Guia básico de Git

Git é um **sistema de controle de versão** que permite gerenciar o histórico do seu código fonte de maneira eficiente. Sempre que você estiver trabalhando no seu código, mudanças podem ser salvas com o Git e você pode voltar atrás para qualquer uma das versões previamente salvas. Sem o uso de ferramentas como o Git, você teria que criar cópias manuais do seu código, o que é uma tarefa complicada, não escalável e quase impossível de se manter conforme seu projeto cresce.

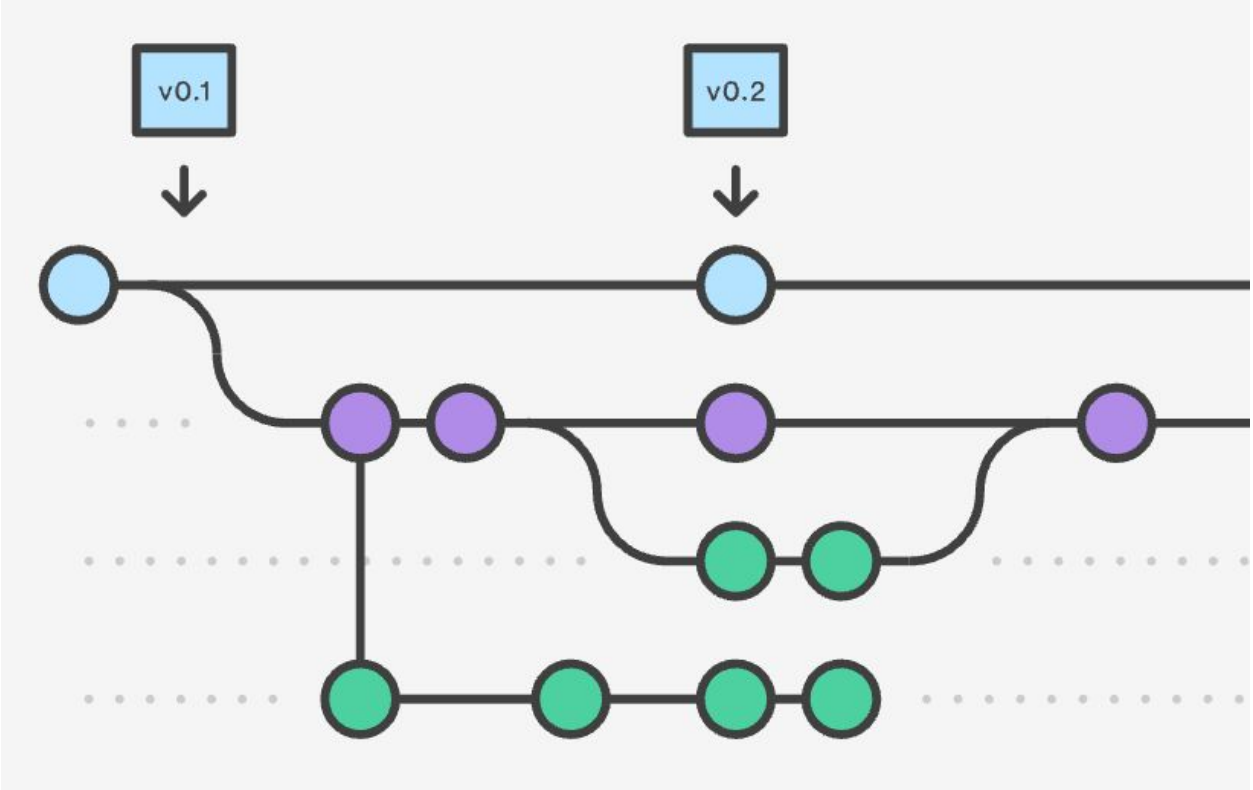
Git vs Github

Esses dois termos podem ser confusos, especialmente para iniciantes. Git é um **sistema de controle de versão** que você pode instalar em seu computador, enquanto GitHub é um **servidor remoto de colaboração** que hosteia seus projetos Git. É uma solução web que te permite fazer o upload dos seus repositórios Git. Falaremos mais sobre isso mais a frente.
Fazer o upload do seu código do seu ambiente Git local para o GitHub torna o seu código disponível para qualquer um que quiser contribuir com o seu projeto. Isso significa que qualquer pessoa pode editar meus repositórios no GitHub? Não. Quando você cria um repositório no GitHub, pode escolher se é um repositório privado, onde apenas as pessoas que você der acesso podem ver o seu código, ou se é um repositório público, em que qualquer um pode visualizar o seu código e clonar o repositório em um servidor local, porém ele só pode interferir no seu repositório hospedado no GitHub caso você dê permissão.

Repositórios, Branches e Commits

Vamos falar agora sobre três termos essenciais na linguagem do Git.
Um **repositório** é a localização onde seu código é armazenado, então é um diretório na sua máquina contendo os arquivos do projeto. Depois de transformar isso em um repositório Git (aprenderemos como fazer isso logo), o Git gerencia o histórico de versões do seu projeto.
No entanto, o código não é armazenado diretamente no repositório. Dentro de um repositório, nós temos “subdiretórios”, chamados de **branches**. Depois de adicionarmos o primeiro código ao nosso repositório, por padrão é criado um branch “master”. Porém não somos limitados a um branch. Na verdade, um dos grandes benefícios de se usar um gerenciador de controle de versão é a possibilidade de criar múltiplos branches com propósitos específicos.
Ok, esses são os repositórios e branches, mas onde nosso código é armazenado?

Dentro dos branches! Um branch contém diferentes versões do nosso código, chamados de **commits**. Cada commit é um snapshot de uma versão específica do código.



Acima há uma representação visual do que acabamos de definir. Cada um dos círculos é uma versão do software, é um commit. Cada linha é uma branch, em que a primeira linha representa a branch master. Por fim, toda essa maquinaria é o repositório.
Em poucas palavras, esses são os termos básicos que todo mundo precisa conhecer ao se usar Git. Vamos ver agora como o Git funciona na prática.

Como instalar e usar Git

O Git é gratuito e pode ser baixado aqui.
Para os usuários de Linux, basta executar `sudo apt-get install git` . Ele roda apenas no terminal (MacOS/Linux) ou no prompt de comando (Windows), isto é, ele não vem com uma interface gráfica. Certifique-se de estar um pouco familiarizado com a linha de comando do seu sistema operacional.

Comandos básicos

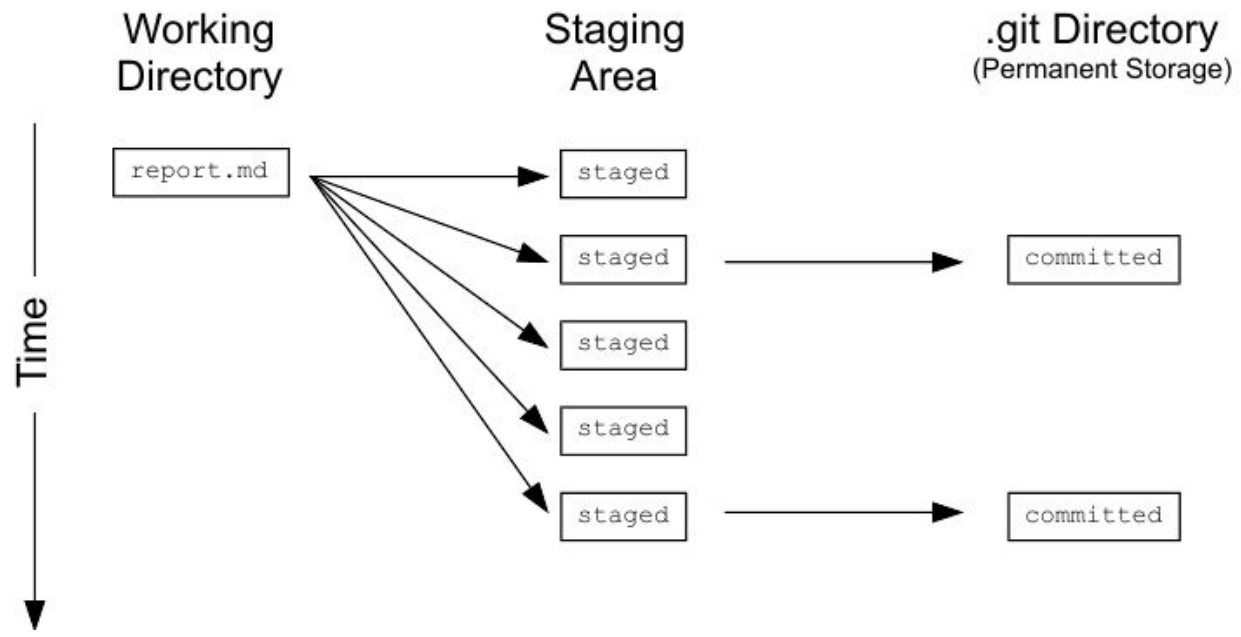
Vamos criar um novo projeto, um diretório que contém um arquivo “index.html” por exemplo, e adicionar um código básico a ele:

```
<p>Olá</p>
<p>Algum texto</p>
```

git init

Vamos escrever nosso primeiro comando Git. Executar `git init` no diretório que você está transformará seu projeto em um projeto gerenciado pelo Git. Você deverá ver a mensagem **Initialized empty Git Repository** no seu terminal, confirmando que o Git agora monitora o projeto. Mas porque nosso projeto está vazio?

O Git tem uma **staging area** no qual armazena os arquivos com mudanças que você quer salvar e que não foram salvas ainda. Colocar arquivos na staging area é como colocar arquivos em uma caixa, enquanto dar um commit nessas mudanças é como enviar essa caixa por email: você pode adicionar mais coisas a caixa ou tirá-las, mas uma vez que você envia o email, você não pode fazer mais mudanças. Abaixo temos um esquema do que foi dito.



git status

git status nos informa sobre o estado atual do repositório. Existem arquivos não rastreados, o que significa que o Git está ciente dos arquivos localizados no nosso diretório, mas que precisamos explicitamente dizer ao Git para rastrear esses arquivos (isto é, checar se há mudanças no arquivo) e salvar quaisquer mudanças em um novo commit.

git add

Para rastrear arquivos (ou seja, mandá-los para a staging area), podemos executar tanto git add filename (em que "filename" é o nome do arquivo que você quer rastrear) ou git add . para rastrear mudanças em todos os arquivos dentro do repositório. Com git status podemos ver que adicionamos um novo arquivo ("index.html", no nosso caso) ao repositório, mas nós não salvamos quaisquer mudanças no código ainda.

git commit

Por último, precisamos commitar ou salvar a versão atual do código em nosso repositório. Quando você commita as mudanças, o Git requisita que você entre com uma **log message**. Isto tem o mesmo propósito de fazer um comentário em um

programa: ele informa a próxima pessoa que usará o repositório por que você fez uma mudança.
Para executar um commit, você pode executar `git commit`, e então o Git abre um editor de texto no terminal para você colocar sua mensagem, ou então você pode executar `git commit -m "Mensagem"`, onde a flag `-m` nos possibilita escrever a mensagem do commit diretamente. Em nosso caso, usamos `git commit -m "Adicionado código inicial"`.

```
guilherme@guilherme-un:~/Repo$ git commit -m "Adicionado código inicial"
[master (root-commit) 047fa5e] Adicionado código inicial
 1 file changed, 2 insertions(+)
 create mode 100644 index.html
guilherme@guilherme-un:~/Repo$
```

Explorando branches e commits

Aqui vale fazer um pequeno resumo antes do que fizemos até agora:

1. Transformamos nosso diretório em um repositório Git com `git init`
2. Pedimos ao Git para rastrear quaisquer mudanças em `index.html` com o comando `git add .`
3. Salvamos a versão atual do nosso código em um commit com o comando `git commit -m "Adicionado código inicial"`

Até agora simples, né? Mas e quanto as branches? Dissemos anteriormente que a branch master é automaticamente criada no nosso primeiro commit, mas onde podemos ver esta branch?

<u>git branch</u>

`git branch` lista todas as branches em nosso repositório. Apesar de não termos dado nenhum nome, a branch master foi criada no primeiro commit:

```
guilherme@guilherme-un:~/Repo$ git branch
* master
guilherme@guilherme-un:~/Repo$
```

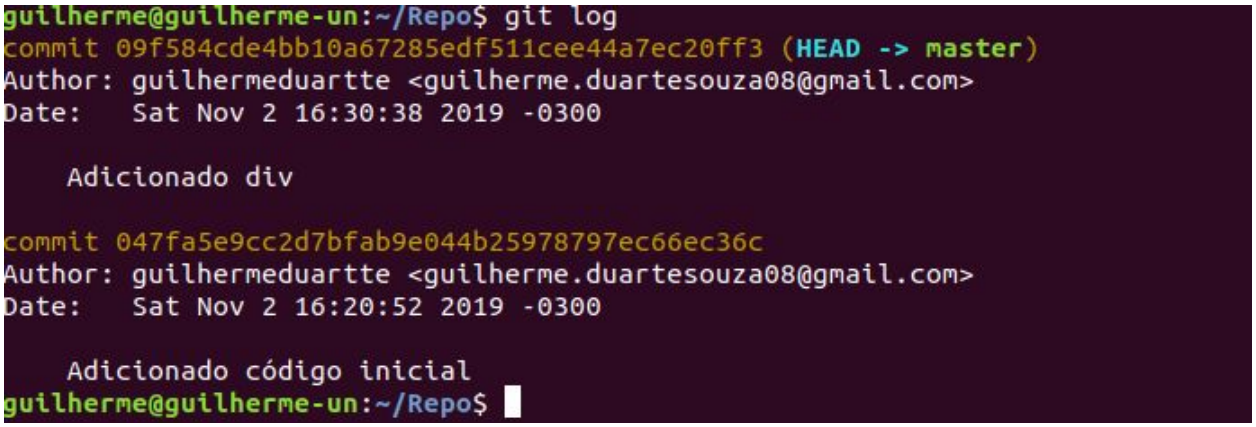
Temos um branch apenas até o momento. O asterisco indica que também estamos atualmente trabalhando nesta branch.
Vamos adicionar mais código ao nosso arquivo index.html:

```
<p>Olá</p>
<p>Algum texto</p>
<div>Outro texto</div>
```

Com `git add .` e `git commit -m "Adicionado div"` podemos adicionar este novo código como um segundo commit em nossa branch.

git log

Executar git log no terminal mostrará todos os commits dentro da nossa branch:



Cada commit contém um ID único, o autor, a data e a mensagem de commit. "HEAD" apenas aponta para o último commit da branch atual. Falaremos mais no próximo capítulo.

Mas como o Git cria esses commits? Cada commit não é uma cópia atualizada do commit anterior. Depois do primeiro commit, Git apenas rastreia as mudanças para cada commit seguinte. Então com git add ., Git checa nossos arquivos para quaisquer mudanças, e então git commit salva essas mudanças em um novo commit.

Rastrear as mudanças dessa maneira torna o Git muito mais rápido e eficiente do que criar inúmeras cópias. Fazer isso consumiria muito espaço e tempo de execução, e é algo que definitivamente queremos evitar em qualquer projeto.

HEAD

Vamos olhar mais de perto para o HEAD.

Adicionamos outro "div" e criamos um novo commit (git add . e git commit -m "Testando o head"). Como explicado anteriormente, este commit se tornou o HEAD pois foi o último commit adicionado em nossa branch:

```
commit edb5432ad73f15fbc03dbacddd4b1de6302de279 (HEAD -> master)
Author: guilhermeduartte <guilherme.duartesouza08@gmail.com>
Date:   Sat Nov 2 16:39:21 2019 -0300

    Testando o head

commit 09f584cde4bb10a67285edf511cee44a7ec20ff3
Author: guilhermeduartte <guilherme.duartesouza08@gmail.com>
```

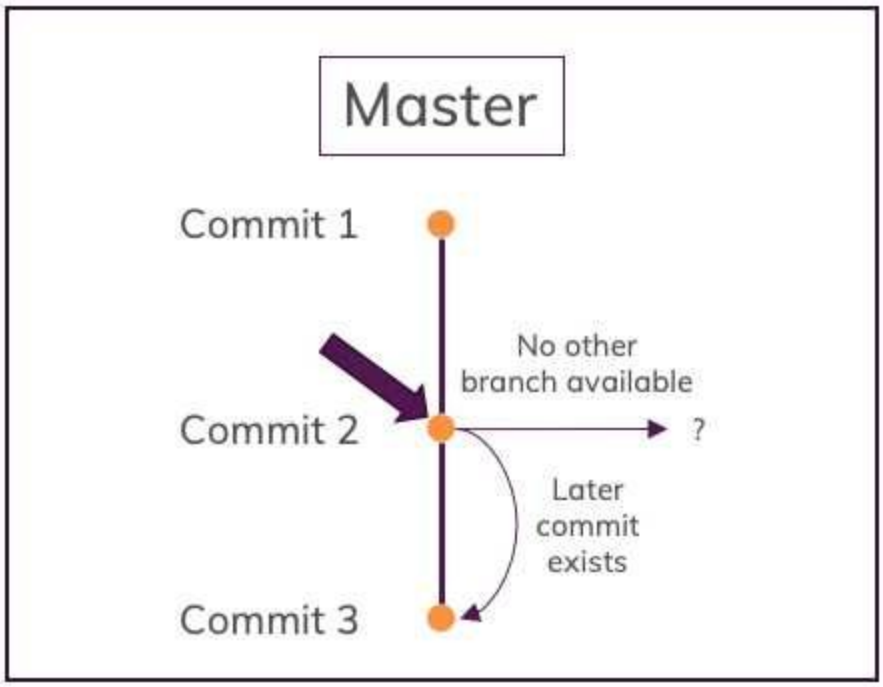
O último commit nem sempre é aquele que queremos trabalhar no momento. Temos a opção de simplesmente dar uma olhada rápida em um commit anterior ou talvez queiramos voltar a um commit anterior e deletar os commits posteriores a ele.

git checkout

git checkout é comando que nos permite consultar uma branch ou um commit específico. No caso, tentaremos checar um commit anterior.
Podemos usar git log para pegar o ID de um commit:

```
commit 09f584cde4bb10a67285edf511cee44a7ec20ff3
Author: guilhermeduartte <guilherme.duartesouza08@gmail.com>
Date:   Sat Nov 2 16:30:38 2019 -0300
```

Vamos checar o segundo commit nomeado “Adicionado div” com o comando git checkout commitID (basta copiar e colar o ID do commit que você está interessado na parte “commitID”).
Nosso HEAD está agora apontando para este commit, mas o commit que estamos checando não faz parte da nossa branch master. É como se estivéssemos no meio do nada nesse estágio, pois esse estado não pertence a nenhuma branch. Então esse comando nos permite apenas voltar atrás para dar uma olhada no código anterior.
Se você quiser voltar a este commit e jogar fora os commits posteriores. O git checkout não vai funcionar.



Para fazer isso, primeiro temos que voltar para nossa branch master com `git checkout master` (ou `git master`). Ao fazer isso, estamos de volta ao último commit da branch (nosso HEAD).

Deletar o último commit é fácil agora. Basta copiar o ID do commit ao qual você quer voltar (e não o que você quer deletar) e usar `git reset --hard commitID`. É isso, agora resetamos o HEAD e deletamos todos os commits posteriores.

Cuidado! Após deletar um commit, não podemos desfazer essa mudança. Então pense duas vezes antes de usar este comando.

Revertendo mudanças que não foram commitadas

Vamos novamente adicionar código:

```
<p>Olá</p>
<p>Algum texto</p>
<div>Outro texto</div>

<div>Alguma coisa que eu não preciso</div>
```

Acontece que em alguns casos, não precisamos desse código e gostaríamos de deletá-lo - as mudanças não foram commitadas ainda. Além de deletar manualmente nosso código (uma tarefa que pode não ser fácil às vezes), o comando `git checkout -- .` nos permite voltar ao estado do último commit.

Trabalhando com branches

Até esse ponto, trabalhamos apenas na branch master. Em um projeto na vida real, tipicamente você tem múltiplas branches, por exemplo:

- A branch master contém a versão estável e atual de um site
- A branch feature onde você está trabalhando atualmente em uma nova funcionalidade

O objetivo final será juntar ambas as branches assim que a nova funcionalidade estiver finalizada e pronta para ser entregue.

git checkout -b

git checkout -b cria uma nova branch baseado no último commit da branch que você está trabalhando. Como estávamos no último commit da branch master que contém o arquivo “index.html”, a nova branch criada vai então incluir também aquele código e todos os commits dentro daquela branch - basicamente, estamos copiando a branch inteira.

git checkout -b nova-feature vai criar uma nova branch chamada “nova-feature”:

```
guilherme@guilherme-un:~/Repo$ git checkout -b "nova-feature"
M       index.html
Switched to a new branch 'nova-feature'
guilherme@guilherme-un:~/Repo$ git branch
  master
* nova-feature
guilherme@guilherme-un:~/Repo$
```

Após criar a branch, automaticamente estamos trabalhando no último commit desta branch (lembre-se que você sempre pode checar isso com git branch e git log). Vamos adicionar apenas um arquivo vazio styles.css para manter as coisas simples.

git add . e git commit -m “Adicionado css” vão criar um novo commit com essas mudanças em nossa branch “nova-feature”.

Isso também resulta em dois diferentes HEADS. Na branch “nova-feature”, o commit que acabamos de criar é o HEAD. Na branch master, temos um HEAD diferente, o commit “Adicionado div”, o segundo da imagem abaixo:

```
commit afc9bc9d9969fd6f8d8e456f7b4fd36b0b8f8c2b (HEAD -> nova-feature)
Author: guilhermeduartte <guilherme.duartesouza08@gmail.com>
Date:   Sat Nov 2 17:12:35 2019 -0300

    Adicionado css

commit 09f584cde4bb10a67285edf511cee44a7ec20ff3 (master)
Author: guilhermeduartte <guilherme.duartesouza08@gmail.com>
```

Juntando branches

De volta a nossa branch master com `git master`, podemos agora combinar (fazer um merge) das nossas duas branches com o comando `git merge nova-feature`. Isto vai adicionar as mudanças feitas na branch "nova-feature" a nossa branch master e portanto nosso último commit "Adicionado css" é agora o HEAD de ambos os branches.

Deletando branches

Conforme nossa nova feature foi implementada na branch master, podemos nos livrar da branch "nova-feature" com `git branch -D nova-feature`.

Resolvendo conflitos

Fazer um merge das branches é uma funcionalidade fantástica, mas as vezes pode causar alguns conflitos.
Atualmente temos esse código no último commit em nossa branch master:

```
<p>Olá</p>
<p>Algum texto</p>
<div>Outro texto</div>
```

Vamos criar uma nova branch com `git checkout -b nova-feature` e vamos mudar o código da seguinte maneira:

```
<p>Tchau</p>
<p>Algum texto</p>
<div>Outro texto</div>
```

Após adicionar e commitar as mudanças, vamos trocar de volta para a master e mudar o código dessa maneira:

```
<p>Adeus</p>
<p>Algum texto</p>
<div>Outro texto</div>
```

Se adicionamos e commitamos essa mudança também, mexemos no mesmo código em duas branches diferentes (o primeiro `<p>`), então o que acontece se dermos um merge das branches agora?
`git merge nova-feature` nos dará um erro: "Automatic merge failed; fix conflicts and then commit the result".
Se olharmos o arquivo index.html, ele estará assim:

```
<<<<<<< HEAD
<p>Adeus</p>
=======
<p>Tchau</p>
>>>>>>> nova-feature
<p>Algum texto</p>
<div>Outro texto</div>
```

O conflito pode ser resolvido deletando o código que você não quer manter. Depois disso, basta fazer o commit da maneira usual. Isto resolve o conflito e junta as branches com sucesso. Depois de resolver o conflito, nosso index.html ficou assim:

```
<p>Olá</p>
<p>Algum texto</p>
<div>Outro texto</div>
```

Novamente, tenha cuidado quando você for deletar código, branches ou commits. Após deletar, a informação é perdida, e você não pode recuperá-la novamente.

Comandos úteis

Precisa de ajuda para achar algum comando? Aqui vai uma pequena ajuda:

- git init: inicializa um repositório Git no diretório atual
- git status: mostra o estado atual do seu repositório Git
- git add .: rastreia mudanças de todos os arquivos no seu repositório
- git commit -m “sua mensagem”: salva o código atualizado em um novo commit chamado “sua mensagem”
- git log: lista todos os commits dentro da sua branch
- git checkout nome-branch: pula para o último commit de uma branch
- git checkout commitID: pula para um commit específico de uma branch (commitID deve ser o ID do commit que você quer checar)
- git checkout -- . : pula para o último commit e remove qualquer mudança não rastreada
- git reset --hard: transforma o commit selecionado no novo HEAD
- git branch: lista todas as branches dentro do seu repositório
- git checkout -b nome-branch: cria uma nova branch chamada nome-branch
- git merge nome-branch: combina duas branches, nome-branch é a branch que você quer unir com a branch que você atualmente está
- git branch -D nome-branch: deleta a branch nome-branch

Guia básico de GitHub

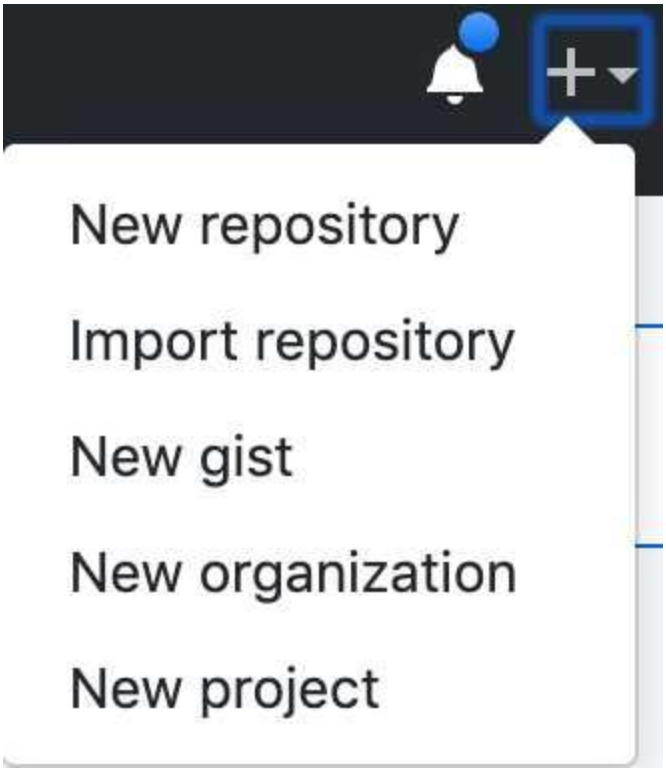
Com Git você pode gerenciar seu código localmente no seu computador. Se você está trabalhando em um projeto grande, não existe basicamente nenhum jeito de tornar seu código disponível para os outros membros de sua equipe. Um dos mais comuns provedores de tal serviço é o GitHub.

Introdução ao GitHub

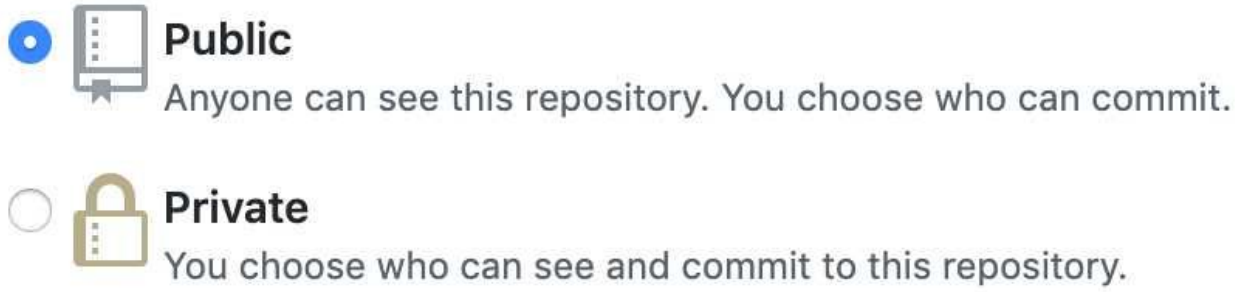
Como podemos fazer o upload de código do nosso repositório Git local para a nuvem em um repositório remoto e o que precisamos para começar? Precisamos apenas de um repositório local e uma conta no GitHub.

Criando um repositório remoto

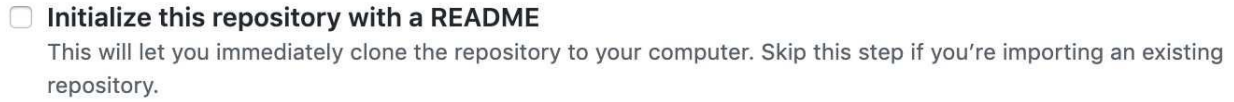
Depois de criar uma conta, você pode criar um novo repositório bem aqui:



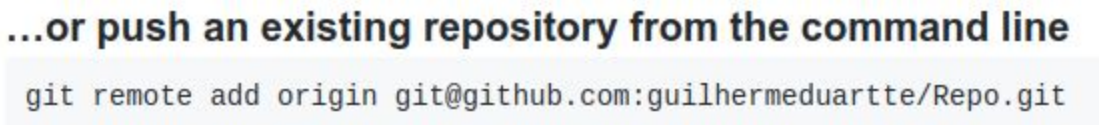
Mas se já temos um repositório local pronto, então por que deveríamos criar outro? A ideia é criar um repositório vazio no GitHub e então dar um push (isto é, fazer o upload) do conteúdo existente no repositório local para o repositório remoto.
Depois de dar um nome ao repositório e uma descrição (opcional, mas recomendada), temos que decidir se vamos criar um repositório público ou privado.



Como o nome indica, repositórios públicos podem ser vistos por qualquer um na sua página do GitHub, enquanto repositórios privados só estão disponíveis para apanas algumas pessoas que você dá permissão.
Para este guia, selecionaremos “Public” nesta parte.
“Initialize this repository with a README”. Marque esta caixa se você não tem código algum que você quer dar push para este novo repositório (sem o arquivo readme, o repositório não conteria nenhuma branch). Como nosso repositório Git local tem um projeto, isto significa que podemos manter esta caixa desmarcada.



Clicar em “Create Repository” vai criar um repositório remoto vazio. A adição do conteúdo do repositório local existente pode ser feita conforme descrito na página para a qual somos redirecionados automaticamente:



Aqui temos um comando importante: git remote add origin URL. Este comando precisa ser executado no seu diretório local do projeto (isto é, o diretório que contém o repositório Git que você quer fazer o push para o GitHub).

- remote add: adiciona um repositório remoto para o seu projeto Git. Em outras palavras, estabelecer uma conexão entre o repositório local e o remoto
- origin: este é o nome do repositório remoto. Ele pode receber outro nome, mas tipicamente é chamado de origin, então também usaremos essa convenção
- URL: é a URL do repositório remoto no GitHub (afinal, nossos dados precisam ser mandados para o endereço correto, certo?). O GitHub automaticamente adiciona o endereço conforme podemos ver no screenshot

git remote

Depois de executar o comando no diretório local do projeto, `git remote` mostra o repositório remoto no terminal (`git remote -v` mostra também a URL do repositório). Isto confirma que nós estabelecemos com sucesso uma conexão entre nossos repositórios local e remoto, mas que ainda não trocamos nenhum dado a partir desse ponto.

git push

`git push -u origin master` nos permite fazer o push (isto é, o upload) do conteúdo do nosso repositório local para o remoto. `-u origin master` diz ao Git para criar um caminho para um repositório e branch específicos, o que significa que queremos fazer o upload dos dados para a branch "master" do nosso repositório "origin".
O GitHub pede as nossas credenciais (isto é, o usuário e a senha da conta do GitHub), e após isso, se recarregarmos a página, devemos ver que nosso branch e os commits do repositório local foram carregados para o GitHub.

Branches local, remota e remota de rastreamento

É hora de se aprofundar um pouco na conexão entre Git e GitHub:

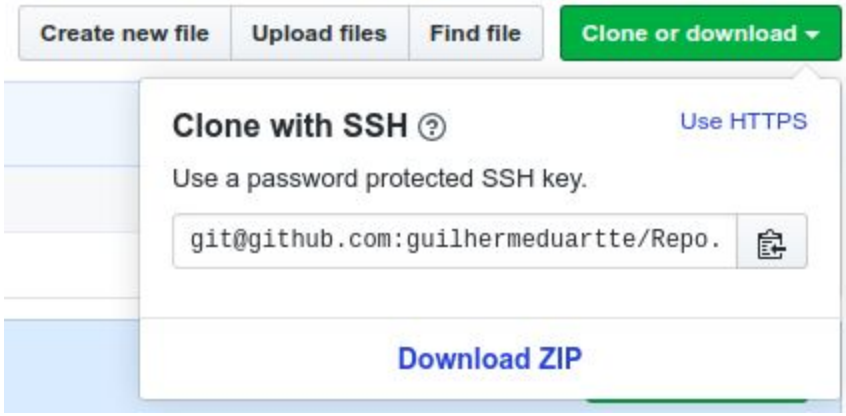
- `git branch` mostra todos os branches atuais em nosso repositório **local**
- `git remote` mostra o repositório **remoto** que criamos no GitHub
- `git branch -r` mostra a **branch remota de rastreamento**

As branches locais são as branches do nosso repositório local na nossa máquina, e o repositório remoto é o repositório na nuvem (no GitHub, nesse caso). Acho que isso já está claro. Mas e quanto a branch remota de rastreamento?
A branch remota de rastreamento é a chamada **representação local da branch remota**. Anteriormente nós usamos `git push -u origin master` para dizer ao Git para onde queríamos fazer um push do nosso código - a branch master no repositório remoto origin.
Depois da primeira vez que usamos "push" desta maneira explícita (o mesmo também é válido para "pull" e outros comandos como "clone", que falaremos mais à frente), esta branch remota de rastreamento é criada automaticamente.
Com a informação desta branch remota de rastreamento, o Git sabe que `git push` deve fazer o upload do código para "origin/master" e com isso o código certo pode ser trocado entre os repositórios e branches corretos.
Isto nos permite usar uma versão mais curta desses comandos no futuro, pois agora o Git está ciente do "endereço" do nosso repositório remoto e de sua branch correspondente.

Sempre que estamos trabalhando em nossa branch master local, executar git push é tudo que nós precisamos fazer para atualizar o código em nosso repositório remoto. Nota de rodapé: com “push”, ambos os dados da branch remota de rastreamento como os do repositório remoto serão atualizados.
Precisamos nos certificar de que todas as três branches (a local, a remota de rastreamento e a remota) estão sincronizadas. Vamos continuar com nosso trabalho no GitHub para ver por que isto é importante.

Permitindo o acesso ao repositório para outros usuários

A opção “Clone or download” em nosso repositório remoto fornece uma URL que pode ser compartilhada para dar a outros usuários o acesso ao repositório (é a mesma URL que usamos anteriormente quando fizemos o push do nosso repositório local para o GitHub):



Vamos criar um novo projeto vazio que deve conter nosso código do Repositório GitHub. Nós já temos a URL, e então com git clone URL (onde “URL” é a URL que acabamos de copiar do GitHub), o repositório remoto inteiro é copiado dentro do diretório e também transforma este diretório em um repositório gerenciado pelo Git - é fácil assim.
Nós clonamos nosso repositório e também estamos habilitados a fazer um push dos dados para este repositório remoto. git push é tudo que precisamos aqui.
Por que o git push é suficiente? Porque o git clone automaticamente cria a branch remota de rastreamento que contém a informação sobre o repositório e sobre a branch do nosso repositório remoto. Como o Git sabe de onde nós clonamos os dados, nós também podemos fazer o push dos dados de volta para este “endereço” agora.
Você sempre pode checar a existência da branch remota de rastreamento com “git branch -r”. Sempre que você usa “git push”, tanto a branch remota de rastreamento quando a branch remota são atualizados com a informação da branch local.

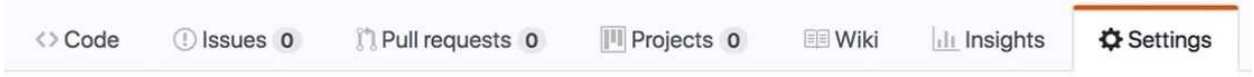
Importante: Você pode controlar quem tem acesso ao seu repositório GitHub. Se seu repositório é público, pessoas sempre podem cloná-lo, mas você pode bloquear o pushing.

Baixando dados do GitHub

Fazer o push de código ou clonar um repositório é ótimo, mas nós também podemos receber informação atualizado mesmo após clonarmos os dados. `git fetch` baixa informação de um repositório remoto para a branch remota de rastreamento. Isto é importante: a branch remota de rastreamento será atualizada, mas a branch local não.
O segundo passo é sincronizar nossa branch local e a branch remota de rastreamento. Nós aprendemos anteriormente no guia de Git o comando merge. Então, basta fazer um merge da branch remota de rastreamento com nossa branch local. Depois de um merge bem sucedido, nossas três branches estarão sincronizadas novamente.
Uma maneira mais conveniente de obter o mesmo resultado é com o comando `git pull` que basicamente é a combinação do `git fetch` e do `git merge`.

Deletando repositórios

Nós também podemos deletar repositórios remotos. No GitHub, vá para a página "Settings":



Se você descer até a "Danger Zone", você vai achar a opção para deletar um repositório específico. Depois de confirmar o nome, o repositório é deletado. **Tenha cuidado**, um repositório remoto deletado pode causar grandes problemas.
Além de deletar o repositório do GitHub, podemos remover repositórios do nosso projeto Git - isto é, a conexão entre Git e GitHub, mas o repositório não será excluído. Basta executar `git remote rm origin`. Certifique-se de checar seus repositórios remotos com "git remote" primeiro, pois repositórios remotos algumas vezes podem ter nomes diferentes do que "origin".

Overview dos comandos core de GitHub

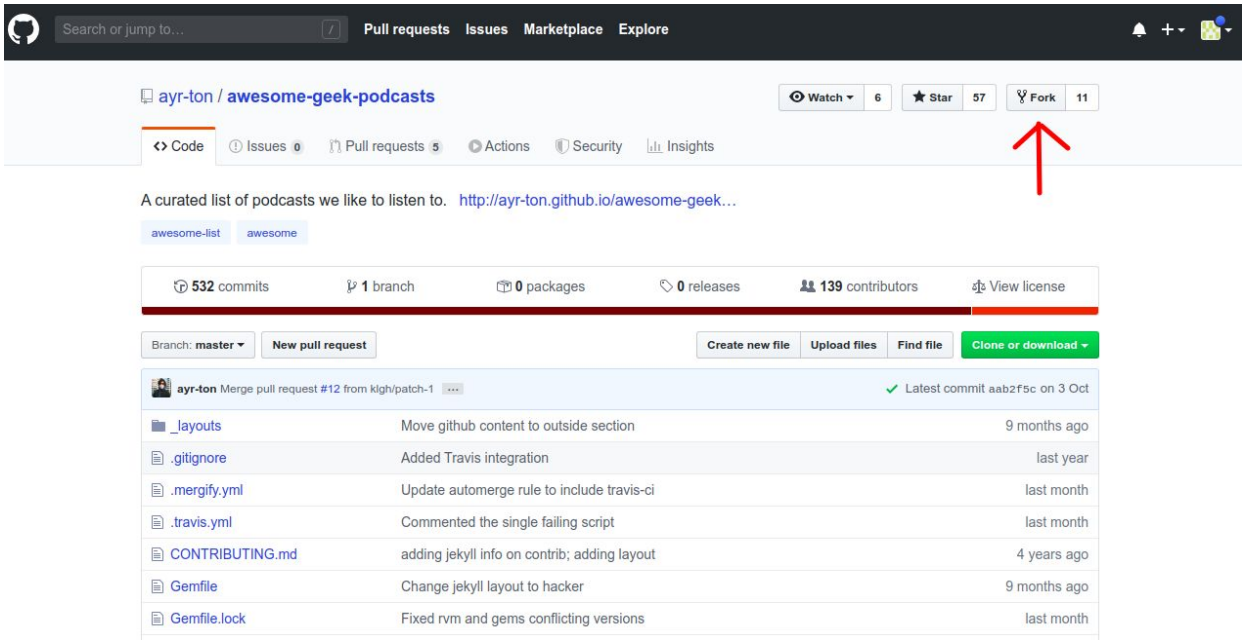
Precisa de uma consulta rápida em um termo ou comando específico? Aqui vamos nós:

- `git remote add origin URL`: estabelece uma conexão entre o repositório Git local que você está trabalhando atualmente e o repositório remoto (URL deve ser a URL deste repositório remoto)

- git remote -v: mostra todos os repositórios remotos ao qual seu repositório local está conectado (“-v” também mostra o URL do repositório remoto)
- git push -u origin master: faz o upload do código da sua branch local para uma branch remota no repositório remoto (“origin” é o nome do repositório remoto, enquanto “master” é o nome da branch remota)
- git clone: copia ou clona um repositório remoto para a sua máquina. Automaticamente cria uma branch remota de rastreamento
- git push e git pull: faz o upload do código para ou baixa o código de um repositório remoto. Apenas possível depois que uma branch remota de rastreamento foi criada
- “branch remota de rastreamento”: Branch “entre” a branch local e a branch remota. Garante que o código recebe um push ou um pull entre os repositórios e branches corretos. Automaticamente criada quando usado “git clone” ou “git push”/”git pull” explicitamente (isto é, com a informação referente ao nome do repositório e da branch)
- git fetch: atualiza a branch remota de rastreamento com o código da branch remota
- git remote rm origin: remove a conexão entre o repositório local e o repositório remoto, onde “origin” é o nome do repositório remoto

Contribuições para projetos Open Source

Como sabemos, o GitHub hospeda o código de diversas iniciativas open source. Projetos open-source que são hospedados em repositórios públicos beneficiam-se de contribuições feitas pela ampla comunidade de desenvolvedores através de pull requests, que solicitam que um projeto aceite as alterações feitas em seu repositório de código. Vamos mostrar agora como fazer um pull request para um repositório Git para que você possa contribuir com projetos de software open-source.
O primeiro passo é fazer um fork do repositório. Para isso, você deve acessar a URL de algum repositório público no GitHub. Quando você estiver na página principal do repositório, você verá um botão “Fork” no seu lado superior direito da página, abaixo do seu ícone de usuário:



No exemplo acima, **ayr-ton** é o nome do usuário e **awesome-geek-podcasts** é o nome do repositório.

Clique no botão fork para iniciar o processo de fork. Quando o processo estiver concluído, o seu navegador irá para uma tela semelhante à imagem do repositório acima, exceto que no topo você verá seu nome de usuário antes do nome do repositório, e na URL ela também mostrará seu nome de usuário antes do nome do repositório.

Então, no exemplo acima, em vez de **ayr-ton** / **awesome-geek-podcasts** na parte superior da página, você verá **seu-nome-de-usuário** / **awesome-geek-podcasts**, e a nova URL será parecida com isto: github.com/seu-nome-de-usuário/awesome-geek-podcasts. Com o fork do repositório realizado, você está pronto para cloná-lo para que você tenha uma cópia de trabalho local da base de código.

Agora você precisa fazer os processos que já vimos anteriormente. Então você deve clonar o repositório em um repositório local, criar uma nova branch para que você possa trabalhar nela e fazer os commits localmente, e fazer o push para o seu repositório remoto.

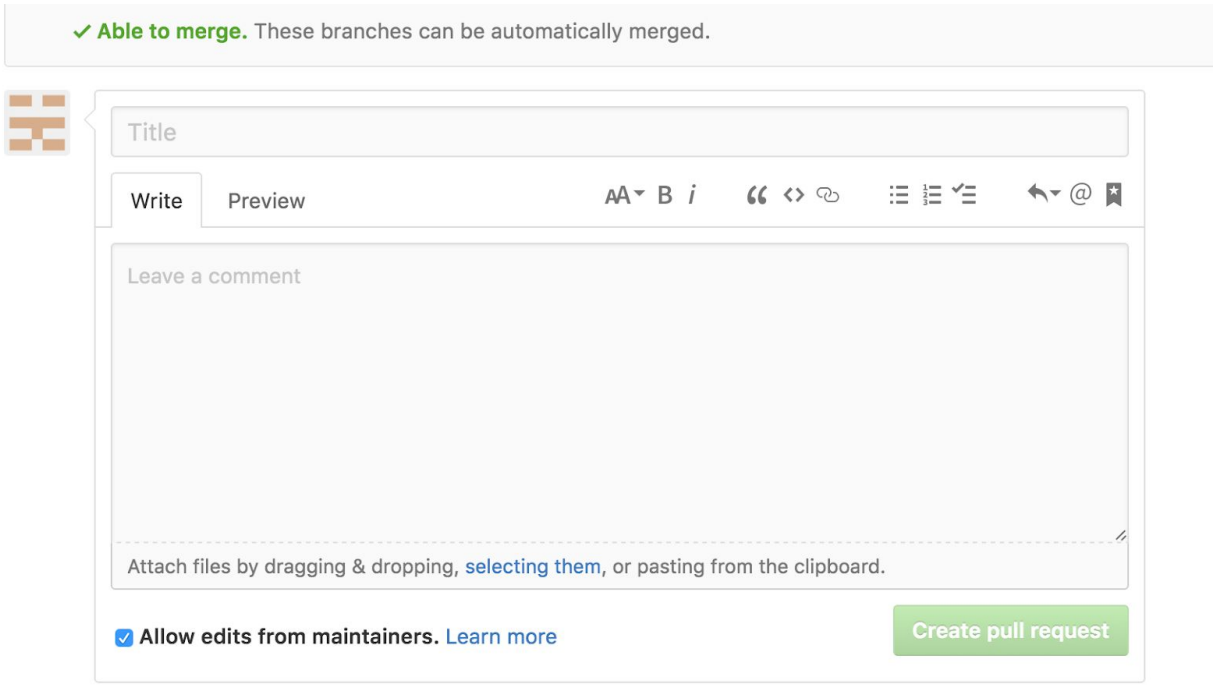
Diferentemente de quando estamos trabalhando em um projeto nosso, precisamos configurar o repositório remoto para o fork. Caso você execute git remote -v, o output deve ser algo como:

origin https://github.com/seu-nome-de-usuário/repositório-forked.git (fetch)
origin https://github.com/seu-nome-de-usuário/repositório-forked.git (push)

Em seguida, vamos especificar um novo repositório remoto upstream para sincronizarmos com o fork. Este será o repositório original do qual fizemos o fork.

Faremos isso com o comando git remote add upstream URL, onde o URL deve ser o URL do repositório original, ou seja, algo da forma https://github.com/nome-de-usuário-do-proprietário-original/repositório-original.git.
Nesse exemplo, upstream é o nome abreviado que fornecemos para o repositório remoto, já que em termos do Git, “Upstream” refere-se ao repositório do qual nós clonamos.
Agora você pode se referir ao upstream na linha de comando em vez de escrever a URL inteira, e você está pronto para sincronizar seu fork com o repositório original.
Depois de configurarmos um repositório remoto que faça referência ao upstream e ao repositório original no GitHub, estamos prontos para sincronizar nosso fork do repositório para mantê-lo atualizado.
Para sincronizar nosso fork, a partir do diretório do nosso repositório local em uma janela de terminal, vamos utilizar o comando git fetch para buscar as branches juntamente com seus respectivos commits do repositório upstream. Como usamos o nome abreviado “upstream” para nos referirmos ao repositório upstream, passaremos o mesmo para o comando: git fetch upstream.
Agora, os commits para o branch master serão armazenados em uma branch local chamada upstream/master. Basta mudar para a branch master local do nosso repositório e executar git merge upstream/master.A branch master do seu fork agora está em sincronia com o repositório upstream, e as alterações locais que você fez não foram perdidas. Dependendo do seu fluxo de trabalho e da quantidade de tempo que você gasta para fazer alterações, você pode sincronizar seu fork com o código upstream do repositório original quantas vezes isso fizer sentido para você. No entanto, você certamente deve sincronizar seu fork antes de fazer um pull request para garantir que não contribuirá com código conflitante.
Neste ponto, você está pronto para fazer um pull request para o repositório original.
Você deve navegar até o seu repositório onde você fez o fork e pressionar o botão “New pull request” no lado esquerdo da página:

Depois de ter escolhido, por exemplo, a branch master do repositório original no lado esquerdo, e a nova-branch do seu fork do lado direito, você deve ver uma tela assim:



O GitHub vai lhe alertar de que é possível mesclar as duas branches porque não há código concorrente. Você deve adicionar um título, um comentário e, em seguida, pressionar o botão “Create pull request”. Neste ponto, os mantenedores do repositório original decidirão se aceitam ou não o seu pull request. Eles podem solicitar que você edite ou revise seu código antes de aceitar o pull request.

Introdução a integração contínua com Jenkins e GitHub

Implementar Integração Contínua (CI, do inglês Continuous Integration) tem o potencial de reduzir erros no código, encurtar ciclos de desenvolvimento, e ajudar sua equipe a entregar o software mais rápido. Agora, mostraremos como começar com CI, e daremos uma olhada mais de perto na integração do GitHub com o Jenkins.

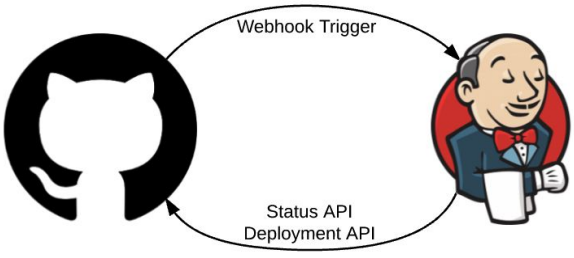
Porque CI é importante?

Atualmente, as empresas precisam lançar software mais rápido do que nunca, e CI tem um papel importante nessa tarefa. Com CI em prática, times podem construir, testar, e atualizar seu código dentro de minutos, aumentando a qualidade e reduzindo o tempo de produção.
Com a constante ambiente de mudança no campo da tecnologia, times podem usar diversos tipos e combinações diferentes de ferramentas. Hoje em dia, o GitHub tem integração com centenas de outras ferramentas third party, incluindo algumas das mais populares e bem-documentadas ferramentas de integração - todas disponíveis no Marketplace.

O que é Jenkins?

Jenkins é uma das mais populares ferramentas de CI no mercado, com o número estimado de pelo menos 1 milhão de usuários e cerca de 150 mil instalações pelo mundo. Também possui suporte de primeira linha para projetos GitHub e GitHub Enterprise na sua instalação padrão. Com conceitos como Pipeline-as-Code, o processo inteiro de compilação pode ser submetido no GitHub e versionado como o resto do código do seu time.

Como funciona com GitHub?



Jenkins escaneia sua organização GitHub inteira e cria rotinas de Pipeline para cada repositório contendo um Jenkinsfile - um arquivo de texto que define o processo de construir, testar e fazer o deploy do seu projeto usando Jenkins. Imediatamente depois do código ser verificado ou quando um novo pull request é criado, Jenkins irá executar a rotina de Pipeline e retornar o status para o GitHub indicando se houve falha ou sucesso. Este processo te permite executar e realizar testes automatizados subsequentes de modo que apenas o melhor código é merged. Detectar bugs cedo e automaticamente reduz o número de problemas introduzidos quando o produto for colocado no ar, permitindo que seu time possa construir software mais eficiente.
Os deployments que ocorrem no Jenkins também pode ser gravados de volta no GitHub, de modo a manter o histórico de todo o ciclo de produção.
Aqui segue um link que mostra o processo de construir um pipeline de entrega contínua com Git & Jenkins.

Conclusão

No guia acima tivemos uma introdução às ferramentas Git e GitHub, e pincelamos o conceito de integração contínua com Git e Jenkins. Passamos pelo conceito de sistema de controle de versão, por que ele é útil e aprendemos os comandos básicos do Git. Depois, aprendemos como usar o GitHub, um servidor remoto de colaboração que hosteia projetos Git, fazendo a conexão entre um repositório local e um repositório remoto, e mostrando como manter os dois ambiente sincronizados. Também mostramos como você pode colaborar com projetos open source fazendo um pull request. Esperamos que esse material seja de grande ajuda para aqueles que querem iniciar no Git e ter uma noção sobre o conceito de integração contínua, ferramentas essenciais no desenvolvimento de software hoje em dia.