você tem suas chaves SSH geradas e adicionadas ao GitHub. Assim, você poderá enviar suas alterações sem precisar digitar seu usuário e senha do GitHub toda vez.

Para mais informações sobre chaves SSH, confira este tutorial [aqui](#).

Git Push

Por fim, depois de fazer todas as suas alterações, prepará-las com o comando `git add .` e realizar o commit com o comando `git commit`, você deve enviar (push) as alterações commitadas do seu repositório local para o repositório remoto no GitHub. Isso garante que o repositório remoto fique atualizado com o seu repositório local.

Criando e vinculando um repositório remoto

Antes de poder enviar para seu repositório remoto no GitHub, você precisa primeiro criar o repositório remoto pelo GitHub, conforme explicado no Capítulo 6.

Depois de ter seu repositório remoto pronto no GitHub, você pode adicioná-lo ao seu projeto local com o seguinte comando:

```
git remote add origin  
git@github.com:seu_usuario/seu_nome_repositorio.git
```

Nota: Certifique-se de alterar os detalhes **seu_usuario** e **seu_nome_repositorio** conforme necessário.

Assim você vincula seu projeto Git local ao repositório remoto no GitHub.

Se você leu o capítulo anterior, provavelmente percebeu que estamos usando SSH como método de autenticação.

No entanto, se você não seguiu os passos do capítulo anterior, pode usar HTTPS em vez de SSH:

```
git remote add origin  
https://github.com/seu_usuario/seu_nome_repositorio.git
```

Para verificar seu repositório remoto, execute o comando:

```
git remote -v
```

Enviando commits

Para enviar suas alterações commitadas para o repositório remoto vinculado, use o comando `git push`:

```
git push -u origin main
```

Nota: Neste comando, `-u origin main` diz ao Git para definir o branch main do repositório remoto como o branch upstream no comando `git push`. Isso é uma boa prática ao usar Git, pois permite que os comandos `git push` e `git pull` funcionem como esperado. Alternativamente, você pode usar `--set-upstream origin main` para isso também.

Se estiver usando SSH com sua chave SSH cadastrada no GitHub, o comando push não pedirá senha e enviará suas alterações diretamente para o GitHub.

Caso você não tenha executado o comando `git remote add` como mostrado anteriormente, receberá o seguinte erro:

```
fatal: 'origin' does not appear to be a git repository
fatal: Could not read from remote repository.

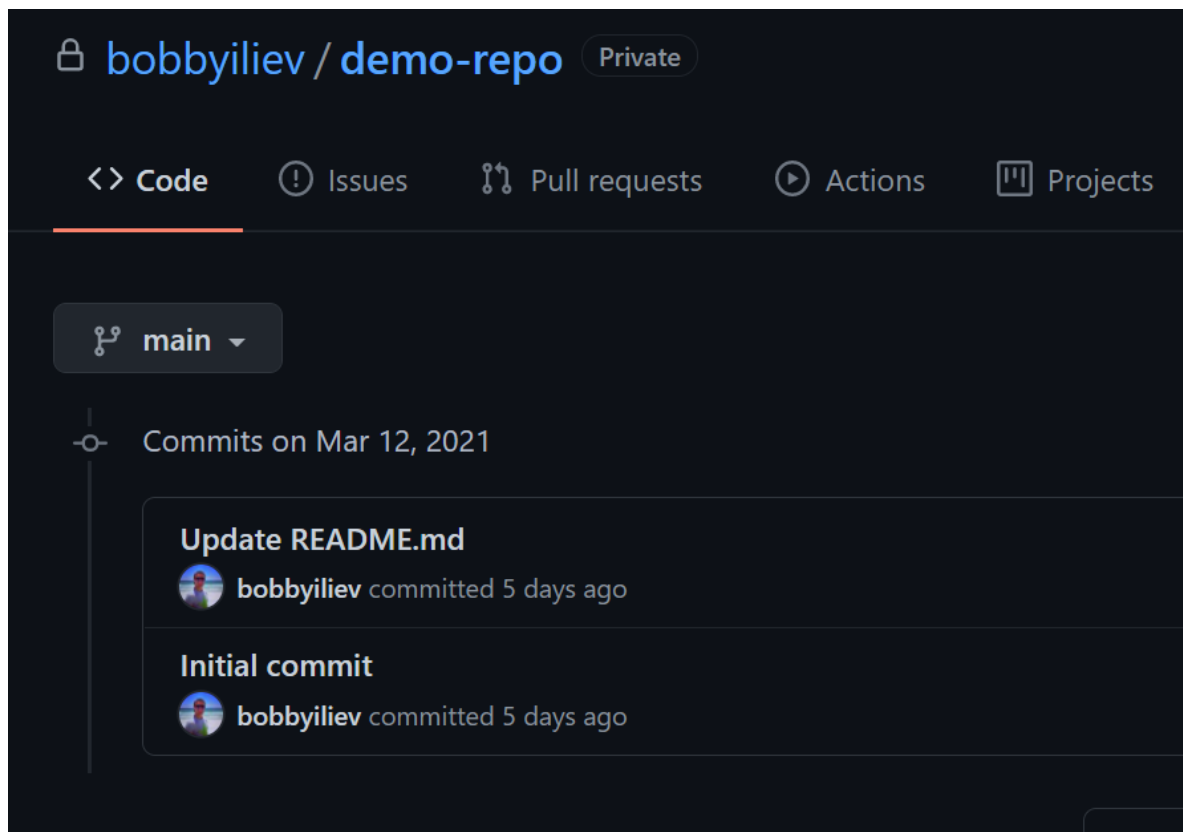
Please make sure you have the correct access rights
and the repository exists.
```

Isso significa que você não adicionou seu repositório do GitHub como repositório remoto. Por isso, usamos o comando `git remote add` para criar essa conexão entre seu repositório local e o remoto.

Note que a conexão já estará configurada se você usou o comando `git clone` para clonar um repositório existente do GitHub para sua máquina local. Vamos abordar o comando `git pull` nos próximos capítulos.

Verificando o repositório remoto

Após executar o comando `git push`, você pode acessar seu projeto no GitHub e verá os commits que fez localmente presentes no repositório remoto. Se clicar no link de `commits`, poderá ver todos os commits, assim como se tivesse executado o comando `git log`:



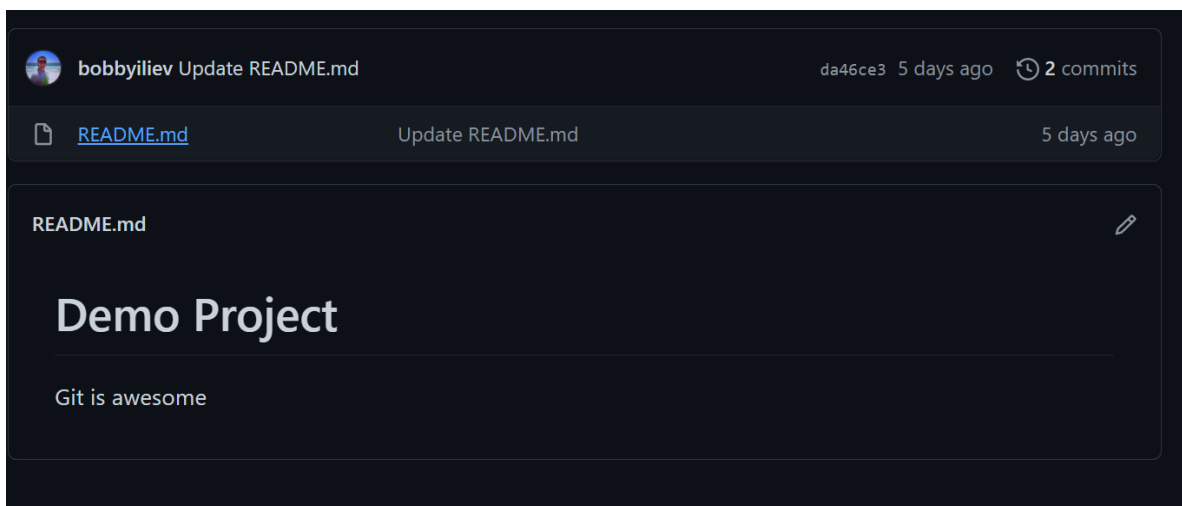
Agora que você sabe como enviar suas últimas alterações do seu projeto Git local para o repositório no GitHub, é hora de aprender como puxar as alterações mais recentes do GitHub para o seu projeto local.

Git Pull

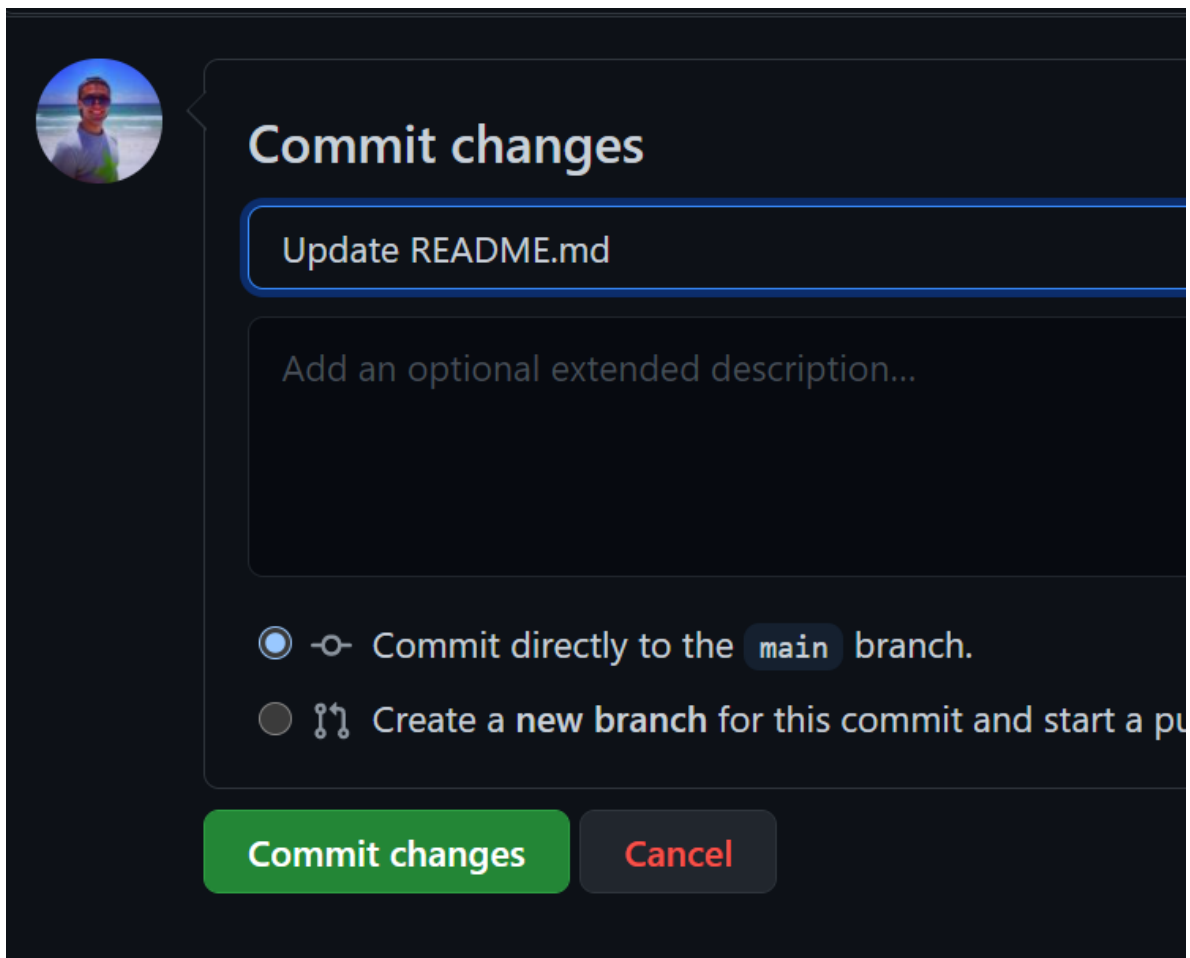
Se você está trabalhando em um projeto com várias pessoas, é provável que o código mude com frequência. Por isso, você precisa de uma forma de obter as alterações mais recentes do repositório do GitHub para sua máquina local.

Você já sabe que pode usar o comando `git push` para enviar seus commits mais recentes, então, para fazer o oposto e puxar os commits mais recentes do GitHub para o seu projeto local, você precisa usar o comando `git pull`.

Para testar isso, vamos fazer uma alteração diretamente no GitHub. Acesse o arquivo `README.md` e clique no ícone de lápis para editar o arquivo:



Faça uma pequena alteração no arquivo, adicione uma mensagem de commit descritiva e clique no botão `Commit Changes`:

A screenshot of the GitHub 'Commit changes' dialog. On the left is a circular profile picture of a person with glasses and a blue shirt. The main area has a title 'Commit changes' in white. Below it is a text input field containing 'Update README.md'. Underneath is a larger text area with the placeholder 'Add an optional extended description...'. At the bottom, there are two radio button options: the first is selected and says 'Commit directly to the main branch.', and the second is unselected and says 'Create a new branch for this commit and start a pull request'. At the very bottom are two buttons: a green 'Commit changes' button and a grey 'Cancel' button.

Commit changes

Update README.md

Add an optional extended description...

☒ Commit directly to the **main** branch.

☐ Create a new branch for this commit and start a pull request

Commit changes Cancel

Com isso, você fez um commit diretamente no GitHub, então seu repositório local ficará desatualizado em relação ao remoto.

Se você tentar enviar uma alteração agora para esse mesmo branch, irá falhar com o seguinte erro:

```
! [rejected]          main -> main (fetch first)
error: failed to push some refs to
'git@github.com:bobbyiliev/demo-repo.git'
hint: Updates were rejected because the remote contains work
that you do
hint: not have locally. This is usually caused by another
repository pushing
hint: to the same ref. You may want to first integrate the
remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help'
for details.
```


Como informado na saída, o repositório remoto está à frente do seu local, então você precisa rodar o comando `git pull` para obter as alterações mais recentes:

```
git pull origin main
```

A saída será parecida com esta:

```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 646 bytes | 646.00 KiB/s, done.
From github.com:bobbyiliev/demo-repo
* branch                main          -> FETCH_HEAD
   da46ce3..442afa5      main          -> origin/main

README.md | 3 ++-
1 file changed, 2 insertions(+), 1 deletion(-)
```

Podemos ver que o arquivo `README.md` foi alterado e que houve 2 novas linhas adicionadas e 1 linha removida.

Agora, se você rodar `git log`, verá o commit que fez no GitHub disponível localmente.

Claro, este é um cenário simplificado. No mundo real, você não faria alterações diretamente no GitHub, mas provavelmente trabalharia com outras pessoas no mesmo projeto e teria que puxar as alterações delas regularmente.

Você precisa garantir que puxa as alterações mais recentes sempre antes de tentar enviar suas próprias alterações.

Agora que você conhece os comandos básicos do Git, vamos aprender o que são Branches no Git.

Branches do Git

Até agora, trabalhamos apenas no nosso branch Main, que é criado por padrão ao criar um novo repositório no GitHub. Neste capítulo, você aprenderá mais sobre branches no Git, por que eles são necessários e como trabalhar com eles.

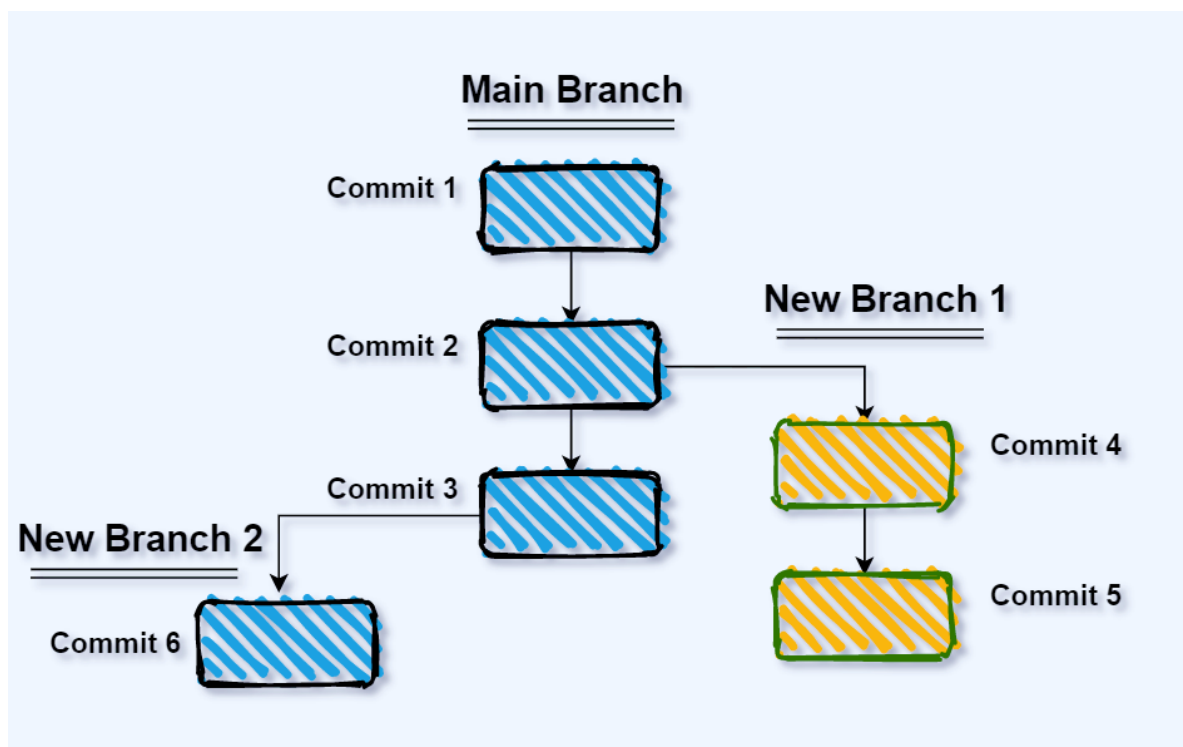
A definição oficial de branch do Git, segundo o site git-scm.com, é a seguinte:

Um branch no Git é simplesmente um ponteiro leve e móvel para um desses commits.

Isso pode ser um pouco confuso se você está começando agora. Então, pense nos branches como uma forma de trabalhar em seu projeto adicionando uma nova funcionalidade ou corrigindo bugs sem afetar o branch Main.

Assim, cada nova funcionalidade ou correção de bug que você estiver desenvolvendo pode viver em um branch separado e, depois que estiver pronto e tiver testado as alterações, você pode mesclar o novo branch ao branch principal. Você aprenderá mais sobre merge no próximo capítulo!

Se olharmos para a ilustração abaixo, onde temos alguns branches, você verá que parece uma árvore, daí o termo "branching":



Graças aos múltiplos branches, várias pessoas podem trabalhar em diferentes funcionalidades ou correções ao mesmo tempo, cada uma em seu próprio branch.

A imagem mostra 3 branches:

- O branch principal (main)
- Novo Branch 1
- Novo Branch 2

O branch principal é o branch padrão, que você já conhece. Podemos considerar os outros dois branches como duas novas funcionalidades que estão sendo desenvolvidas. Um desenvolvedor pode estar trabalhando em um novo formulário de contato para sua aplicação web no branch #1, e outro pode estar desenvolvendo uma funcionalidade de cadastro de usuários no branch #2.

Graças aos branches separados, ambos podem trabalhar no mesmo projeto sem atrapalhar um ao outro.

Agora, vamos aprender como criar novos branches e ver isso na prática!

Criando um novo branch

Vamos começar criando um novo branch chamado **novaFuncionalidade**. Para criar o branch, use o comando:

```
git branch novaFuncionalidade
```

Agora, para mudar para esse novo branch, execute:

```
git checkout novaFuncionalidade
```

Nota: Você pode usar o comando **git checkout** para alternar entre diferentes branches.

Os dois comandos acima podem ser combinados em um só, assim você não precisa criar o branch primeiro e depois mudar para ele. Use este comando:

```
git checkout -b novaFuncionalidade
```

Ao executar esse comando, verá a seguinte saída:

```
Switched to a new branch 'novaFuncionalidade'
```

Para verificar em qual branch você está atualmente, use:

```
git branch
```

Saída:

```
main
* novaFuncionalidade
```

Podemos ver que temos 2 branches: o **main** e o **novaFuncionalidade** que acabamos de criar. O asterisco antes do nome indica que estamos atualmente no branch **novaFuncionalidade**.

Se você usar o comando **git checkout** para voltar ao branch **main**:

```
git checkout main
```

E rodar **git branch** novamente, verá:

```
* main
novaFuncionalidade
```

Fazendo alterações no novo branch

Agora vamos fazer uma alteração no branch de nova funcionalidade. Primeiro, mude para o branch com o comando:

```
git checkout novaFuncionalidade
```

Nota: só é necessário usar o argumento `-b` ao criar novos branches.

Verifique se você está no branch correto:

```
git branch
```

Saída:

```
main
* novaFuncionalidade
```

Agora, crie um novo arquivo com conteúdo de demonstração:

```
echo "<h1>Meu Primeiro Branch de Funcionalidade</h1>" >
funcionalidade1.html
```

O comando acima irá criar o arquivo `funcionalidade1.html` com o conteúdo `<h1>Meu Primeiro Branch de Funcionalidade</h1>`.

Depois disso, prepare o arquivo e faça o commit:

```
git add funcionalidade1.html
git commit -m "Adicionar funcionalidade1.html"
```

O novo arquivo estará presente apenas no branch `novaFuncionalidade`. Se você mudar para o branch `main` e rodar o comando `ls` ou verificar o `git log`, verá que o arquivo não está lá.

Você pode verificar isso usando:

```
git log
```

Com isso, usamos vários dos comandos que vimos nos capítulos anteriores!

Comparando branches

Você também pode comparar dois branches com os comandos abaixo.

- Mostra os commits em **branchA** que não estão em **branchB**:

```
git log BranchA..BranchB
```

- Mostra as diferenças do que está em **branchA** mas não em **branchB**:

```
git diff BranchB...BranchA
```

Renomeando um branch

Se você criou um branch com o nome errado ou acha que o nome pode ser melhor, pode renomear com:

```
git branch -m nome-antigo nome-novo
```

Se quiser renomear o **branch atual**, basta rodar:

```
git branch -m nome-do-branch
```

Depois, se rodar **git branch** novamente, verá o nome correto.

Excluindo um branch

Para excluir um branch específico, execute:

```
git branch -d nome_do_branch
```

Isso exclui o branch apenas do seu repositório local. Se já tiver enviado o branch para

o GitHub, use:

```
git push origin --delete nome_do_branch
```

Se quiser sincronizar seus branches locais com os remotos, rode:

```
git fetch
```

Conclusão

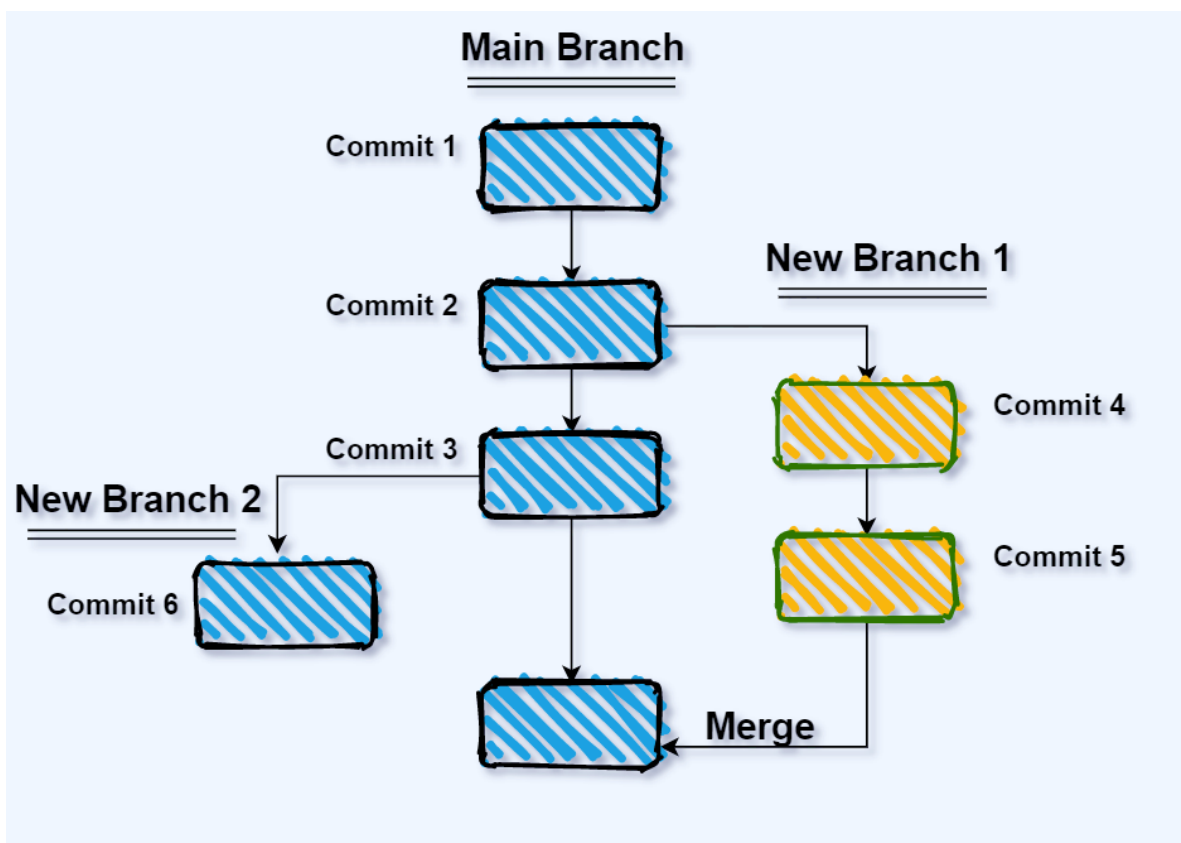
Com isso, nosso branch **novaFuncionalidade** está à frente do branch principal com 1 commit. Para trazer essas novas alterações para o branch principal, precisamos fazer o merge do branch **novaFuncionalidade** no **main**.

No próximo capítulo, você aprenderá como mesclar suas alterações de um branch para outro!

Uma coisa importante: antigamente, ao criar um novo repositório no GitHub, o nome padrão do branch era **master**. Agora, novos repositórios usam **main** como padrão, como parte do esforço do GitHub para usar termos mais inclusivos.

Git Merge

Quando os desenvolvedores terminam suas alterações, eles podem mesclar seus branches de funcionalidade ao branch principal e tornar essas funcionalidades disponíveis no site.



Se você seguiu os passos do capítulo anterior, seu branch **novaFuncionalidade** está à frente do branch principal com 1 commit. Para trazer essas novas alterações para o branch principal, precisamos fazer o merge do branch **novaFuncionalidade** no **main**.

Mesclando um branch

Siga estes passos:

- Primeiro, mude para o branch **main**:


```
git checkout main
```

- Depois, para mesclar o branch **novaFuncionalidade** e as alterações que criamos no capítulo anterior, execute:

```
git merge novaFuncionalidade
```

Saída:

```
Updating ab1007b..a281d25
Fast-forward
 funcionalidade1.html | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 funcionalidade1.html
```

Como você estava no branch **main** ao rodar o comando, o Git irá pegar todos os commits daquele branch e mesclá-los ao branch principal.

Agora, se rodar o comando **ls**, verá o novo arquivo **funcionalidade1.html**, e se verificar o histórico de commits com **git log**, verá o commit do branch **novaFuncionalidade** presente no branch principal.

Antes de fazer o merge, você pode usar o comando **git diff** para checar as diferenças entre seu branch atual e o branch que deseja mesclar. Por exemplo, se estiver no branch **main**, pode usar:

```
git diff novaFuncionalidade
```

Neste caso, o merge ocorreu sem conflitos, pois não havia conflitos. Porém, em projetos reais com várias pessoas, podem ocorrer conflitos. Isso acontece quando alterações são feitas na mesma linha de um arquivo, ou quando um desenvolvedor edita um arquivo em um branch e outro exclui o mesmo arquivo.

Resolvendo conflitos

Vamos simular um conflito. Para isso, crie um novo branch:

```
git checkout -b demoConflito
```

Depois edite o arquivo `funcionalidade1.html`:

```
echo "<p>Demonstração de Conflito</p>" >> funcionalidade1.html
```

O comando acima adiciona `<p>Demonstração de Conflito</p>` ao final do arquivo. Você pode verificar o conteúdo com:

```
cat funcionalidade1.html
```

Saída:

```
<h1>Meu Primeiro Branch de Funcionalidade</h1>
<p>Demonstração de Conflito</p>
```

Você pode rodar `git status` e `git diff` para ver o que foi modificado antes de fazer o commit.

Depois, faça o commit da alteração:

```
git commit -am "Demo de Conflito 1"
```

Note que não usamos o comando `git add`, mas sim a flag `-a`, que significa `add`. Você pode usar isso para arquivos já rastreados pelo git e que foram modificados. Para arquivos novos, é preciso usar `git add` primeiro.

Agora, volte para o branch principal:

```
git checkout main
```

Se verificar o arquivo `funcionalidade1.html`, verá apenas a linha `<h1>Meu Primeiro Branch de Funcionalidade</h1>`, pois a alteração feita está apenas no branch `demoConflito`.

Agora, faça uma alteração no mesmo arquivo:

```
echo "<p>Conflito: alteração no branch principal</p>" >>
funcionalidade1.html
```

Adicionamos uma linha ao final do arquivo com conteúdo diferente.

Prepare e faça o commit da alteração:

```
git commit -am "Conflito no main"
```

Agora, tanto o branch principal quanto o branch **demoConflito** têm alterações no mesmo arquivo, na mesma linha. Vamos rodar o comando de merge e ver o que acontece:

```
git merge demoConflito
```

Saída:

```
Auto-merging funcionalidade1.html
CONFLICT (content): Merge conflict in funcionalidade1.html
Automatic merge failed; fix conflicts and then commit the
result.
```

Como vemos, o merge falhou pois houve alterações no mesmo arquivo, na mesma linha, e o Git não sabe qual alteração é a correta.

Existem várias formas de resolver conflitos. Aqui vamos mostrar uma delas.

Se você verificar o conteúdo do arquivo **funcionalidade1.html**, verá:

```
<h1>Meu Primeiro Branch de Funcionalidade</h1>
<<<<<<< HEAD
<p>Conflito: alteração no branch principal</p>
=====
<p>Demonstração de Conflito</p>
>>>>>>> demoConflito
```

Inicialmente pode parecer confuso, mas vamos revisar:

- **<<<<<<< HEAD**: indica o início das alterações do branch atual. No nosso caso, a linha **<p>Conflito: alteração no branch principal</p>** está no branch principal, que é o branch atual.
- **=====**: indica onde terminam as alterações do branch atual e começam as do novo branch. No nosso caso, a alteração do novo branch é **<p>Demonstração de Conflito</p>**.
- **>>>>>>> demoConflito**: indica o nome do branch de onde vêm as alterações.

Você pode resolver o conflito removendo manualmente as linhas que não são necessárias, deixando o arquivo assim:

```
<h1>Meu Primeiro Branch de Funcionalidade</h1>
<p>Conflito: alteração no branch principal</p>
```

Se estiver usando uma IDE como o VS Code, ela permite escolher quais alterações manter com um clique.

Depois de resolver o conflito, é necessário fazer outro commit:

```
git commit -am "Resolver conflito de merge"
```

Conclusão

Branches e merges do Git permitem que você trabalhe em projetos colaborativos. Uma dica importante é garantir que você puxe as alterações do branch principal local regularmente para não ficar atrás do remoto.

Alguns comandos úteis para quem já está confortável com o que vimos até agora são **git rebase** e **git cherry-pick**, que permitem escolher quais commits de um

branch específico você quer trazer para o branch atual.

Revertendo alterações

Como em tudo, existem várias maneiras de fazer uma determinada coisa. Mas o que eu normalmente faço quando quero desfazer meu último commit e então commitar minhas novas alterações é o seguinte.

- Suponha que você fez algumas alterações e realizou o commit:

```
git commit -m "Commitando alterações erradas"
```

- Depois disso, se você rodar `git log`, verá o histórico de tudo que foi commitado no repositório.
- Infelizmente, após o commit das alterações erradas, você percebe que esqueceu de adicionar arquivos ao commit ou esqueceu de fazer uma pequena alteração nos arquivos já commitados.
- Para resolver isso, basta fazer essas alterações e prepará-las com `git add`, então você pode "amendar" o último commit rodando o seguinte comando:

```
git commit --amend
```

Nota: O comando acima também permite alterar a mensagem do commit, se necessário.

Resetando alterações (⚠ Resetar é perigoso ⚠)

É preciso ter cuidado com comandos de reset, pois eles apagam commits do repositório e os removem do histórico.

Exemplo:

```
git reset --soft HEAD~1
```

O comando acima irá resetar voltando 1 commit.

Nota: O comando acima desfaz seu commit, mas mantém as alterações no código. Se quiser se livrar das alterações também, faça um reset hard: `git reset --hard HEAD~1`

Sintaxe:

```
git reset [--soft|--hard] [<referência-do-commit>]
```

- Depois disso, faça suas novas alterações
- Quando terminar, rode `git add` para adicionar os arquivos que deseja incluir no próximo commit:

```
git add .
```

- Em seguida, use `git commit` normalmente para commitar suas novas alterações:

```
git commit -m "Sua nova mensagem de commit"
```

- Depois disso, você pode novamente verificar o histórico rodando:

```
git log
```

Aqui está um exemplo do processo:


```

root@do-dev:~/demo# git init .
Initialized empty Git repository in /root/demo/.git/
root@do-dev:~/demo# touch demo.txt
root@do-dev:~/demo# git add .
root@do-dev:~/demo# git commit -m "First commit"
[master (root-commit) 45f651d] First commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 demo.txt
root@do-dev:~/demo# echo "Wrong changes..." > demo.txt
root@do-dev:~/demo# git add .
root@do-dev:~/demo# git commit -m "Wrong commit..."
[master 9688e23] Wrong commit...
 1 file changed, 1 insertion(+)
root@do-dev:~/demo# git log
commit 9688e23761a6ccbbfaa4362a391b50a63d4ba39a (HEAD -> master)
Author: Bobby Iliev <bobby@bobbyiliev.com>
Date:   Wed Jan 8 08:49:49 2020 +0000

    Wrong commit...

commit 45f651dd4e83205fe1a72ac16bf0d7a3ecfba904
Author: Bobby Iliev <bobby@bobbyiliev.com>
Date:   Wed Jan 8 08:49:27 2020 +0000

    First commit
root@do-dev:~/demo# git reset --soft HEAD~1
root@do-dev:~/demo# git log
commit 45f651dd4e83205fe1a72ac16bf0d7a3ecfba904 (HEAD -> master)
Author: Bobby Iliev <bobby@bobbyiliev.com>
Date:   Wed Jan 8 08:49:27 2020 +0000

    First commit
root@do-dev:~/demo# echo "Fixed changes..." > demo.txt
root@do-dev:~/demo# git add .
root@do-dev:~/demo# git commit -m "Fixed commit..."
[master 081f0fb] Fixed commit...
 1 file changed, 1 insertion(+)
root@do-dev:~/demo# git log
commit 081f0fbf3d32ad7e7946e1ec4cc5b432fc699fef (HEAD -> master)
Author: Bobby Iliev <bobby@bobbyiliev.com>
Date:   Wed Jan 8 08:50:28 2020 +0000

    Fixed commit...

commit 45f651dd4e83205fe1a72ac16bf0d7a3ecfba904
Author: Bobby Iliev <bobby@bobbyiliev.com>
Date:   Wed Jan 8 08:49:27 2020 +0000

    First commit
root@do-dev:~/demo# █

```

Nota: Você pode resetar suas alterações voltando mais de um commit usando a seguinte sintaxe:

```
git reset --soft HEAD~n
```

onde **n** é o número de commits que você quer voltar.

Outra abordagem seria usar **git revert COMMIT_ID**.

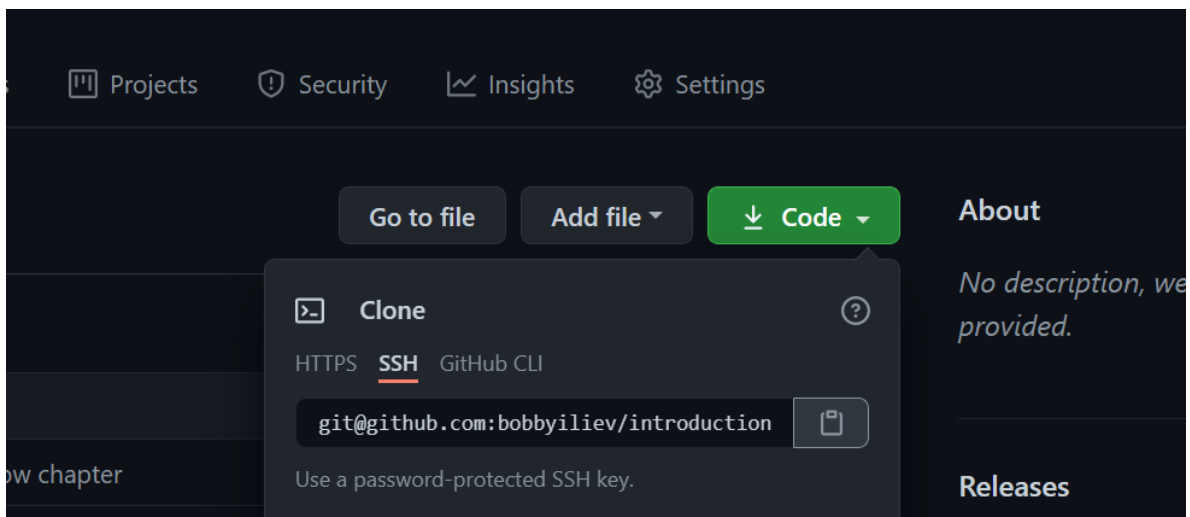
Aqui está um vídeo rápido demonstrando como fazer isso:

[Revertendo alterações](#)

Git Clone

Na maioria das vezes, em vez de começar um projeto do zero, você vai entrar em uma empresa e começar a trabalhar em um projeto existente, ou contribuir para um projeto open source já estabelecido. Nesse caso, para obter o repositório do GitHub para sua máquina local, você precisa usar o comando `git clone`.

A maneira mais simples de clonar seu repositório do GitHub é primeiro visitar o repositório no navegador, clicar no botão verde `Code` e escolher o método que deseja usar para clonar o repositório:



No meu caso, prefiro o método SSH, pois já tenho minhas chaves SSH configuradas conforme o capítulo 14.

Como estou clonando este repositório [aqui](#), a URL ficará assim:

```
git@github.com:bobbyiliev/introduction-to-bash-scripting.git
```

Depois de copiar isso, volte ao terminal, vá até o diretório onde deseja clonar o repositório e execute o seguinte comando:

```
git clone git@github.com:bobbyiliev/introduction-to-bash-scripting.git
```

A saída será parecida com isto:

```
Cloning into 'introduction-to-bash-scripting'...
remote: Enumerating objects: 21, done.
remote: Counting objects: 100% (21/21), done.
remote: Compressing objects: 100% (16/16), done.
remote: Total 215 (delta 7), reused 14 (delta 4), pack-reused
194
Receiving objects: 100% (215/215), 3.08 MiB | 5.38 MiB/s,
done.
Resolving deltas: 100% (114/114), done.
```

Basicamente, o comando `git clone` faz o download do repositório do GitHub para sua pasta local.

Agora você pode começar a fazer alterações no projeto criando um novo branch, escrevendo código e finalmente commitando e enviando suas alterações!

Uma coisa importante a lembrar é que, caso você não seja o mantenedor do repositório e não tenha permissão para enviar alterações diretamente, será necessário primeiro fazer um fork do repositório original e então clonar o repositório forkado da sua conta. No próximo capítulo, veremos o processo completo de como fazer um fork de um repositório!

Fork no Git

Ao contribuir para um projeto open source, você não poderá fazer alterações diretamente no projeto. Somente os mantenedores do repositório têm esse privilégio.

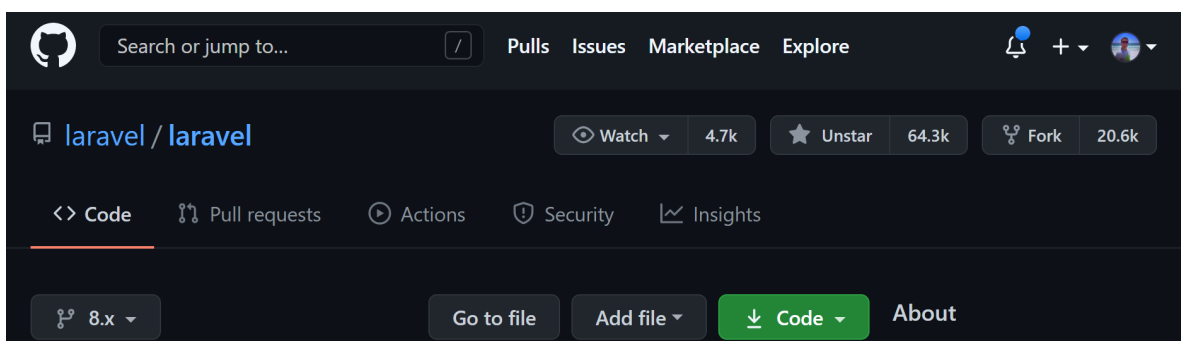
O que você precisa fazer é fazer um fork do repositório específico, realizar as alterações no projeto forkado e então enviar um pull request para o projeto original. Você aprenderá mais sobre pull requests nos próximos capítulos.

Se você clonar um repositório ao qual não tem acesso e tentar enviar alterações diretamente para esse repositório, receberá o seguinte erro:

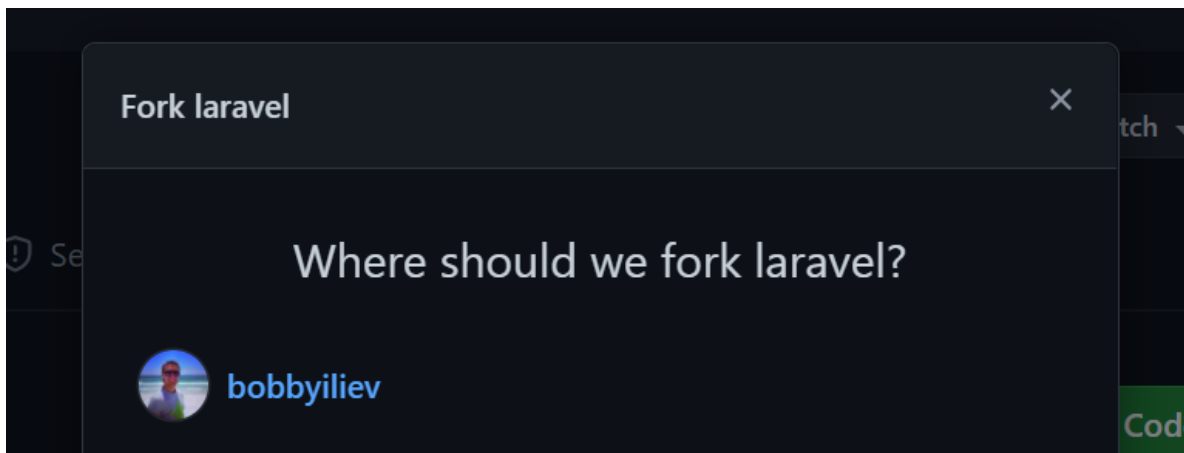
```
ERROR: Permission to laravel/laravel.git denied to bobbyiliev.  
Fatal: Could not read from remote repository.  
  
Please make sure you have the correct access rights  
and the repository exists.
```

É aí que entra o Fork!

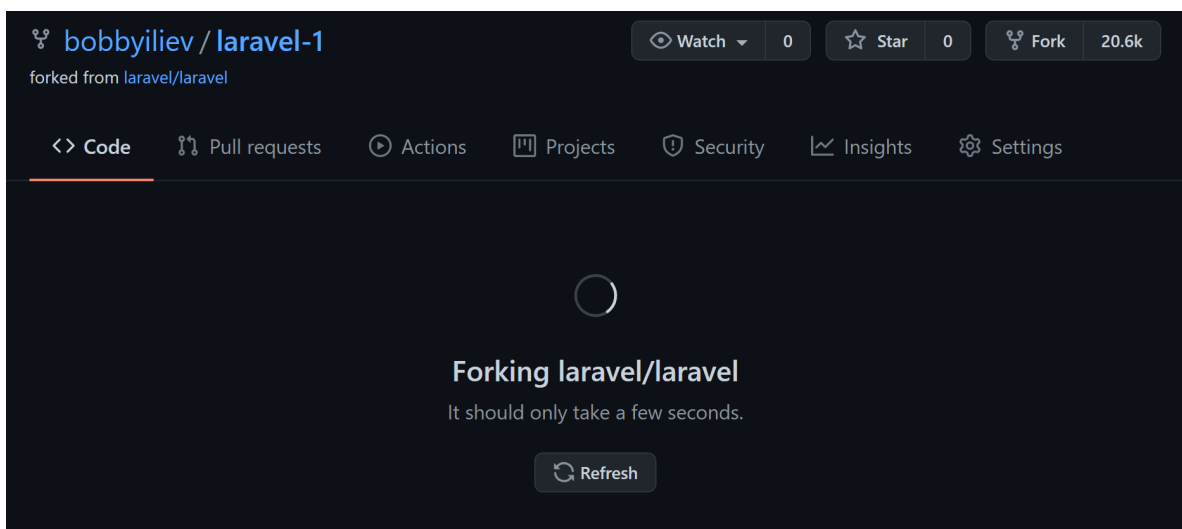
Para fazer um fork de um repositório, acesse o repositório pelo navegador e clique no botão Fork no canto superior direito:



Depois, escolha a conta para a qual deseja fazer o fork do repositório:



Pode levar alguns segundos para o repositório ser forkado para sua conta:



Com isso, você terá uma cópia exata do repositório em questão na sua conta.

A vantagem é que agora você pode clonar o repositório forkado da sua conta, fazer as alterações normalmente e, quando estiver pronto, enviar um pull request para o repositório original contribuindo com suas alterações.

Como já mencionamos pull requests algumas vezes, vamos aprender mais sobre eles no próximo capítulo!

Fluxo de Trabalho no Git

Agora que você conhece os comandos básicos, vamos juntar tudo e passar por um fluxo de trabalho básico com Git.

Normalmente, o fluxo de trabalho é assim:

- Primeiro, você clona um projeto existente com o comando `git clone`, ou, se estiver começando um novo projeto, inicializa com o comando `git init`.
- Depois disso, antes de começar a alterar o código, é melhor criar um novo branch Git onde irá trabalhar. Você pode fazer isso com o comando `git checkout -b NOME_DO_SEU_BRANCH`.
- Com o branch pronto, você começa a fazer as alterações no seu código.
- Quando terminar as alterações, precisa prepará-las com o comando `git add`.
- Para salvar/registrar as alterações no seu repositório Git local, execute o comando `git commit` e forneça uma mensagem de commit descritiva.
- Para enviar suas alterações locais para o projeto remoto no GitHub, use o comando `git push origin NOME_DO_SEU_BRANCH`.
- Por fim, depois de enviar suas alterações, você precisa abrir um pull request (PR) do seu branch para o branch principal do repositório.
- É considerado uma boa prática adicionar algumas pessoas como revisores e pedir para revisarem as alterações.
- Finalmente, depois que as alterações forem aprovadas, o PR será mesclado ao branch principal, trazendo todas as suas alterações do seu branch para o branch principal.

O processo geral fica assim:



Minha dica é criar um novo repositório e passar por esse processo algumas vezes até se sentir totalmente confortável com todos os comandos.

Pull Requests

Você já sabe como mesclar alterações de um branch para outro no seu repositório Git local.

Para fazer o mesmo no GitHub, você precisa abrir um Pull Request (ou Merge Request, se estiver usando GitLab), ou PR para abreviar, e solicitar a mesclagem do seu branch de funcionalidade para o branch **main**.

Os passos para abrir um Pull Request são:

- Se você está trabalhando em um projeto open source do qual não é mantenedor, primeiro faça um fork do repositório conforme explicado no capítulo 21. Pule esta etapa se você for o mantenedor do repositório.
- Depois, clone o repositório localmente com o comando **git clone**:

```
git clone git@github.com:seu_usuario/seu_repositorio
```

- Crie um novo branch com o comando **git checkout**:

```
git checkout -b nome_do_branch
```

- Faça suas alterações no código
- Prepare as alterações com **git add**:

```
git add .
```

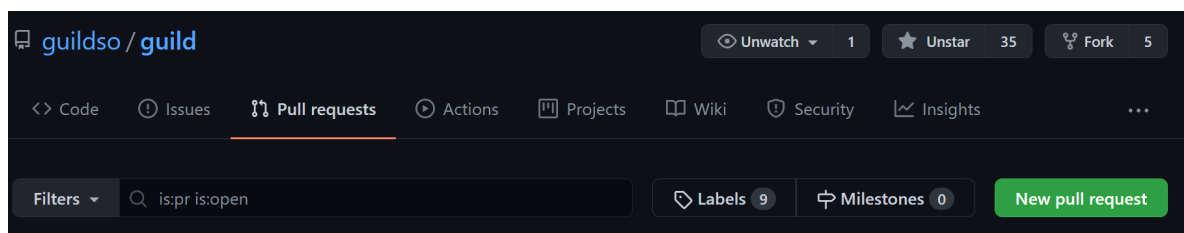
- E então faça o commit com **git commit**:

```
git commit -m "Mensagem do Commit"
```

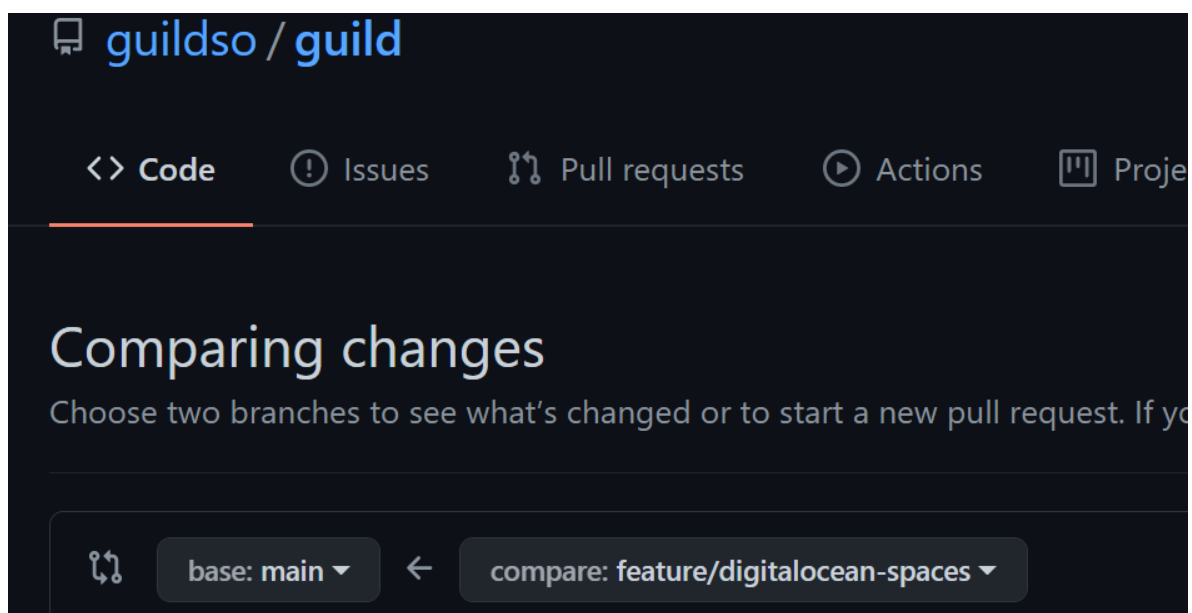
- Depois envie seu novo branch para o GitHub com **git push**:

```
git push origin nome_do_branch
```

- Após isso, acesse o repositório no GitHub e clique no botão **Pull Requests** e depois no botão verde **New pull request**:



- Lá, escolha o branch para o qual deseja mesclar e o branch de onde deseja mesclar:



- Depois revise as alterações, adicione um título e descrição e clique no botão de criar.
- Se estiver trabalhando em um projeto com múltiplos colaboradores, selecione alguns revisores. Revisores são pessoas que você gostaria que revisassem seu código antes de ele ser mesclado ao branch **main**.

Para uma representação visual de todo o processo, confira este tutorial passo a passo:

- [Como enviar seu primeiro Pull Request no GitHub](#)

Git e VS Code

Por mais que eu goste de usar o terminal para minhas tarefas diárias, no final das contas prefiro realizar várias tarefas em uma única janela (GUI) ou fazer tudo diretamente pelo terminal.

No passado, eu usava editores de texto (vim, nano, etc.) no terminal para editar o código dos meus repositórios e depois utilizava o cliente git para commitar minhas alterações. Mas depois migrei para o Visual Studio Code para gerenciar e desenvolver meu código.

Recomendo que você confira este artigo sobre por que usar o Visual Studio. É um artigo do próprio site do Visual Studio.

[Por que você deve usar o Visual Studio](#)

O Visual Studio Code possui gerenciamento de controle de versão integrado (SCM) e inclui suporte ao Git nativamente. Muitos outros provedores de controle de versão estão disponíveis através de extensões no Marketplace do VS Code. Ele também suporta múltiplos provedores de controle de versão simultaneamente, permitindo abrir todos os seus projetos ao mesmo tempo e fazer alterações sempre que necessário.

Instalando o VS Code

Você precisa instalar o Visual Studio Code. Ele roda nos sistemas operacionais macOS, Linux e Windows.

Siga os guias específicos para cada plataforma abaixo:

- [macOS](#)
- [Linux](#)
- [Windows](#)

Você precisa instalar o Git antes de obter esses recursos. Certifique-se de instalar pelo menos a versão 2.0.0. Se não tiver o git instalado em sua máquina, confira este artigo útil sobre [Como começar com Git](#)

Você precisa definir seu nome de usuário e e-mail na configuração do Git, ou o git irá usar informações da sua máquina local ao commitar. Precisamos fornecer essas informações porque o Git as insere em cada commit que fazemos.

Para definir isso, execute os seguintes comandos:

```
git config --global user.name "John Doe"
git config --global user.email "johnde@domain.com"
```

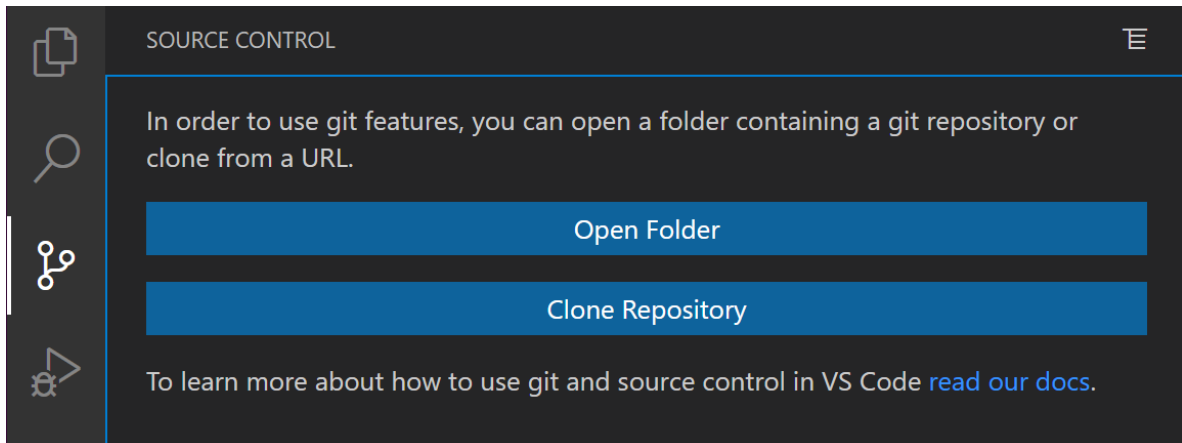
As informações serão salvas no seu arquivo `~/.gitconfig`.

```
[user]
  name = John Doe
  email = johndoe@domain.com
```

Com o Git instalado e configurado em sua máquina local, você está pronto para usar o Git para controle de versão com o Visual Studio ou usando o terminal.

Clonando um repositório no VS Code

O bom é que o Visual Studio detecta automaticamente se você abriu uma pasta que é um repositório. Se já abriu um repositório, ele será visível na Visualização de Controle de Código Fonte.



Se ainda não abriu uma pasta, a visualização de controle de código fonte dará a opção de Abrir Pasta da sua máquina local ou Clonar Repositório.

Se selecionar Clonar Repositório, será solicitado a URL do repositório remoto (por exemplo, no GitHub) e o diretório pai onde o repositório local será colocado.

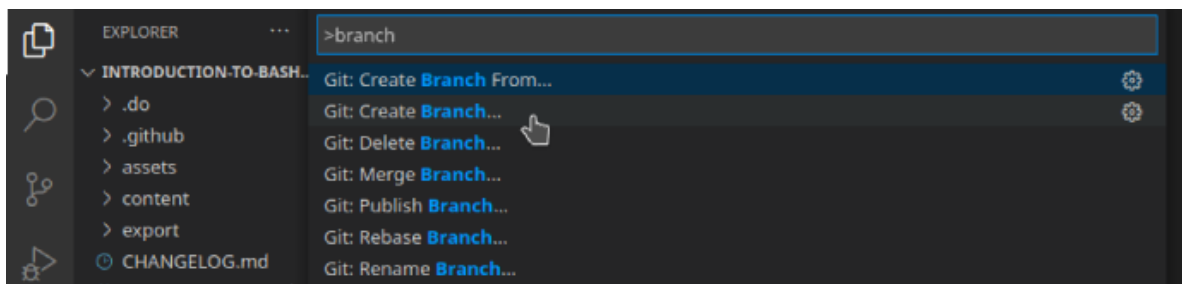
Para um repositório do GitHub, você encontra a URL na caixa de diálogo Code do GitHub.

Criar um branch

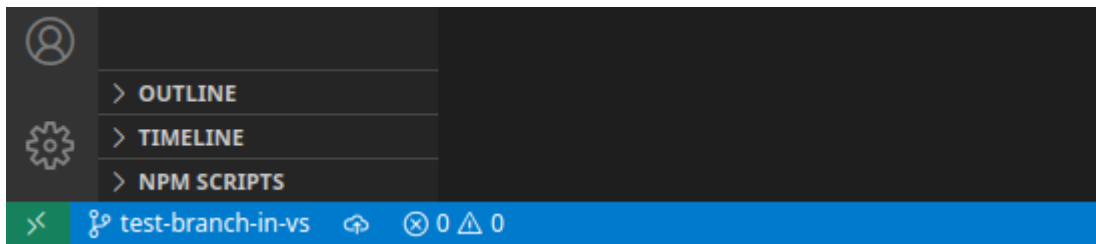
Para criar um branch, abra o comando pallet:

- Windows: Ctrl + Shift + P
- Linux: Ctrl + Shift + P
- MacOS: Shift + CMD + P

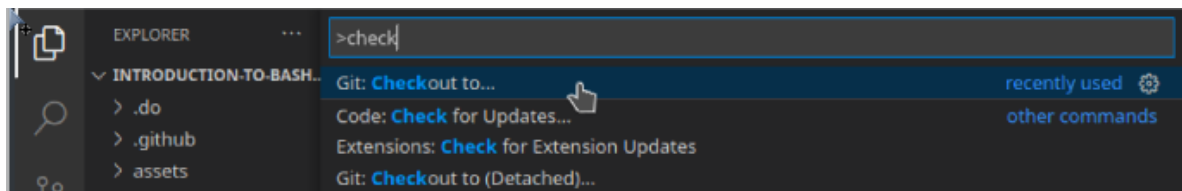
E selecione **Git Create Branch...**



Depois basta digitar um nome para o branch. Lembre-se que no canto inferior esquerdo você pode ver em qual branch está. O padrão será o main, e se criar o branch com sucesso, verá o nome do novo branch criado.



Se quiser alternar entre branches, abra o comando pallet e procure por **Git checkout to** e selecione o branch main ou outro branch.



Configurar um template de mensagem de commit

Se quiser agilizar o processo e ter um template pré-definido para suas mensagens de commit, pode criar um arquivo simples com essas informações.

Para isso, abra o terminal se estiver no Linux ou macOS e crie o seguinte arquivo: `.gitmessage` no seu diretório pessoal. Para criar o arquivo, abra-o no seu editor favorito, coloque o conteúdo padrão desejado e salve. Exemplo de conteúdo:

```
cat ~/.gitmessage
```

```
#Título
```

```
#Resumo do commit
```

```
#Inclua Co-authored-by para todos os colaboradores.
```

Para informar ao Git para usar esse arquivo como template padrão que aparece no editor ao rodar `git commit`, defina o valor da configuração `commit.template`:

```
$ git config --global commit.template ~/.gitmessage
$ git commit
```


Conclusão

Se você prefere programar no Visual Studio Code e também usa controle de versão, recomendo que experimente interagir com os repositórios no VS Code. Acredito que cada um tem seu estilo e pode fazer as coisas de forma diferente dependendo do momento. Desde que você possa adicionar/modificar seu código e depois commitar as alterações no repositório, não existe um jeito certo ou errado de fazer isso. Por exemplo, você pode editar seu código no vim e enviar as alterações usando o git no terminal, ou programar no Visual Studio e commitar usando o terminal. Você é livre para fazer do jeito que achar mais conveniente. Usar o git dentro do VS Code pode tornar seu fluxo de trabalho mais eficiente e robusto.

Fontes adicionais:

- [Controle de Versão](#) - Leia mais sobre o suporte integrado ao Git.
- [Visão geral da configuração](#) - Configure e comece a usar o VS Code.
- [GitHub com Visual Studio](#) - Leia mais sobre o suporte ao GitHub no VS Code
- Você também pode conferir este mini tutorial em vídeo sobre como usar o básico do controle de versão Git no Visual Studio Code

Fonte:

- Contribuído por: [Alex Georgiev](#).
- Publicado originalmente [aqui](#).

GitHub CLI

A GitHub CLI ou **gh** é basicamente o GitHub na linha de comando.

Você pode interagir com sua conta do GitHub diretamente pelo terminal e gerenciar coisas como pull requests, issues e outras ações do GitHub.

Neste tutorial, vou dar uma visão rápida de como instalar o **gh** e como usá-lo!

Instalação do GitHub CLI

Como estou usando Ubuntu, para instalar o **gh** você precisa rodar os seguintes comandos:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-key  
C99B11DEB97541F0  
sudo apt-add-repository https://cli.github.com/packages  
sudo apt update  
sudo apt install gh
```

Se você estiver no Mac, pode instalar o **gh** usando o Homebrew:

```
brew install gh
```

Para outros sistemas operacionais, recomendo seguir os passos da documentação oficial [aqui](#).

Depois de instalar o **gh**, você pode verificar se está funcionando com o comando:

```
gh --version
```

Isso mostrará a versão do **gh**:

```
gh version 1.0.0 (2020-09-16)  
https://github.com/cli/cli/releases/tag/v1.0.0
```

No meu caso, estou usando a versão mais recente **gh** v1.0.0, lançada há poucos dias.

Autenticação

Depois de instalar o **gh**, você precisa fazer login na sua conta do GitHub.

Para isso, execute o comando:

```
gh auth login
```

Você verá a seguinte saída:

```
? What account do you want to log into? [Use arrows to move,  
type to filter]  
> GitHub.com  
  GitHub Enterprise Server
```

Você pode escolher entre GitHub.com ou GitHub Enterprise. Pressione enter e siga o processo de autenticação.

Outro comando útil é o **gh help**. Ele mostra uma lista dos comandos disponíveis do **gh** que você pode usar:

USAGE

gh <command> <subcommand> [flags]

CORE COMMANDS

gist: Criar gists
issue: Gerenciar issues
pr: Gerenciar pull requests
release: Gerenciar releases do GitHub
repo: Criar, clonar, fazer fork e visualizar repositórios

ADDITIONAL COMMANDS

alias: Criar atalhos de comando
api: Fazer uma requisição autenticada à API do GitHub
auth: Login, logout e atualizar autenticação
completion: Gerar scripts de autocompletar para o shell
config: Gerenciar configuração do gh
help: Ajuda sobre qualquer comando

FLAGS

--help Mostrar ajuda do comando
--version Mostrar versão do gh

EXEMPLOS

```
$ gh issue create  
$ gh repo clone cli/cli  
$ gh pr checkout 321
```

VARIÁVEIS DE AMBIENTE

Veja '[gh help environment](#)' para a lista de variáveis de ambiente suportadas.

SAIBA MAIS

Use '[gh <command> <subcommand> --help](#)' para mais informações sobre um comando.

Leia o manual em <https://cli.github.com/manual>

FEEDBACK

Abra uma issue usando '[gh issue create -R cli/cli](#)'

Agora vamos clonar um projeto existente para testar. Como exemplo, podemos usar o repositório [LaraSail](#). Em vez de usar o comando padrão **git clone**, vamos usar o **gh**:

```
gh repo clone thedevdojo/larasail
```

Você verá a seguinte saída:

```
Cloning into 'larasail'...
```

Depois disso, entre na pasta:

```
cd larasail
```

Agora estamos prontos para usar alguns dos comandos mais úteis do **gh**!

Comandos úteis do GitHub CLI

Usando o **gh**, você pode obter praticamente todas as informações do seu repositório no GitHub sem sair do terminal.

Aqui estão alguns comandos úteis:

Trabalhando com issues do GitHub

Para listar todas as issues abertas, execute:

```
gh issue list
```

A saída será:

```
Showing 4 of 4 open issues in thedevdojo/larasail

#25  Add option to automatically create database
      (enhancement)  about 3 months ago
#22  Remove PHP mcrypt as it is no longer needed
      about 3 months ago
#11  Add redis support
      about 8 months ago
#10  Wondering about the security of storing root MySQL
      password in /etc/.larasail/tmp/mysqlpass          about
      3 months ago
```

Você pode criar uma nova issue com o comando:

```
gh issue create --label bug
```

Ou visualizar uma issue existente:

```
gh issue view '#25'
```


Isso retorna todas as informações da issue específica:

```
Add option to automatically create a database
Open • bobbyiliev opened about 3 months ago • 0 comments

Labels: enhancement

Add an option to automatically create a new database, a
database user and
possibly update the database details in the .env file for a
specific project

View this issue on GitHub:
https://github.com/thedevdojo/larasail/issues/25
```

Trabalhando com seu repositório GitHub

Você pode usar o comando `gh repo` para criar, clonar ou visualizar um repositório existente:

```
gh repo create
gh repo clone cli/cli
gh repo view --web
```

Por exemplo, ao rodar `gh repo view`, você verá as informações do README do seu repositório diretamente no terminal.

Trabalhando com Pull Requests

Você pode usar o comando `gh pr` com vários argumentos para gerenciar seus pull requests.

Algumas opções disponíveis são:

checkout:	Fazer checkout de um pull request no git
checks:	Mostrar status de CI para um pull request
close:	Fechar um pull request
create:	Criar um pull request
diff:	Ver alterações em um pull request
list:	Listar e filtrar pull requests neste repositório
merge:	Mesclar um pull request
ready:	Marcar um pull request como pronto para revisão
reopen:	Reabrir um pull request
review:	Adicionar uma revisão a um pull request
status:	Mostrar status dos pull requests relevantes
view:	Visualizar um pull request

Com esses comandos, você está pronto para executar as principais ações do GitHub diretamente no terminal!

Conclusão

Agora você sabe o que é a ferramenta GitHub CLI e como usá-la! Para mais informações, recomendo conferir a documentação oficial:

<https://cli.github.com/manual/>

Sou fã de ferramentas de linha de comando! Elas podem facilitar sua vida e automatizar muitas tarefas repetitivas do dia a dia!

Publicado originalmente aqui: [O que é GitHub CLI e como começar](#)

Git Stash

`git stash` é um comando útil que ajuda em situações em que você precisa guardar temporariamente suas alterações locais e resetar seu código para o commit mais recente, para trabalhar em um bug ou funcionalidade mais urgente.

Em outras palavras, esse comando permite reverter seu diretório de trabalho para coincidir com o commit `HEAD`, mantendo todas as modificações locais salvas.

Quando estiver pronto para voltar ao código que guardou, basta restaurá-lo com um único comando!

Guardando seu trabalho

```
git stash
```

Por exemplo, considere um arquivo chamado `index.html` que foi modificado desde o último commit.

```
Rajat@DESKTOP-J8MKHE6 MINGW64 ~/Desktop/my-project (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")

Rajat@DESKTOP-J8MKHE6 MINGW64 ~/Desktop/my-project (master)
$ git stash
Saved working directory and index state WIP on master: d8bdd24 initial commit

Rajat@DESKTOP-J8MKHE6 MINGW64 ~/Desktop/my-project (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Note que ao rodar o comando `git status` após o comando `git stash`, não há mais alterações pendentes!

Aqui, WIP significa Work-In-Progress e é usado para indexar as várias cópias guardadas do seu trabalho.

Um ponto importante antes de guardar todas as alterações é que, por padrão, o **git stash não guarda arquivos não rastreados e ignorados**. (Arquivos não rastreados são aqueles que não faziam parte do último commit, ou seja, arquivos novos no seu repositório local)

Se quiser incluir esses arquivos não rastreados no stash, use a opção **-u**.

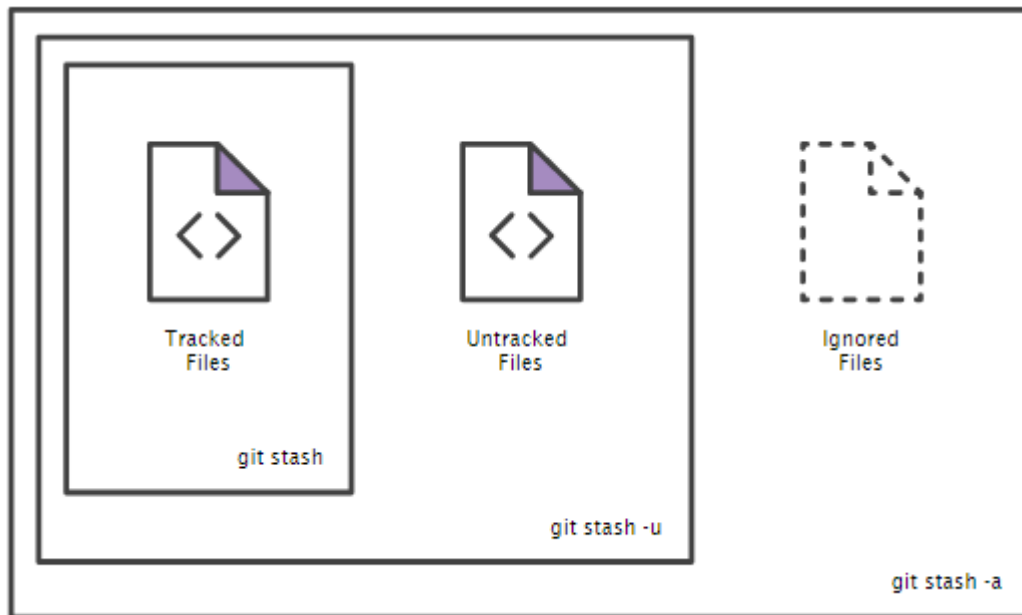
```
git stash -u
```

Da mesma forma, todos os arquivos listados no `.gitignore` (arquivos ignorados)

também serão excluídos do stash. Para incluí-los, use a opção **-a**:

```
git stash -a
```

As ilustrações abaixo mostram o comportamento do comando **git stash** quando essas opções são usadas:



Restaurando as alterações guardadas

```
git stash apply
```

Esse comando é usado para reaplicar todas as modificações locais feitas antes de guardar o trabalho.

Outro comando que pode ser usado para isso é o `git stash pop`. Aqui, "pop" significa remover o conteúdo mais recente do stash e reaplicá-lo ao seu diretório de trabalho.

A diferença entre os dois comandos é que o `git stash pop` **remove essas alterações do stash**, enquanto o `git stash apply` **mantém as alterações guardadas** mesmo após restaurá-las.

Considere o exemplo anterior, em que o arquivo `index.html` foi guardado. Na imagem abaixo, você vê como restaurar essas alterações afeta seu repositório local.

```
Rajat@DESKTOP-J8MKHE6 MINGW64 ~/Desktop/my-project (master)
$ git status
On branch master
nothing to commit, working tree clean

Rajat@DESKTOP-J8MKHE6 MINGW64 ~/Desktop/my-project (master)
$ git stash apply
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Mas e se você tiver vários stashes e não souber qual deseja restaurar? É aí que entra o próximo comando!

Lidando com múltiplas cópias guardadas do seu trabalho

Assim como ao resetar o repositório local para um commit específico, o primeiro passo para lidar com múltiplos stashes é **ver as várias cópias guardadas disponíveis**.

```
git stash list
```

Esse comando mostra uma lista indexada de todos os stashes disponíveis junto com uma **mensagem correspondente ao commit recente**.

Veja o exemplo abaixo, onde há dois stashes disponíveis: um quando um novo arquivo de script foi adicionado e outro quando esse arquivo foi alterado.

```
Rajat@DESKTOP-J8MKHE6 MINGW64 ~/Desktop/my-project (master)
$ git stash list
stash@{0}: WIP on master: 94e3760 alter the script file
stash@{1}: WIP on master: a628e30 add a script file
```

Note que o stash mais recente sempre tem o índice 0.

Depois de saber qual stash deseja restaurar, use o comando:

```
git stash apply n
```

Ou, alternativamente:

```
git stash apply "stash@{n}"
```

Aqui, **n** é o índice do stash que você quer restaurar.

Git Alias

Se existe um comando Git comum, mas complexo, que você digita com frequência, considere configurar um alias simples para ele. Aliases permitem fluxos de trabalho mais eficientes, exigindo menos teclas para executar um comando. É importante notar que **não existe um comando git alias direto**. Os aliases são criados usando o comando `git config` e os arquivos de configuração do Git.

```
git config --global alias.co checkout
```

Agora, quando eu uso o comando `git co`, é como se tivesse digitado o comando mais longo `git checkout`.

```
git co -b branch1
```

Saída

```
Switched to a new branch 'branch1'
```

Criar aliases não modifica os comandos originais. Então `git checkout` continuará disponível mesmo que agora tenhamos o alias `git co`.

```
git checkout -b branch2
```

Saída

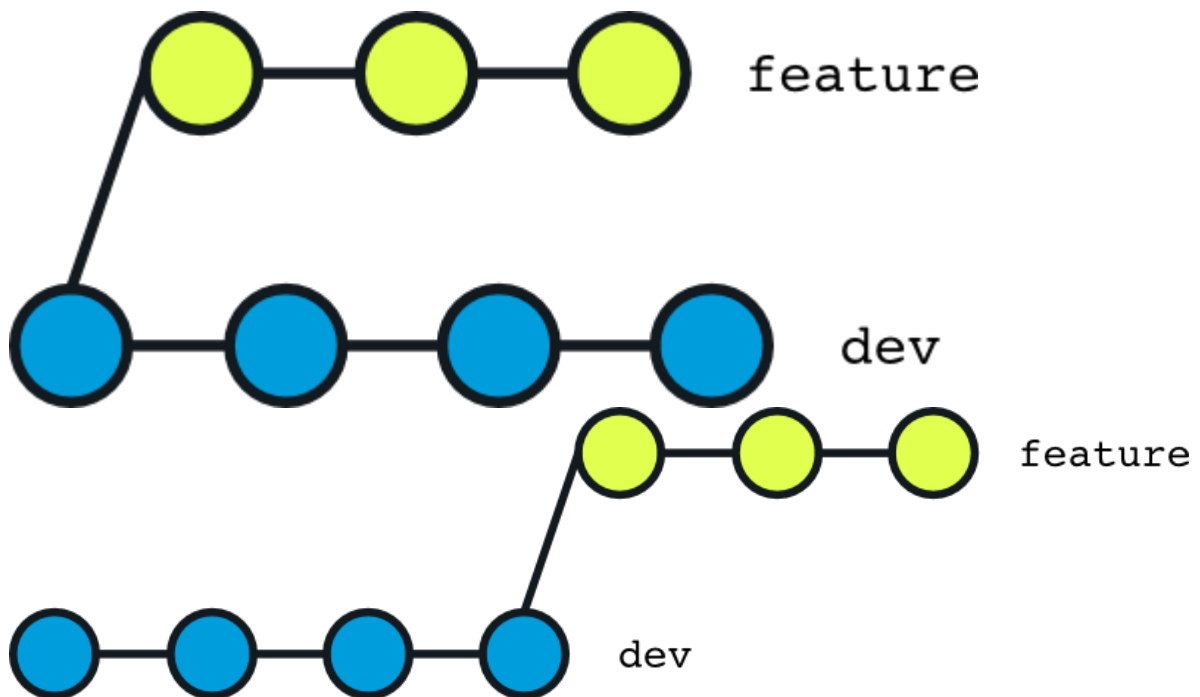
```
Switched to a new branch 'branch2'
```

Aliases também podem ser usados para agrupar uma sequência de comandos Git em um novo comando Git.

Git Rebase

O rebase é frequentemente usado como alternativa ao merge. Fazer rebase em um branch atualiza um branch com outro, aplicando os commits de um branch sobre os commits de outro branch. Por exemplo, se você está trabalhando em um branch de funcionalidade que está desatualizado em relação ao branch dev, fazer rebase do branch de funcionalidade sobre o dev permitirá que todos os novos commits do dev sejam incluídos na funcionalidade. Veja como isso fica visualmente:

Visualização do comando:



Sintaxe:

```
git rebase feature dev
```

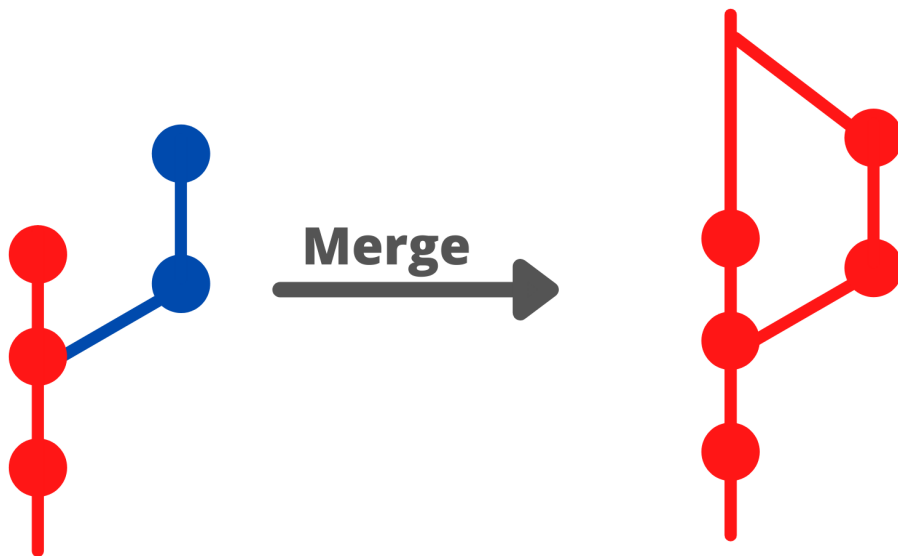
onde branch1 é o branch que queremos fazer rebase para o master.

Diferença entre Merge e Rebase:

Muitas pessoas acham que os comandos **Merge** e **Rebase** fazem o mesmo trabalho, mas na verdade são completamente diferentes e vamos discutir isso a seguir.

- **Merge:**

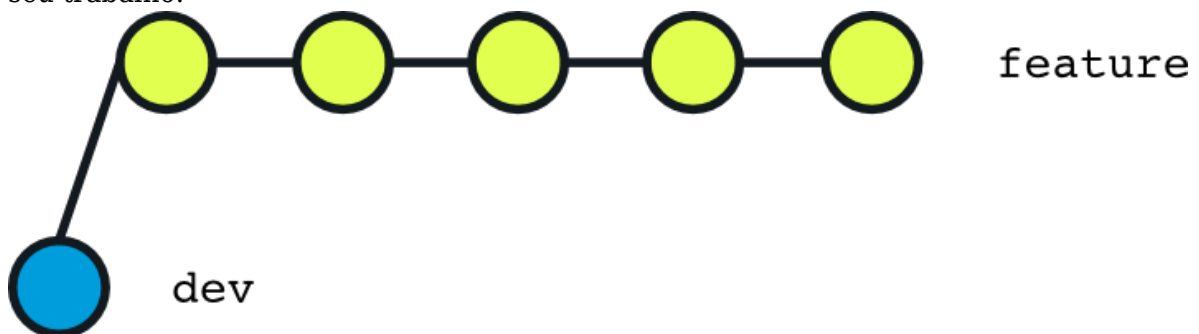
Este comando é usado para integrar alterações de um branch para outro, mantendo o branch mesclado como base, assim você pode facilmente retornar a uma versão anterior do código se quiser. A imagem abaixo mostra isso:



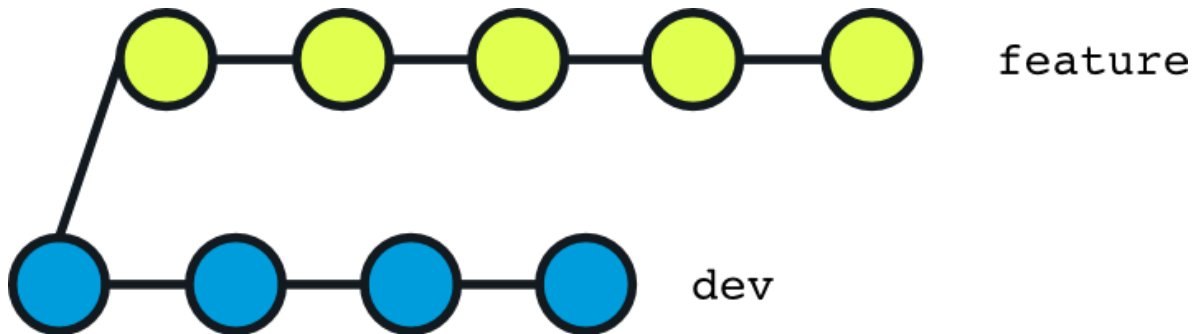
Uso típico do rebase

Atualizando um Branch de Funcionalidade

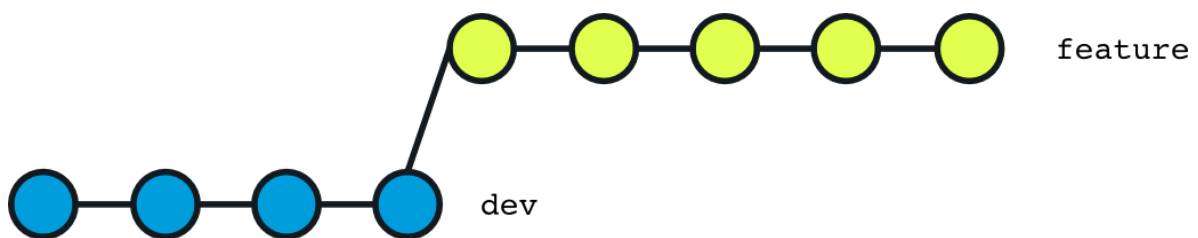
Vamos supor que você está trabalhando em um branch de funcionalidade, focado no seu trabalho.



Então você percebe que há novos commits no dev que gostaria de ter no seu branch de funcionalidade, já que esses novos commits podem afetar como você implementa a funcionalidade.



Você decide rodar `git rebase dev` a partir do seu branch de funcionalidade para ficar atualizado com o dev. No entanto, ao rodar o comando rebase, há alguns conflitos entre as alterações que você fez na funcionalidade e os novos commits no dev. Felizmente, o processo de rebase passa por cada commit um de cada vez e, assim que percebe um conflito em um commit, o git fornece uma mensagem no terminal indicando quais arquivos precisam ser resolvidos. Depois de resolver o conflito, você usa `git add` para adicionar suas alterações ao commit e roda `git rebase --continue` para continuar o processo de rebase. Se não houver mais conflitos, você terá feito o rebase do seu branch de funcionalidade sobre o dev com sucesso.



Agora você pode continuar trabalhando na sua funcionalidade com os commits mais recentes do dev incluídos, e tudo está bem novamente. Esse processo pode se repetir se o branch dev for atualizado com novos commits.

- **Rebase:**

Por outro lado, o comando **Rebase** é usado para transferir a base do branch para ser baseada no último commit do branch atual, tornando-os como um só branch, como mostrado na imagem acima.

Rebase interativo:

Você também pode fazer rebase de um branch sobre outro de forma **interativa**. Isso significa que você será solicitado a escolher opções. O comando básico é:

```
git rebase -i feature main
```

Isso abrirá seu editor favorito (provavelmente **vi** ou **vscode**).

Vamos criar um exemplo:

```
git switch main
git checkout -b feature-interactive
echo "<p>Rebasing interactively is super cool</p>" >>
feature1.html
git commit -am "Commit to test Interactive Rebase"
echo "<p>With interactive rebasing you can do really cool
stuff</p>" >> feature1.html
git commit -am "Commit to test Interactive Rebase"
```

Agora você está no branch **feature-interactive** com um commit que não existe no branch **main**, e há um novo commit no **main** que não está no seu branch.

Agora usando o comando de rebase interativo:

```
git rebase -i main
```

Seu editor favorito (**vi** aqui ou **vscode** se configurado corretamente) deve abrir algo assim:

```

pick a21b178 Commit to test Interactive Rebase
pick cd3400c Commit to test Interactive Rebase

# Rebase 1d152d4..a21b178 onto 1d152d4
#
# p, pick <commit> = usar o commit
# r, reword <commit> = usar o commit, mas editar a mensagem do
commit
# e, edit <commit> = usar o commit, mas parar para alterar
# s, squash <commit> = usar o commit, mas mesclar ao commit
anterior
# f, fixup <commit> = como "squash", mas descarta a mensagem
do commit
# x, exec <command> = rodar comando (restante da linha) usando
shell
# b, break = parar aqui (continuar rebase depois com 'git
rebase --continue')
# d, drop <commit> = remover commit
# l, label <label> = marcar HEAD atual com um nome
# t, reset <label> = resetar HEAD para um label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .      criar um commit de merge usando a mensagem do commit
original
# .      (ou o oneline, se não houver commit original).
# .      Use -c <commit> para editar a mensagem do commit.
#
# Essas linhas podem ser reordenadas; são executadas de cima
para baixo.
#
# Se você remover uma linha aqui, ESSE COMMIT SERÁ PERDIDO.
#
# Se remover tudo, o rebase será abortado.
#
# Commits vazios são comentados

```

Como pode ver, as primeiras linhas são os commits feitos nesse branch **feature-interactive**. O restante do arquivo é a mensagem de ajuda. Se olhar atentamente para essa mensagem de ajuda, verá que há várias opções. Para usá-las, basta prefixar a linha do commit que deseja trabalhar com o nome do comando ou sua letra de atalho.

O comando básico é **pick**, que significa que o rebase atual usará esses dois commits para fazer seu trabalho.

No nosso caso, se quiser atualizar a mensagem do segundo commit, basta atualizar o

arquivo assim (mostramos só as primeiras linhas):

```
pick a21b178 Commit to test Interactive Rebase
r cd3400c Commit to test Interactive Rebase

# Rebase 1d152d4..a21b178 onto 1d152d4
#
...
```

Depois salve o arquivo, se for **vi**, pressione **:** e digite **wq**. Você verá outro arquivo aberto no mesmo editor, onde pode editar sua mensagem de commit, salvar e pronto.

Você pode ver o resultado com o comando:

```
git log
```

Agora que você sabe o básico de como usar esse comando, dê uma olhada na mensagem de ajuda. Existem comandos úteis ali. Meus favoritos são **reword**, **fixup** e **drop**, mas sinta-se livre para experimentar por conta própria.

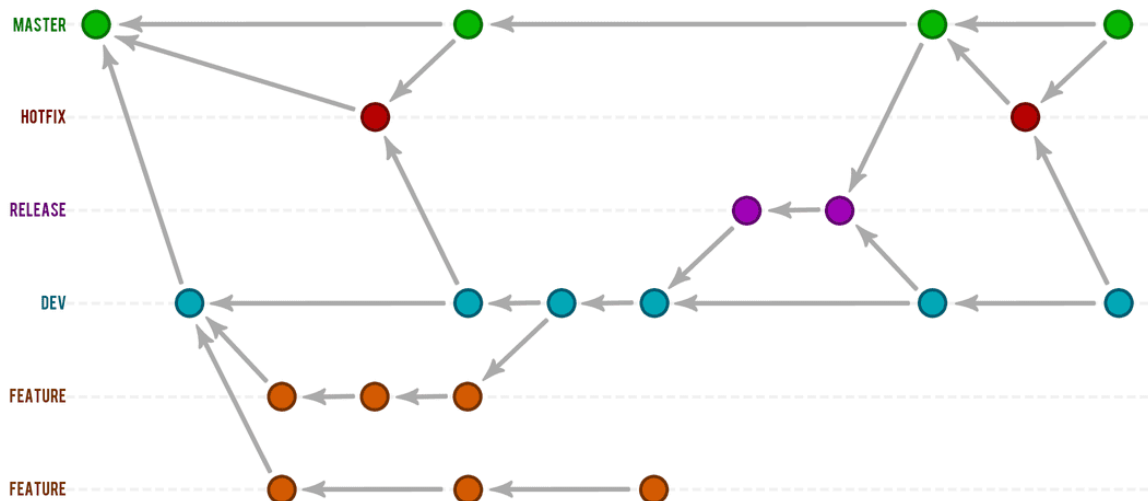
Git Switch

`git switch` não é uma funcionalidade nova, mas um comando adicional para alternar/trocar de branch, recurso que já estava disponível no comando sobrecarregado `git checkout`. Por isso, recentemente a comunidade Git decidiu publicar um novo comando: `git switch`. Como o nome sugere, ele serve para alternar branches.

Sintaxe:

```
git switch <nome-do-branch>
```

Visualização do comando:



Diferença entre Switch e Checkout:

Como você pode ver, o uso é bem direto e semelhante ao "git checkout". Mas a grande vantagem sobre o comando "checkout" é que "switch" NÃO tem um milhão de outros significados e capacidades.

Como é um membro relativamente novo da família de comandos do Git, você deve verificar se sua instalação do Git já inclui esse comando.

Alternar entre dois branches repetidamente

Em alguns cenários, pode ser necessário alternar entre dois branches várias vezes. Em vez de sempre digitar os nomes completos dos branches, você pode simplesmente usar o seguinte atalho:

Sintaxe:

```
git switch -
```

Usando o caractere traço como único parâmetro, o comando "git switch" fará checkout do branch anteriormente ativo. Como dito, isso pode ser muito útil se você precisa ir e voltar entre dois branches várias vezes.

Guia Rápido de Markdown do GitHub

Título 1

Marcação : `# Título 1 #`

-OU-

Marcação : `=====` (abaixo do texto do Título 1)

Título 2

Marcação : `## Título 2 ##`

-OU-

Marcação: `-----` (abaixo do texto do Título 2)

Título 3

Marcação : `### Título 3 ###`

Título 4

Marcação : `#### Título 4 ####`

Texto comum

Marcação : Texto comum

Texto enfatizado

Marcação : _Texto enfatizado_ ou *Texto enfatizado*

~~Texto riscado~~

Marcação : ~~Texto riscado~~

Texto forte

Marcação : __Texto forte__ ou **Texto forte**

Texto forte enfatizado

Marcação : ____Texto forte enfatizado____ ou ***Texto forte enfatizado***

Link nomeado e **http://www.google.fr/** ou **http://example.com/**

Marcação : [Link nomeado](http://www.google.fr/ "Título do link nomeado") e http://www.google.fr/ ou <http://example.com/>

título-1

Marcação: [título-1](#titulo-1 "Ir para título-1")

Tabela, como esta:

Primeiro Cabeçalho Segundo Cabeçalho

Célula de Conteúdo Célula de Conteúdo

Célula de Conteúdo Célula de Conteúdo

Primeiro Cabeçalho	Segundo Cabeçalho
Célula de Conteúdo	Célula de Conteúdo
Célula de Conteúdo	Célula de Conteúdo

Adicionando uma barra vertical | em uma célula:

Primeiro Cabeçalho Segundo Cabeçalho

Célula de Conteúdo Célula de Conteúdo

Célula de Conteúdo |

Primeiro Cabeçalho	Segundo Cabeçalho
Célula de Conteúdo	Célula de Conteúdo
Célula de Conteúdo	\

Tabela alinhada à esquerda, direita e centro

Cabeçalho Esquerda Cabeçalho Direita Cabeçalho Centro

Célula de Conteúdo Célula de Conteúdo Célula de Conteúdo

Célula de Conteúdo Célula de Conteúdo Célula de Conteúdo

Cabeçalho Esquerda	Cabeçalho Direita	Cabeçalho Centro
:---	---:	:---:
Célula de Conteúdo	Célula de Conteúdo	Célula de Conteúdo
Célula de Conteúdo	Célula de Conteúdo	Célula de Conteúdo

code()

Marcação : ``code()``

```
var codigo_especifico =
{
  "data": {
    "lookedUpPlatform": 1,
    "query": "Kasabian+Test+Transmission",
    "lookedUpItem": {
      "name": "Test Transmission",
      "artist": "Kasabian",
      "album": "Kasabian",
      "picture": null,
      "link":
"http://open.spotify.com/track/5jhJur5n4fasbLLSC0crTp"
    }
  }
}
```

Marcação : ````javascript`
`````

## Lista não ordenada

- Item da lista
  - Subitem
    - Sub-subitem etc
- Item da lista 2

```
Marcação : * Item da lista
 * Subitem
 * Sub-subitem etc
 * Item da lista 2
-OU-
Marcação : - Item da lista
 - Subitem
 - Sub-subitem etc
 - Item da lista 2
```

## Lista ordenada

1. Lista numerada
  1. Lista numerada aninhada
  2. Que é numerada
2. Que é numerada

```
Marcação : 1. Lista numerada
 1. Lista numerada aninhada
 2. Que é numerada
2. Que é numerada
```

- [ ] Uma tarefa não concluída
- [x] Uma tarefa concluída

```
Marcação : - [] Uma tarefa não concluída
 - [x] Uma tarefa concluída
```

- [ ] Uma tarefa não concluída
  - [ ] Uma sub-tarefa

```
Marcação : - [] Uma tarefa não concluída
 - [] Uma sub-tarefa
```

## Citação

Citação aninhada Marcação : > Citação >> Citação aninhada

## ***Linha horizontal :***

---

Marcação : - - - -

## ***Imagem com alt :***



Marcação : ![texto alternativo](http://via.placeholder.com/200x150 "Título opcional")

## **Texto recolhível:**

Título 1

Conteúdo 1 Conteúdo 1 Conteúdo 1 Conteúdo 1 Conteúdo 1

Título 2

Conteúdo 2 Conteúdo 2 Conteúdo 2 Conteúdo 2 Conteúdo 2

Marcação : <details>  
    <summary>Título 1</summary>  
    <p>Conteúdo 1 Conteúdo 1 Conteúdo 1 Conteúdo 1  
Conteúdo 1</p>  
    </details>

<h3>HTML</h3>  
<p> Algum código HTML aqui </p>

Link para uma parte específica da página:

## Ir para o TOPO

```
Marcação : [texto aqui](#nome_da_secao)
 titulo_secao
```

## Tecla de atalho:

⌘F

⇧⌘F

```
Marcação : <kbd>⌘F</kbd>
```

## Lista de teclas de atalho:

| Tecla     | Símbolo |
|-----------|---------|
| Option    | ⌘       |
| Control   | ⌃       |
| Command   | ⌘       |
| Shift     | ⇧       |
| Caps Lock | ⇧       |
| Tab       | ⇧       |
| Esc       | ⌫       |
| Power     | ⏻       |
| Return    | ↵       |
| Delete    | ⌫       |
| Cima      | ↑       |
| Baixo     | ↓       |
| Esquerda  | ←       |
| Direita   | →       |

## Emoji:

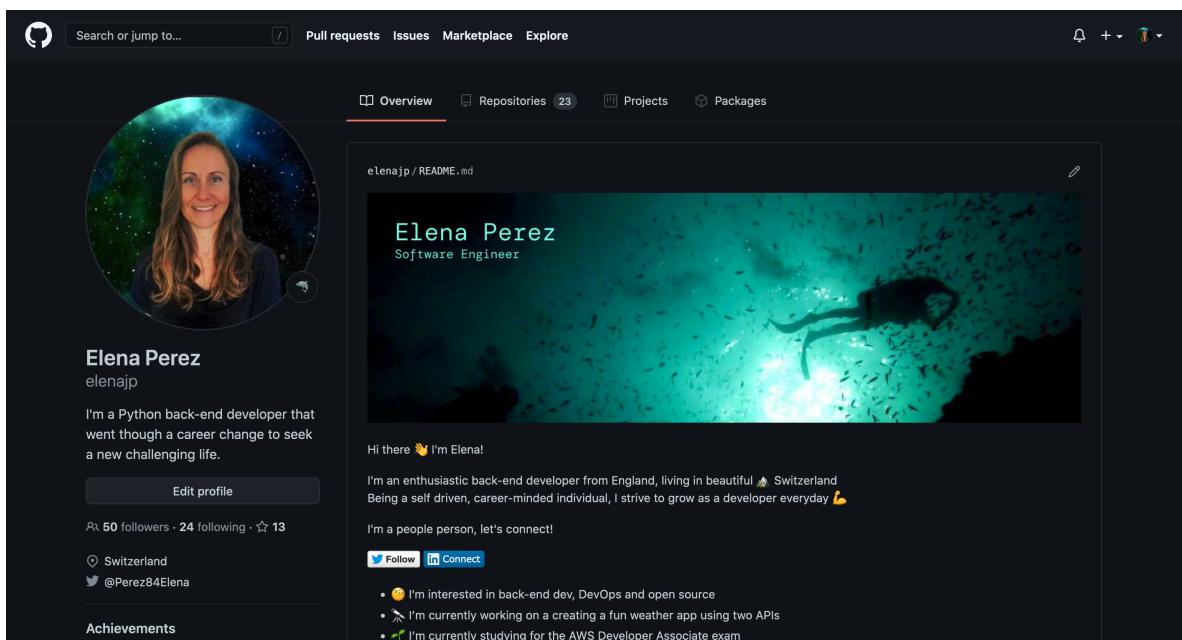
:exclamation: Use ícones de emoji para destacar o texto. :+1: Veja os códigos de emoji em [emoji-cheat-sheet.com](https://emoji-cheat-sheet.com)

Marcação : 0 código aparece entre dois pontos :CODIGOEMOJI:



# Crie seu perfil no GitHub

Além de parecer incrível, um bom perfil no GitHub mostra às pessoas o que você sabe fazer e te dá a chance de mostrar isso. Além disso, candidaturas de emprego geralmente incluem um campo para adicionar o link do seu perfil no GitHub, então essa pode ser sua chance de brilhar ☑



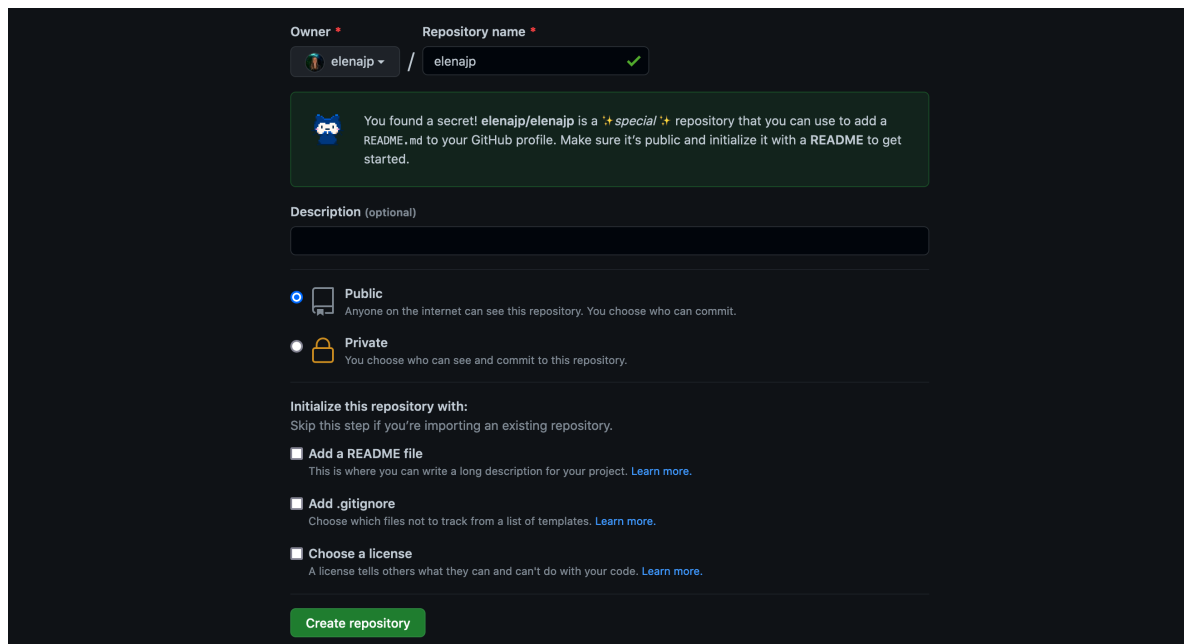
[Ver perfil completo](#)

## Markdown

Para criar um perfil no GitHub, você precisa entender Markdown. Markdown é uma linguagem de marcação leve para criar texto formatado usando um editor de texto simples. Confira este [Guia rápido de Markdown](#).

## Crie um repositório GitHub para sua página de perfil

Primeiro, você precisa criar um novo repositório. Use este link <https://github.com/new> para fazer isso. Você deve nomear seu repositório exatamente igual ao seu nome de usuário do GitHub, como mostrado abaixo:



Você irá descobrir seu repositório secreto/especial! Torne o repositório público e selecione adicionar um arquivo README. Pronto, você criou seu repositório. Recomendo deixar o repositório privado enquanto trabalha nele e torná-lo público quando estiver pronto. Agora é hora de adicionar conteúdo e deixá-lo incrível.

**Dica:** Ao trabalhar em um arquivo markdown no seu ambiente de desenvolvimento, você pode visualizar como a página ficará pressionando **CMD+Shift+V** ou **Ctrl+Shift+V**

## Adicionando um cabeçalho

A primeira coisa que as pessoas notam no seu perfil é o cabeçalho. Talvez você possa adicionar uma foto de algo que você ama, seu hobby, arte que te inspira, uma imagem de algum código que você fez, etc. Você pode até combinar com sua foto de perfil. Recomendo uma foto real sua, em vez de um desenho. O markdown necessário é:

```
![Repository Banner](https://raw.githubusercontent.com/GITHUB-
USERNAME/GITHUB-USERNAME/banner_photo.png)
```

Aqui está um site que permite gerar imagens de cabeçalho para o README do seu perfil no GitHub: <https://reheader.glitch.me/>

## Apresente-se

Cumprimente seus visitantes. Escreva um parágrafo introdutório. Fale um pouco sobre

you, where you are, what inspires you. This section is where you can let your personality shine.

## Say what you're doing

You can show visitors to your profile what you're doing and what you're working on. It's a great way for them to learn more about you.

- ☐ I'm interested in...
- ☐ I'm currently working on...
- ☐ I'm currently studying...
- ☐ I'm looking for...
- ☐ I'm looking for help with...
- ☐ Ask me about...
- ☐ Curiosity:...

## Include statistics

You can display your GitHub statistics on your profile. This includes showing total commits, total pull requests, total issues, etc. Check out these themes for displaying your statistics:



[Click here](#) to use one of these themes

## Adicione botões de redes sociais

Adicionar botões de redes sociais é uma ótima forma de direcionar visitantes ou até empregadores para suas plataformas sociais. Por exemplo, para adicionar o Twitter:



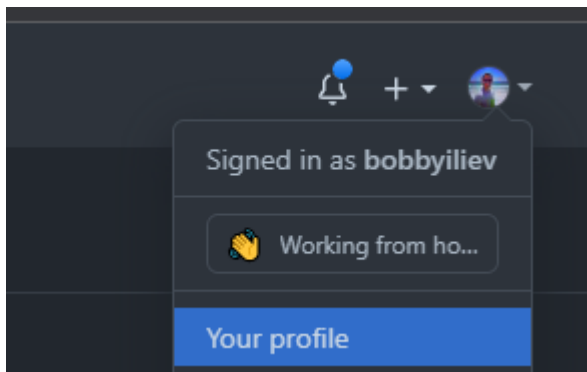
```
[![Twitter
URL](https://img.shields.io/twitter/url/https/twitter.com/USER
-
NAME.svg?style=social&label=Twitter)](https://twitter.com/USER
-NAME)
```

## Fixe repositórios no seu perfil do GitHub

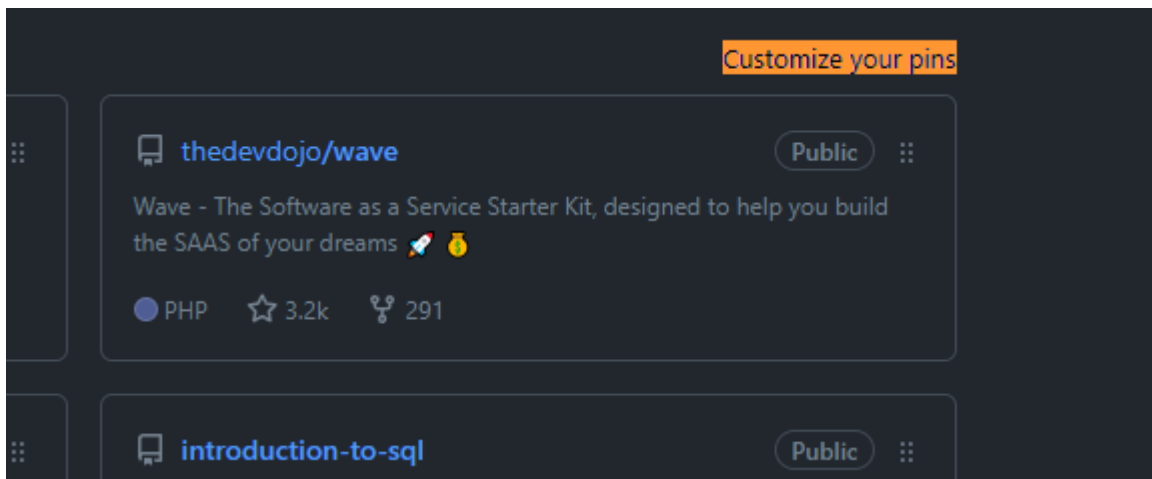
À medida que o número de repositórios que você criou ou contribuiu cresce, talvez queira destacar alguns deles diretamente no seu perfil. Esse recurso é útil para apresentar seu portfólio a um possível empregador e mostrar primeiro os trabalhos dos quais você mais se orgulha!

Veja como fazer:

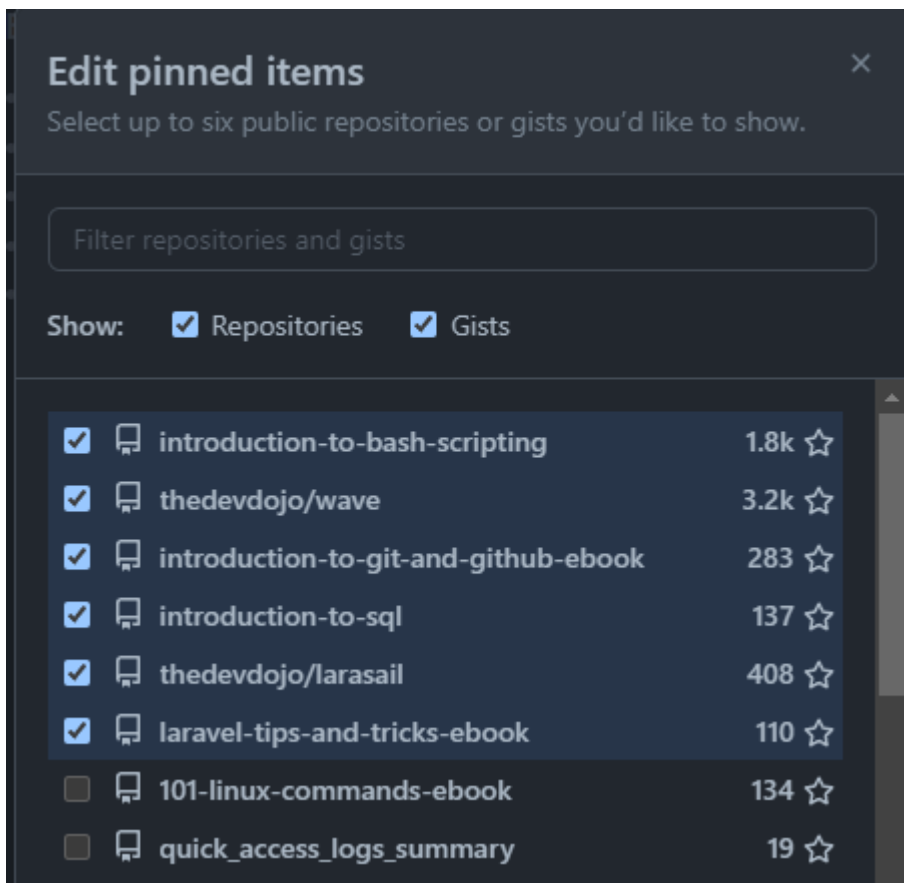
No canto superior direito, clique na sua foto de perfil e selecione "Seu perfil" no menu:



Na seção "Popular repositories" ou "Pinned", clique em "Customize your pins":



Depois disso, a janela "Edit pinned items" será aberta. Aqui, você pode escolher até seis repositórios e/ou gists para exibir:



Depois de decidir quais repositórios quer fixar, clique em "Save pins".

## Inspire-se

Confira estes links úteis para se inspirar e ajudar com seu perfil do GitHub:

- <https://github.com/abhisheknaidu/awesome-github-profile-readme>
- <https://github.com/elangosundar/awesome-README-templates>

## Seja criativo

Esse é um projeto divertido para trabalhar. Seja criativo! Há muito mais que você pode adicionar ao seu perfil. Desde diferentes estatísticas, tecnologias e ferramentas que você usa, GIFs de emoji, etc. Você pode pedir opiniões de amigos ou outros desenvolvedores sobre seu perfil. Lembre-se de não exagerar. Ninguém quer animações rápidas demais ou imagens piscando que machucam os olhos. Pense em temas de cores agradáveis, o que combina bem, e adicione uma boa foto de perfil. Boa sorte!

# Guia Rápido de Comandos Git

Aqui está uma lista dos comandos Git mencionados ao longo do eBook

- Configuração do Git

Antes de inicializar um novo repositório git ou começar a fazer commits, você deve configurar sua identidade no git.

Para alterar o nome associado aos seus commits, use o comando `git config`:

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

O mesmo vale para alterar seu endereço de e-mail associado aos commits:

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

Assim, quando você fizer um commit e depois verificar o log do git, verá que o commit está associado aos detalhes que você configurou acima.

```
git log
```

No meu caso, a saída é assim:

```
commit 45f96b8c2ef143011f11b5f6cc7a3ae20db5349d (HEAD -> main,
origin/master, origin/HEAD)
Author: Bobby Iliev <bobby@bobbyiliev.com>
Date: Fri Jun 19 17:03:53 2020 +0300

 Nginx server name for www version (#26)
```

## Inicializando um projeto

Para inicializar um novo projeto git local, abra seu terminal git ou bash, use **cd** para ir até o diretório onde deseja armazenar seu projeto e então execute:

```
git init .
```

Se você já tem um projeto existente no GitHub, por exemplo, pode cloná-lo usando o comando git clone:

```
git clone url_do_seu_projeto
```

## Status atual

Para verificar o status atual do seu repositório git local, use o seguinte comando:

```
git status
```

Esse é provavelmente um dos comandos mais usados, pois você precisa verificar o status do seu repositório local com frequência para saber quais arquivos foram alterados, preparados ou excluídos.

## Adicionar um arquivo à área de staging

Suponha que você tem um projeto HTML estático e já inicializou seu repositório git.

Depois, em algum momento, decide adicionar um novo arquivo HTML chamado **about-me.html**, já com algum código HTML. Para adicionar esse novo arquivo para que ele seja rastreado pelo git, primeiro use o comando **git add**:

```
git add nome_do_arquivo
```

Isso irá preparar seu novo arquivo, o que significa que na próxima vez que você fizer um commit, a alteração fará parte do commit.



Para verificar, você pode rodar novamente o comando `git status`:

```
git status
```

Você verá a seguinte saída:

```
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)

 new file: about-me.html
```

## Remover arquivos

Para remover um arquivo do seu projeto git, use o seguinte comando:

```
git rm algum_arquivo.txt
```

Depois disso, se rodar `git status` novamente, verá que o arquivo `algum_arquivo.txt` foi excluído:

```
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)

 deleted: algum_arquivo.txt
```

## Descartar alterações de um arquivo

Caso tenha cometido um erro e queira descartar as alterações de um arquivo específico e restaurar o conteúdo desse arquivo como estava no último commit, use o comando abaixo:

```
git checkout -- nome_do_arquivo
```

Esse comando é conveniente para reverter rapidamente um arquivo ao seu conteúdo original.

## Commit local

Depois de fazer suas alterações e prepará-las com o comando `git add`, você precisa fazer o commit das alterações.

Para isso, use o comando `git commit`:

```
git commit
```

Isso abrirá um editor de texto onde você pode digitar sua mensagem de commit.

Ou, você pode usar a flag `-m` para especificar a mensagem de commit diretamente no comando:

```
git commit -m "Mensagem legal de commit aqui"
```

## Listar branches

Para listar todos os branches locais disponíveis, basta rodar o seguinte comando:

```
git branch -a
```

Você verá uma lista de branches locais e remotos, a saída será assim:

```
bugfix/nginx-www-server-name
develop
* main
remotes/origin/HEAD -> origin/master
remotes/origin/bugfix/nginx-www-server-name
remotes/origin/develop
remotes/origin/main
```

A palavra **remotes** indica que esses branches são remotos.

## Buscar alterações do remoto e mesclar o branch atual com o upstream

Se você está trabalhando com uma equipe de desenvolvedores no mesmo projeto, muitas vezes precisará buscar as alterações que seus colegas fizeram para tê-las localmente no seu PC.

Para isso, basta usar o comando **git pull**:

```
git pull origin nome_do_branch
```

Note que isso também irá mesclar as novas alterações ao branch atual em que você está.

## Criar um novo branch

Para criar um novo branch, basta usar o comando **git branch**:

```
git branch nome_do_branch
```

Em vez disso, prefiro usar o comando abaixo, pois ele cria um novo branch e também muda para o branch recém-criado:

```
git checkout -b nome_do_branch
```

Se o `nome_do_branch` já existir, você receberá um aviso de que o branch existe e não será mudado para ele.

## Enviar alterações locais para o remoto

Por fim, depois de fazer todas as suas alterações, prepará-las com o comando `git add .` e fazer o commit com o comando `git commit`, você precisa enviar essas alterações para o repositório git remoto.

Para isso, use o comando `git push`:

```
git push origin nome_do_branch
```

## Excluir um branch

```
git branch -d nome_do_branch
```

## Mudar para um novo branch

```
git checkout nome_do_branch
```

Como mencionado acima, se você adicionar a flag `-b`, o branch será criado se não existir.

## Conclusão

Conhecendo os comandos acima, você poderá gerenciar seu projeto como um profissional!

Se quiser melhorar suas habilidades no terminal em geral, recomendo este [curso básico de linha de comando Linux](#)!

# Conclusão

Parabéns! Você acaba de concluir o guia básico de Git!

Se achou útil, não deixe de dar uma estrela no projeto no [GitHub](#)!

Se tiver sugestões de melhorias, contribua com pull requests ou abra issues.

Neste eBook introdutório sobre Git e GitHub, cobrimos apenas o básico, mas você já tem conhecimento suficiente para começar a usar o Git e contribuir para projetos open source incríveis!

Como próximo passo, tente criar um projeto no GitHub, cloná-lo localmente e enviar um projeto em que você está trabalhando para o GitHub! Recomendo também este [treinamento do GitHub aqui](#).

Caso este eBook tenha te inspirado a contribuir com algum projeto open-source, não deixe de tuitar sobre isso e marcar [@bobbyiliev](#) para que possamos conferir!

Parabéns novamente por concluir este eBook!