GIT & GITHUB

Do Básico ao Avançado

Domine Git e GitHub com este material completo. Com ele você aprenderá a gerenciar fluxos de trabalho de maneira simples e dinâmica.

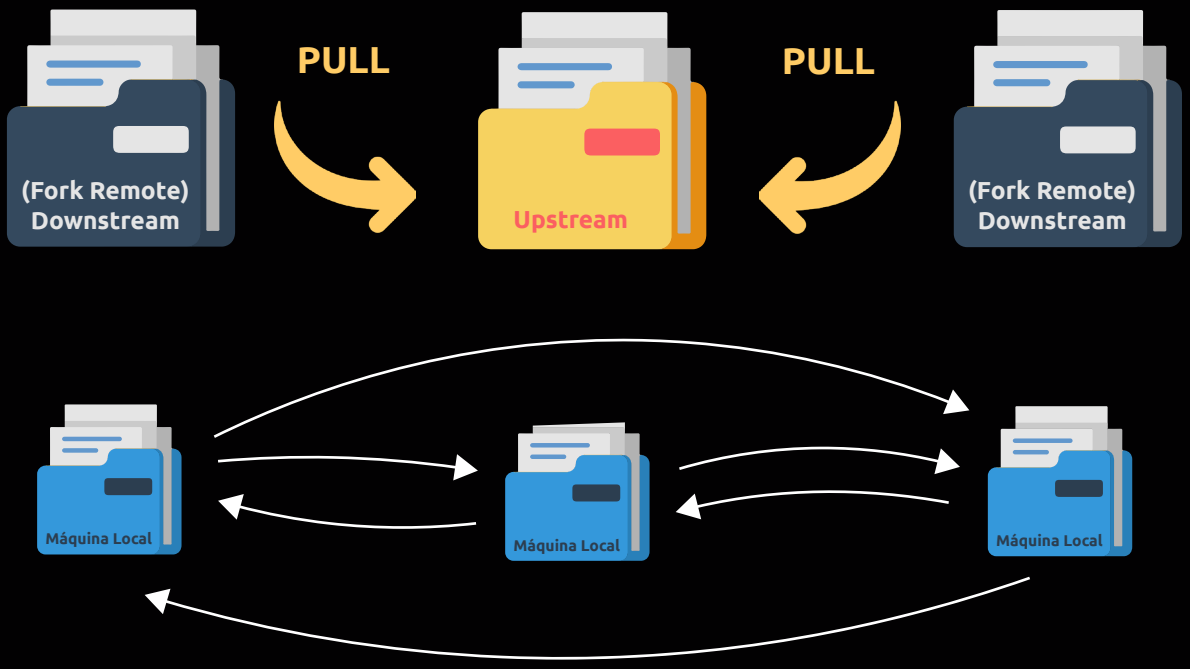
Git Workflow

No material anterior nós aprendemos um pouco como funciona o processo de colaboração entre os desenvolvedores usando o **Fork Workflow**.

Recapitulando, com ele nós começamos com uma cópia de um determinado repositório, realizamos modificações nos códigos para que mais tarde, possamos fazer um **Pull Request**, e esperar que o dono do repositório original faça um **merge**.

E o mesmo processo se aplica a cada colaborador existente naquele repositório:

Entretanto, quando existe um fluxo intenso de trabalho que está acontecendo de forma simultânea entre os colaboradores, aonde muitas vezes diversos colaboradores precisam estar a par das modificações de outros colaboradores, nós podemos ter essa pequena confusão aqui:



Imagina todos os colaboradores tendo que rastrear o Fork dos outros colaboradores, para entender o que cada um está fazendo ou deixando de fazer no mesmo projeto?

Complicado, não?

Apesar dessas complicações, geralmente os donos dos repositórios estabelecem regras de modo a **"barrar"** essas complexidades, permitindo que elas não ocorram.

Então é possível configurar certos filtros nas configurações de um determinado repositório?

Não, quando eu digo **"regras"**, digo que determinados donos de repositório criam documentos do tipo **.README** que estabelecem regras e dicas de como desenvolver e trabalhar melhor em equipe.

De modo a evitar que complicações como essas, aconteçam de fato.

Exemplo de uma possível regra criada por usuários do GitHub:

+ Regra N° 1 – Se você quer adicionar uma nova funcionalidade, faça isso sem precisar depender de outra funcionalidade ou modificação catalogada nas Issues.

Um exemplo prático, é quando temos duas Issues relacionadas a um determinado repositório.

A **primeira (Issue #1)** diz a respeito da necessidade de uma tela de login (Back-end e Front-end), e a **segunda (Issue #2)** diz que é necessário criar um banco de dados para a tela de login e cadastro de clientes.

Imaginando que o projeto tenha dois colaboradores, se o primeiro colaborador pegar a Issue #1, e o segundo a Issue #2, em algum ponto do projeto eles vão precisar sincronizar seus Forks.

Uma vez que o colaborador um vai precisar se comunicar com o banco de dados para recuperar algumas informações, do mesmo modo que o colaborador dois, vai precisar entender a lógica da tela de login para criar uma tabela otimizada, não é verdade?

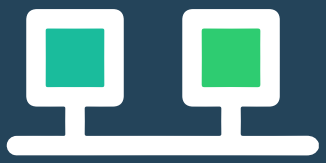
Se a **Regra N° 1** for respeitada pelos colaboradores, o colaborador que pegou a Issue #1, inevitavelmente terá que resolver também a Issue #2.

Já com a metodologia do **GitHub Workflow**, nós temos um fluxo de trabalho mais simplificado.

No Git Workflow, cada pessoa que colabora com um determinado projeto, é um desenvolvedor cadastrado dentro do repositório original, permitindo que eles façam um clone direto do repositório original, e envie alterações diretamente para ele, excluindo a necessidade de se realizar um **Pull Request**.

É como se o repositório original fosse seu, de modo que quando você fizer um **PUSH**, todos os outros colaboradores terão acesso as suas modificações com um simples **PULL**.

Para você que ainda não entendeu, é como se você criasse uma nova branch em um repositório que você tem acesso.



Feature Branch Strategy

Seguindo o fluxo de trabalho do GitHub Workflow, ficou claro que a partir do momento que você integra outros colaboradores para dentro do seu repositório, ou quem sabe, quando você é integrado como colaborador em um repositório.

Você, ou eles, tem acesso direto ao seu repositório, incluindo todas as branchs (master/main), além do histórico total do projeto, permitindo a modificação de arquivos diretamente no repositório principal.

Meio perigoso, não acha?

Por esse motivo, que os colaboradores que você escolher, devem ser de confiança, para que você evite dores de cabeça com “colaboradores” mal-intencionados.

Se tratando da **Feature Branch Strategy**, ela nada mais é do que uma metodologia que diz a respeito de que todas as novas funcionalidades devem ser trabalhadas em uma branch dedicada em vez da branch principal (master/main).

Esse encapsulamento facilita o trabalho de vários colaboradores que estão trabalhando em um mesmo recurso específico do projeto, sem perturbar a base de código principal.

Isso também significa que a branch principal nunca conterá códigos quebrados ou códigos que ainda não foram concluídos, o que é uma grande vantagem para ambientes de integração contínua.

O desenvolvimento de recursos de encapsulamento também possibilita o aproveitamento de solicitações pull, que são uma maneira de iniciar discussões em torno de uma ramificação.

Eles dão a outros desenvolvedores a oportunidade de aprovar um recurso antes que ele seja integrado ao projeto oficial.

Na **Feature Branch Strategy**, nós assumimos um novo repositório central chamado de **main**, onde ela representa a linha do tempo oficial do seu projeto.

Uma vez que **main** será a partir de agora, a única branch **"especial"** que fica responsável por armazenar vários ramos de recursos no repositório central sem nenhum problema.

Master VS Main

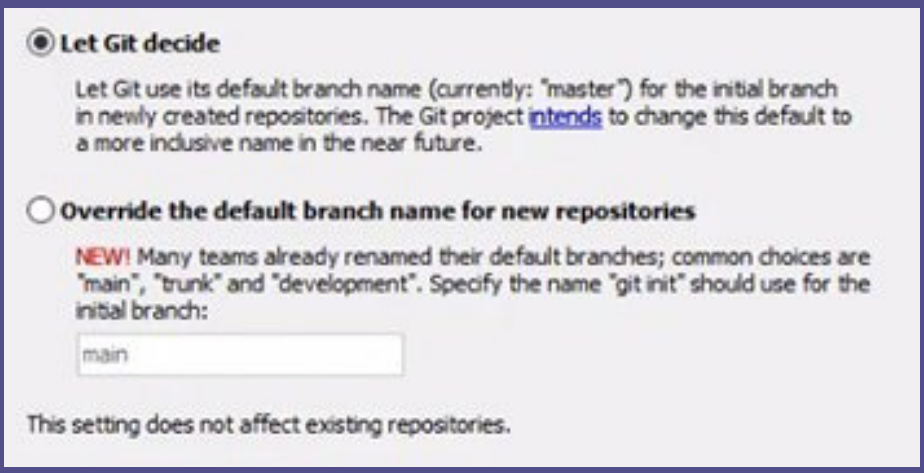
No verão de 2020, houve uma crescente onda de protestos por parte de alguns desenvolvedores com relação ao termo **"Master"**, cujo significado é **"Mestre"**, e também ao termo **"Slave"**, cujo significado é **"Escravo"**.

No mundo dos computadores, é comum encontrarmos esses dois termos **"Master e Slave"**, tanto em livros quanto também em diversas documentações de algumas linguagens de programação.

O motivo principal dos protestos é que tais termos remontavam a época da escravidão, sendo considerados termos nocivos, antiquados, e que não eram mais considerados apropriados para a sociedade atual.

Como resultado, **o GitHub renomeou o branch master para branch main**.

Tanto é que se você perceber, nas novas versões dos instaladores do Git, durante a configuração ele nós dá a possibilidade de alterar o nome da branch principal, que por padrão é **"Master"** para **"Main"**:



M D

Master VS Deploy

Em alguns repositórios do GitHub, você poderá se deparar com uma branch chamada **“Deploy”**.

Alguns donos de repositório gostam de criar essa branch, para não permitir que ninguém jogue tudo para a master/main diretamente.

Aonde todos os colaboradores jogam suas modificações para a branch deploy, e de lá eles realizam todos os testes, até terem certeza de que todo o código é seguro para ser inserido na branch master/main.

M D B

Master/Main - Depoloy - Branch

Talvez você tenha ficado um pouco confuso com essa história toda dos nomes das branchs.

Se você for uma pessoa muito organizada, você poderá trabalhar da seguinte forma:

1) O seu repositório ainda conterá a sua **branch principal**, que pode ser chamada de **Master** ou **Main**.

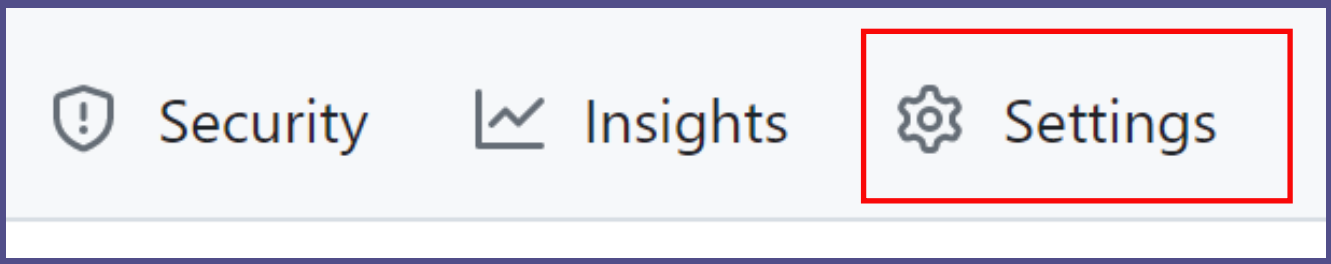
2) Abaixo dessa Branch, você poderá ter uma nova branch chamada **Deploy**, que é responsável por armazenar todas as novas modificações realizadas, para que futuramente elas sejam adicionadas a master/main.

3) Abaixo da branch deploy, ainda existirão as **branchs secundárias**, que são aquelas que criamos para resolver issues ou adicionar novas funcionalidades para depois serem sincronizadas com a branch deploy.

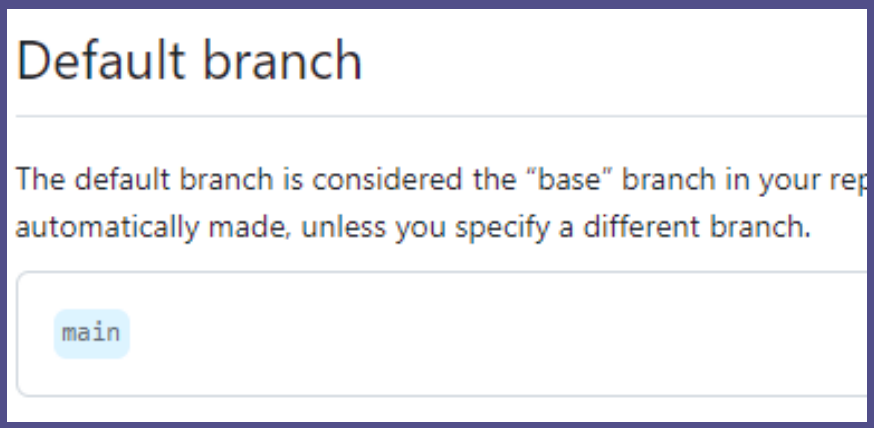
Alterando branch padrão

Caso você queria alterar a branch padrão do seu repositório do GitHub.

Entre no seu repositório e acesse a aba **Settings**:



Na seção de **Branches**, você pode modificar a branch do seu repositório para qualquer outra que existir no seu projeto:



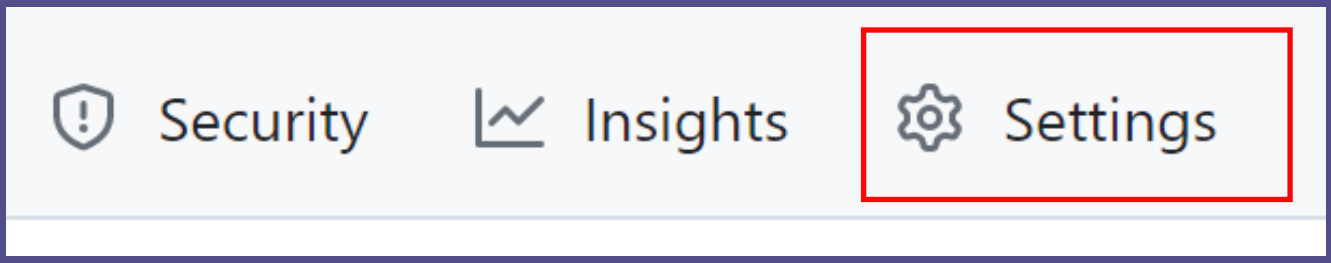
Forçando um Pull Request

Como mencionei no início desde material, você como dono de um determinado repositório, pode trazer outros contribuidores para dentro do seu repositório, desse modo, não haverá nada que os impeça de realizar modificações diretamente na branch principal.

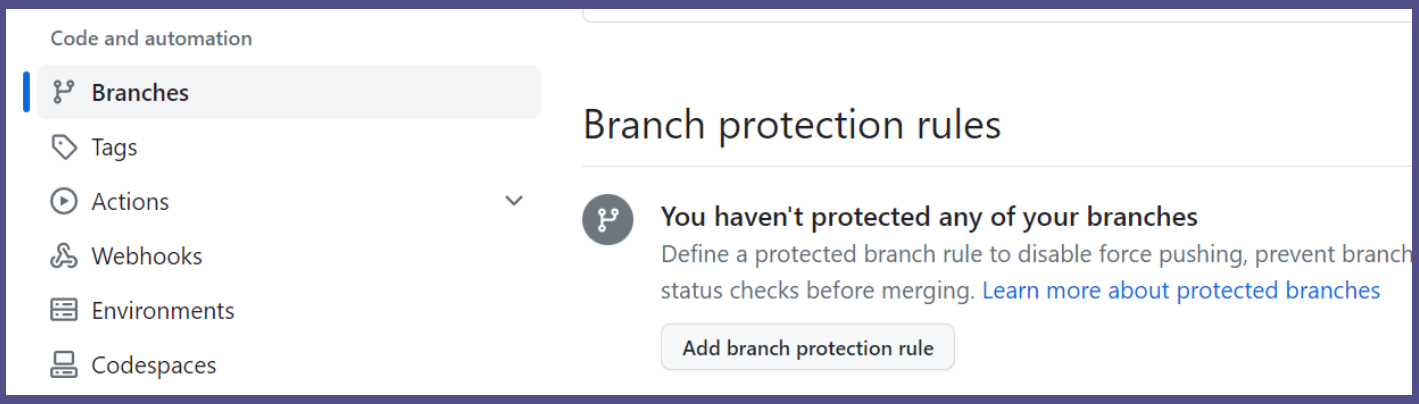
O que acarretaria na violação de alguma regra de negócio do seu projeto. E mesmo que você acredite que os contribuidores façam um esforço de boa fé de modo a nunca realizarem modificações na branch principal, quase certamente haverá momentos em que essas regras serão violadas.

Então para evitar dores de cabeça futuras, o GitHub nos dá a possibilidade de configurar filtros que impedem modificações em branchs principais de forma direta.

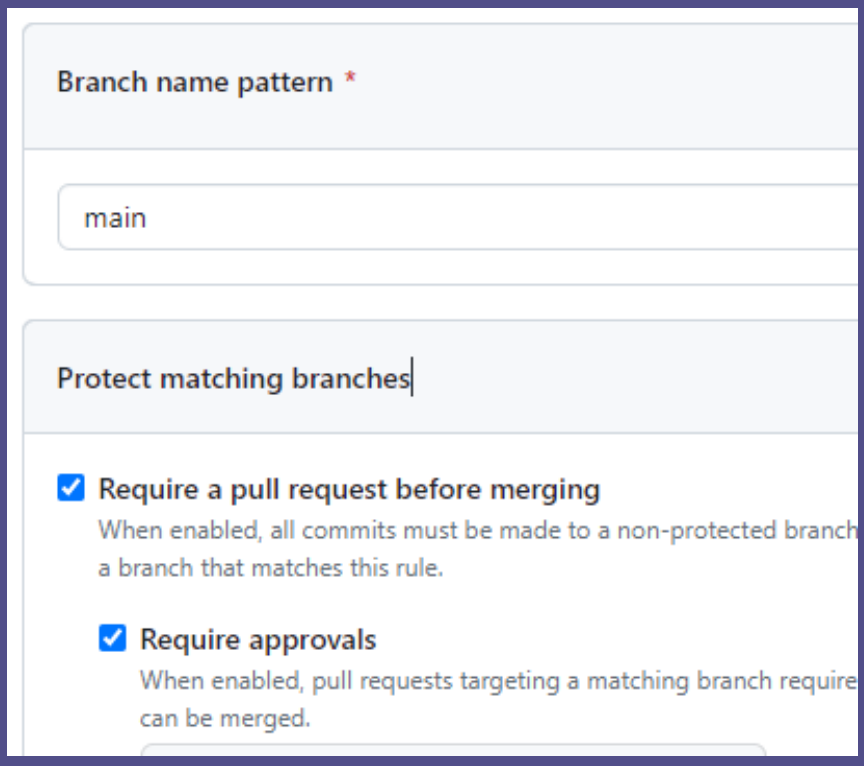
Para adicionar essa regra, entre no seu repositório, e vá em **Settings**.



Na seção de **Branches**, clique no botão **'add branch protection rule'**.



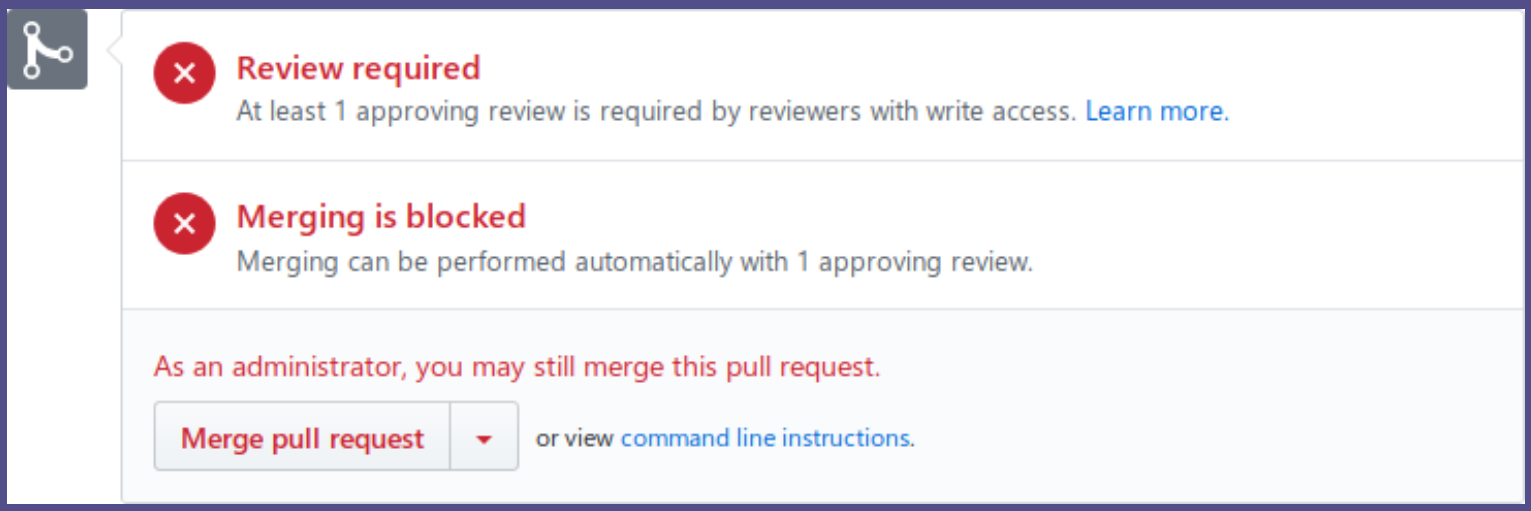
No campo relacionado a **"branch name pattern"** insira o nome da sua branch principal, e não se esqueça de ativar a opção **"Require a Pull Request Before Merging"**:



Por fim, no final da página clique no botão **"create"** para criar essa nova regra.

Com essas configurações, os colaboradores precisarão realizar um Pull Request caso eles queriam fazer um merge na branch principal.

Veja como é o comportamento do GitHub quando você é o dono do repositório e habilita a opção acima:



Ele diz a respeito que você como o administrador daquele repositório é o único responsável por realizar aquele **merge**.

Revisando o fluxo do Workflow

O fluxo do **GitHub Workflow** segue o mesmo fluxo do **Fork Workflow**, a única diferença é que não precisamos mais realizar o **Fork**.

Sendo assim, o fluxo segue essa linha de raciocínio:

1) Nos **tornamos um contribuidor** de um determinado repositório.

2) Realizamos um **clone** do repositório para nossa máquina local. **[git clone]**

3) **Criamos uma branch** específica para a funcionalidade/bug que vamos trabalhar. **[git checkout -b]**

4) Executamos um **Push do repositório local** para o remoto. **[git push]**

5) Damos início ao processo de **Pull Request** pelo GitHub.

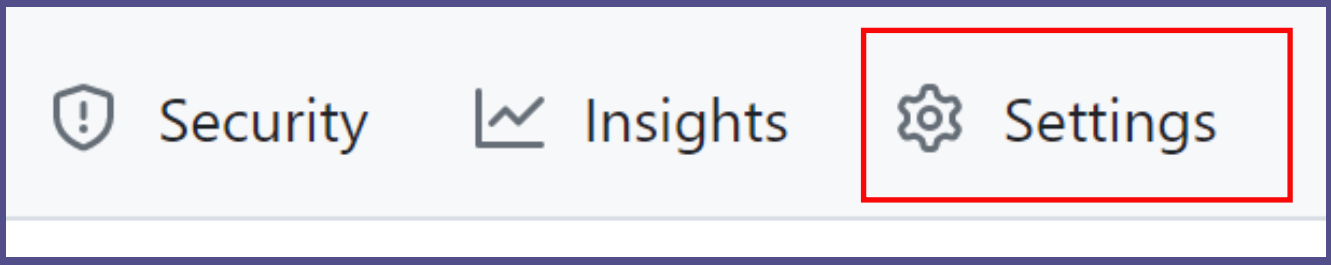
6) Por fim, quando nossa modificação for aprovada, **sincronizamos a atualização**. **[git pull]**

Adicionando contribuidores

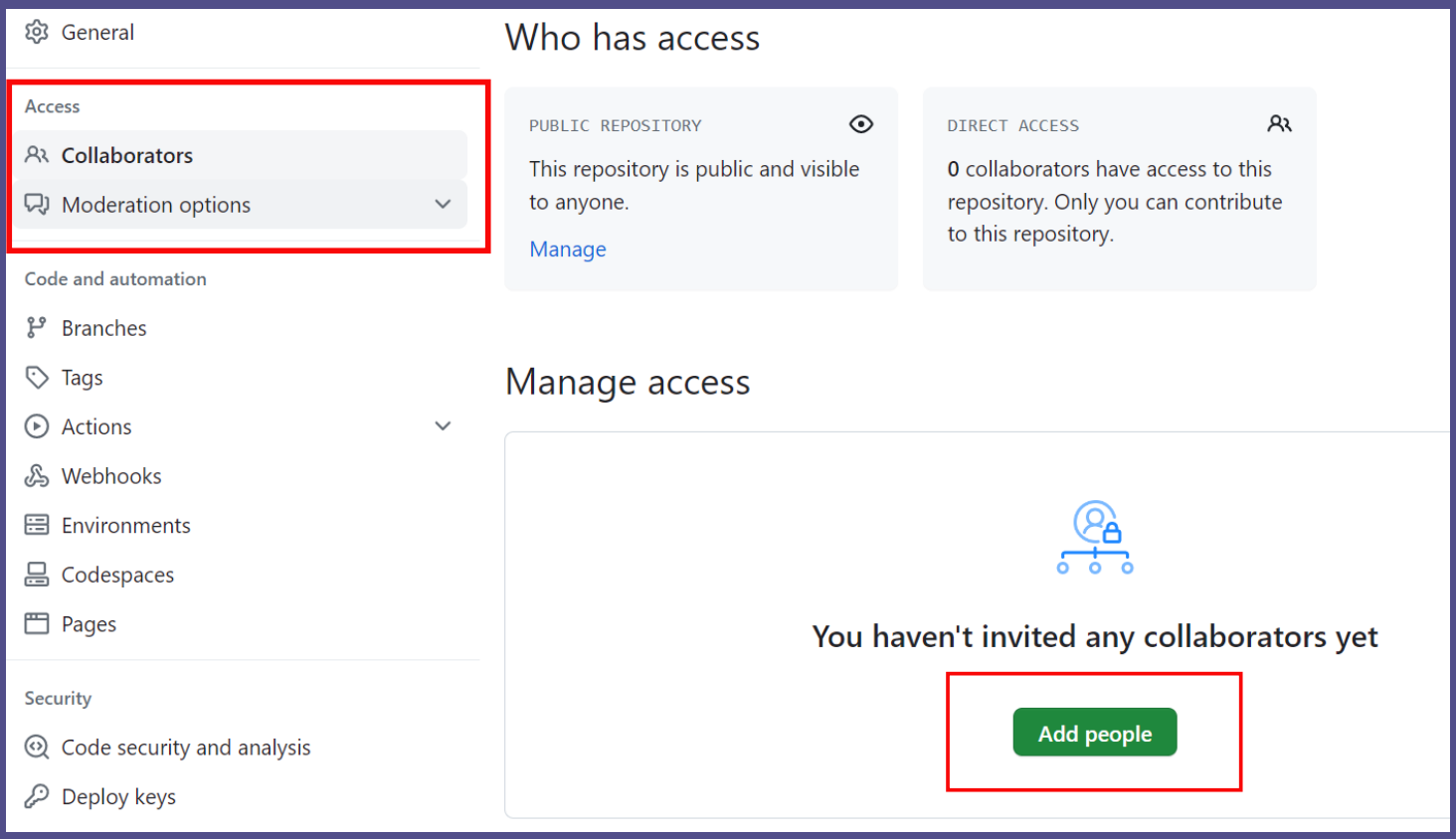
O processo para adicionar contribuidores ao seu repositório, é bem simples.

A conta gratuita do GitHub nós a possibilidade de adicionar colaboradores ilimitados em repositórios públicos e privados.

Primeiro, você precisa abrir a página inicial do repositório que você adicionar contribuidores, e entrar na aba de **Settings**:

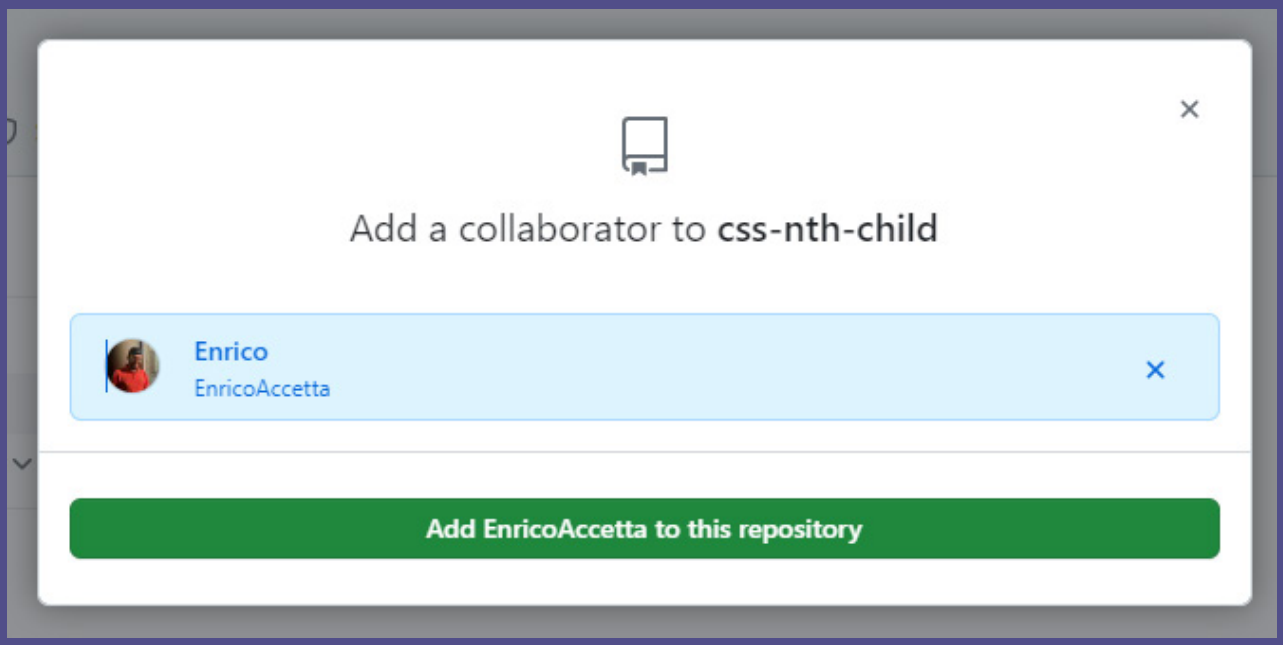


Em seguida, selecione a opção **"Collaborators"** e clique no botão **"Add People"**:

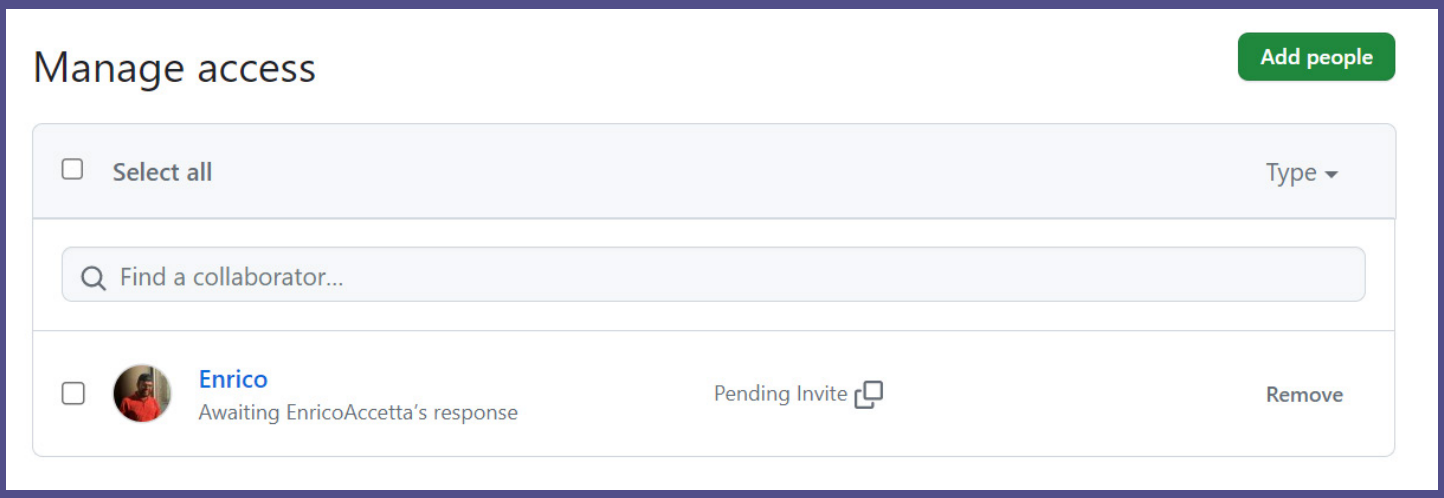


(Talvez seja necessário digitar sua senha do Github novamente para habilitar esta opção)

Um **Modal** irá se abrir, bastando apenas que você digite o nome do colaborador, ou quem sabe o e-mail dele:



Após encontrar o colaborador escolhido, clique no botão **"Add Fulano to this repository"**.



Automaticamente aquele colaborador receberá um e-mail de pedido para fazer parte do seu repositório.

Quando ele aceitar o seu pedido, ele poderá realizar as modificações no seu repositório.

Caso queira remover o colaborador, basta entrar nessa tela e clicar no botão escrito **"Remove"** que está localizado ao lado de seu nome.

Fluxo de Conflitos

Essa parte vai depender exclusivamente como você vai querer que a sua equipe de colaboradores trabalhe no projeto.

Você pode seguir a ideia do **Rebase**, quanto seguir o caminho do **Merge** (sem fazer rebase).

Lembrando que o fluxo do Rebase consegue deixar o histórico do projeto mais limpo, e o fluxo direto via merge.... Bem você sabe como funciona.

END (y/n)