GIT

Sua maior arma durante o Inverno que Está Chegando

Quem sou eu?

Por que estou aqui?

Pra onde vou?

Objetivos do tutorial

Pra quem não conhece git

Apresentar git né?

Pra quem já conhece

Apresentar como alguns comandos essenciais funcionam por dentro para que:

- Usar o ferramental seja mais intuitivo
- Todo mundo tenha jiu gitsu suficiente pra saber desenrolar quando um problema aparecer
- As buscas por comandos avançados ou *snippets* seja mais objetiva

O que *iremos* ver nesse tutorial?

- Aprender a usar alguns comandos essenciais do git
- Aprender como os comandos funcionam internamente
- Colaboração básica usando GitHub
- ~~Versão corrigida do final de GOT~~ sonha

O que *não iremos* ver nesse tutorial?

- Dicas ninja de como reduzir seu workflow a uma linha de comando
- Comandos avançados
- Comparação entre VCS

SETUP

[X] Conta no GitHub

[X] git instalado

O que *é* git?

Programa que ajuda no controle de versão

Speaker notes

- git é um programa feito para controle de versão de um projeto

Todas as modificações no projeto são monitoradas e podem ser versionadas

No geral, git dá a liberdade de adicionar:

- "comentários"
- "versões nomeadas"
- "variações do projeto"

Speaker notes

- ... e muito mais é possível ser feito com git em qualquer projeto

E também é muuuuito fácil fazer *colaboração*

Agora vamos começar nosso projeto

```
$ git clone https://github.com/giuliocc/git-tut-pybr15-files.git
$ mkdir git-tutorial-pybr15
$ cd git-tutorial-pybr15
$ unzip ../git-tut-pybr15-files/00files.zip
```

Speaker notes

se não estiver em um terminal, o que estes comandos estão fazendo é:

- baixar o repositório com os arquivos necessários
- criar um diretório chamado "git-tutorial-pybr15"
- "entrar" no diretório
- extrair o conteúdo de 00files.zip para git-tutorial-pybr15

E agora? Onde entra o git?

git init

Ao executar `git init`, seu projeto se torna um *repositório git*

Speaker notes

- o projeto é um diretório que passa a ser versionado pelo git a partir do momento que é executado o comando `git init` nele
  - a partir desse momento, o diretório pode ser chamado de "repositório"

A partir de agora tudo pode ser monitorado e versionado

Speaker notes

- tudo que é feito no repositório é monitorado pelo git, e, utilizando os devidos comandos, qualquer alteração pode ser adicionada à história do projeto

Mas *como* git faz tudo isso?

Speaker notes

- ao executar o comando git init, o git irá criar um diretório que contém todas as informações sobre o projeto que está sendo versionado
- .git, o diretório que diferencia um diretório comum de um repositório
  - nele são encontrados os objetos mais importantes que compõem toda a arquitetura git:

- **blob**:
  - Geralmente a representação binária de um arquivo
- **tree object**:
  - São como abstrações de diretórios UNIX, um *tree object* contém referências para um ou mais *blobs* ou outros *tree objects*

- **commit object**:
  - Objeto que possui a referência para um *tree object* e alguns metadados
- **reference**:
  - Objeto que possui referência para um *commit object* ou *tag object*

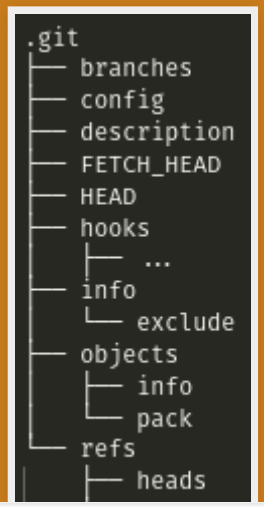
Mais sobre isso depois...

Speaker notes

só passar por cima desses tópicos, a gente vai voltar pra eles depois

Hora de executar a mágica:

```
$ git init
```



Speaker notes

o .git continua "vazio"

Agora como é que a gente faz pra encher o .git?

Tudo que adicionamos até agora faz parte do nosso *working directory*

- **working directory**:
  - Estado do projeto que estou vendo e trabalhando no momento

É possível inspecionar o *working directory* a qualquer momento

git status

```
On branch master

No commits yet

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

    .gitignore
    LICENSE
    README.md
    index.html

nothing added to commit but untracked files present (use "git
add" to track)
```

Speaker notes

aqui pode ser visto que ao executar `git status` o git mostra os arquivos como *untracked*

- **untracked**:
  - É o que ainda não está sob versionamento do git no *working directory* atual

Speaker notes

normalmente existem arquivos que são criados mas que não devem estar sob versionamento (.pyc, por exemplo), então o git não versiona eles por padrão e deixa essa decisão a seu cargo, o desenvolvedor

o .gitignore que adicionamos no projeto não tem nada, mas se ele tivesse incluiria ou excluiria arquivos no git status

Vamos fazer a primeira versão do nosso projeto com dois comandos:

`git add`

`git commit`

Speaker notes

Para encher o .git e criarmos a primeira versão do nosso projeto precisamos utilizar a sequência dos comandos `git add` e `git commit`

git add

```
$ git add .gitignore LICENSE README.md index.html
$ git status
```

```
On branch master

No commits yet

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

    new file:   .gitignore
    new file:   LICENSE
    new file:   README.md
    new file:   index.html
```

Speaker notes

aqui pode ser visto que depois `git status` e ver que eles estão marcados como "changes do be committed", ou seja, staged

- **staged**:
  - É o que está pronto para ser versionado, mas ainda não foi versionado. Aquilo que constitui as alterações que serão feitas neste instante e serão gravadas na história do seu projeto, é o chamado *staged snapshot*
- **unstaged**:
  - É o que não está no *staged snapshot*

Speaker notes

então a função do `git add` é colocar arquivos no staged snapshot

Beleza, temos arquivos prontos para serem versionados...

git commit

Com o `git add` já executado, agora só é necessário executar o

```
$ git commit
```

Ops, um erro foi (talvez) identificado

```
*** Please tell me who you are.

Run

  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.
```

Execute os comandos exibidos na tela com o email do seu cadastro no GitHub e seu nome

Speaker notes

Os commits usam alguma identificação para que outros colaboradores possam saber que fez aquela versão

De novo

```
$ git commit
```

```
Adiciona setup do repositório
# Please enter the commit message for your changes. Lines
# starting with '#' will be ignored, and an empty message aborts
# the commit.
#
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   new file:   .gitignore
#   new file:   LICENSE
#   new file:   README.md
#   new file:   index.html
#
```

Speaker notes

Todo commit precisa de uma mensagem para comentá-lo, seja objetivo mas ao mesmo tempo descritivo nas mensagens que coloca. Outros colaboradores ou até você mesmo irão agradecer

Um commit é como uma nova versão do seu projeto, aquilo que você decidiu colocar na história, é o chamado *committed snapshot*

A mensagem do commit é o comentário que você grava sobre aquele snapshot

E para vermos o commit que acabamos de fazer?

git log

Basta só executar o comando

```
$ git log
```

```
commit 3be10b5165f29ee4f35b008e00f65cc24ac2c0b5 (HEAD -> master)
Author: Giulio Carvalho <gcc@cin.ufpe.br>
Date:   Tue Oct 22 22:03:30 2019 -0300

    Adiciona setup do repositório
```

Speaker notes

Aqui podemos ver que o commit tem um hash, além de outras informações.

Para entender o que isso significa, vamos voltar àqueles conceitos de *blob*, *tree object* e *commit object*

....ignora a data....

- **blob**:
  - Geralmente a representação binária de um arquivo

- **tree object**:
  - São como abstrações de diretórios UNIX, um *tree object* contém referências para um ou mais *blobs* ou outros *tree objects*

Speaker notes

A raíz do repositório é uma tree que, ao subir nos galhos, atinge-se todo o snapshot

- **commit object**:
  - Objeto que possui a referência para um *tree object* e alguns metadados

```
commit 3be10b5165f29ee4f35b008e00f65cc24ac2c0b5
tree bcc00f2d6d5410595bb304b5159fb86e4bcd2477
author Giulio Carvalho <gcc@cin.ufpe.br> 1571792610 -0300
committer Giulio Carvalho <gcc@cin.ufpe.br> 1571792610 -0300

    Adiciona setup do repositório

diff --git a/.gitignore b/.gitignore
new file mode 100644
index 0000000..e69de29
diff --git a/LICENSE b/LICENSE
new file mode 100644
index 0000000..368dcf3

...
```

Speaker notes

um commit aponta para o snapshot completo, ou seja, cada commit é uma cópia completa do seu repositório

Beleza, agora sabemos como que o versionamento funciona, vamos adicionar mais commits

```
$ unzip -o ../git-tut-pybr15-files/01files.zip
$ git add index.html
$ git status
```

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

    modified:   index.html
```

Speaker notes

aqui a gente pode ver que os arquivos modificados agora aparecem como *tracked*

- **tracked**:
  - Arquivos *tracked* são aqueles que já estão no working directory daquele commit e estão sendo modificados

Continuando...

```
$ git commit -m "Adiciona relevância de Ygritte pro plot"
```

Speaker notes

o -m aqui é um atalho que já pula a parte de inserir a mensagem do commit caso deseje

E mais alguns..

```
$ unzip -o ../git-tut-pybr15-files/02files.zip
$ git add index.html
$ git commit -m "Adiciona resultado da decisão do Snôu"
$ unzip -o ../git-tut-pybr15-files/03files.zip
$ git add index.html
$ git commit -m "Adiciona fim da história do Snôu"
$ git log
```

```
commit 964d3c2552a63c8e54e5d8a3ed69ad5e767a6537 (HEAD -> master)
Author: Giulio Carvalho <gcc@cin.ufpe.br>
Date:   Tue Oct 22 23:19:25 2019 -0300

    Adiciona fim da história do Snôu

commit c981492416ef8c02cf0b6ddcd6ddc7773a606f8c
Author: Giulio Carvalho <gcc@cin.ufpe.br>
Date:   Tue Oct 22 23:16:28 2019 -0300

    Adiciona resultado da decisão do Snôu

commit 990a49c9415e67bba644b186788586786cd6be6d

...
```

Speaker notes

Beleza, já temos alguns commits, uma coisa que não falei sobre commit objects é que o hash do ancestral deles é um dos metadados

Parent

```
commit 964d3c2552a63c8e54e5d8a3ed69ad5e767a6537
tree 3783291106bf1bfb15d15e5b68f28318ce5df865
parent c981492416ef8c02cf0b6ddcd6ddc7773a606f8c
author Giulio Carvalho <gcc@cin.ufpe.br> 1571797165 -0300
committer Giulio Carvalho <gcc@cin.ufpe.br> 1571797165 -0300

    Adiciona fim da história do Snôu

diff --git a/index.html b/index.html
index e24ba1b..2680bfa 100644
--- a/index.html
+++ b/index.html

...
```

Speaker notes

Com essa informação é possível criar a lista encadeada de commits que vemos no git log

- **commit history**:
  - É a conexão entre vários commits. Isso quer dizer que, a partir de um commit você consegue recuperar toda a história dele, do fim até o primeiro commit na cadeia

Uma dúvida:

Se cada commit desses contém todo o repositório, o que acontece quando temos muitos commits?

O repositório explode?

O git reaproveita os *blobs* que já existem no repositório para montar os *tree objects* :)

E se quisermos fazer *variações* desse projeto?

git branch

- **branch**:
  - É como uma variação da história do projeto. Cada branch tem sua própria *commit history*

Com o comando `git branch` podemos ver as branches locais

```
$ git branch
```

```
* master
```

Speaker notes

Aqui podemos ver que a branch que estamos atualmente é a marcada com asterisco, a branch master é a branch principal padrão do git. Mas nada impede de você usar outra branch como principal

Vamos criar uma branch e visualizar a branches locais novamente

```
$ git branch revive-jon
$ git branch
```

```
* master
  revive-jon
```

Como fazemos para mudar de branch?

git checkout

Agora vamos mudar de branch e listar novamente a branches locais

```
$ git checkout revive-jon
$ git branch
```

```
  master
* revive-jon
```

O que aconteceu agora?

- **HEAD**:
  - Arquivo especial do git que aponta para outra arquivo que representa a branch atual

Speaker notes

o arquivo especial HEAD aponta para o commit que o seu "working directory" está vendo no momento

Speaker notes

A branch master era a branch que estávamos, o git sabe disso por conta do arquivo especial HEAD, este arquivo aponta para um arquivo que representa a branch master

O arquivo HEAD aponta para o arquivo que representa a branch master

```
$ git checkout master
$ less .git/HEAD
```

```
ref: refs/heads/master
```

```
$ less .git/refs/heads/master
```

```
964d3c2552a63c8e54e5d8a3ed69ad5e767a6537
```

Este hash é o hash do último commit quando executo um `git log` na master

- **reference**:
  - Objeto que possui referência para um *commit object* ou *tag object*

Speaker notes

a gente não vai falar sobre *tag object* hoje, mas é basicamente um objeto que normalmente é usado pra identificar uma release

quem quiser olhar depois `git tag`, fique a vontade :)

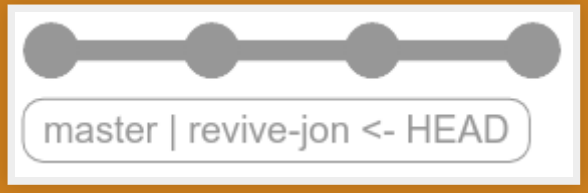
No fim das contas, a branch master é só uma *reference* que aponta pro último commit naquela branch

Speaker notes

cada branch tem um arquivo em `.git/refs/heads`, que são references. estes arquivos apontam para um commit, que é o último commit naquela branch. assim, quando damos `git checkout <nome da branch>`, o que estamos fazendo é tornando nosso "working directory" igual ao commit que está na reference daquela branch

Então, o `git branch revive-jon` criou a *reference* da revive-jon no mesmo commit que a *reference* da master apontava e o `git checkout` alterou o *HEAD*


master <- HEAD | revive-jon


master | revive-jon <- HEAD

Massa, agora vamos adicionar mais commits e mais branches!

revive-jon

```
$ git checkout revive-jon
$ unzip -o ../git-tut-pybr15-files/04files.zip
$ git add index.html
$ git commit -m "Adiciona ressurreição"
$ unzip -o ../git-tut-pybr15-files/05files.zip
$ git add index.html aliados.html
$ git commit -m "Adiciona zona norte toda unida"
```

dany-rainha-resto-nadinha

```
$ git checkout master
$ git checkout -b dany-rainha-resto-nadinha
$ unzip -o ../git-tut-pybr15-files/06files.zip
$ git add dany.html
$ git commit -m "Adiciona player two"
$ unzip -o ../git-tut-pybr15-files/07files.zip
$ git add dany.html
$ git commit -m "Modifica estado do exército"
```

`git checkout -b dany-rainha-resto-nadinha?`

Speaker notes

Este é um atalho que faz o `git branch` e o `git checkout` ao mesmo tempo

OK. Mais branches!

night-king

```
$ git checkout master
$ git checkout -b night-king
$ unzip -o ../git-tut-pybr15-files/08files.zip
$ git add inverno.html index.html
$ git commit -m "Adiciona player three"
```

bran

```
$ git checkout master
$ git checkout -b bran
$ unzip -o ../git-tut-pybr15-files/09files.zip
$ git add bran.html
$ git commit -m "Adiciona alguma coisa que Bran fez"
```

E agora se eu quiser introduzir alguma dessas variações na branch principal?

git merge

Antes de falar de `git merge`, vamos para a master

```
$ git checkout master
```

O que desejamos fazer é mesclar na master os commits feitos na `revive-jon`

Para isso vamos executar

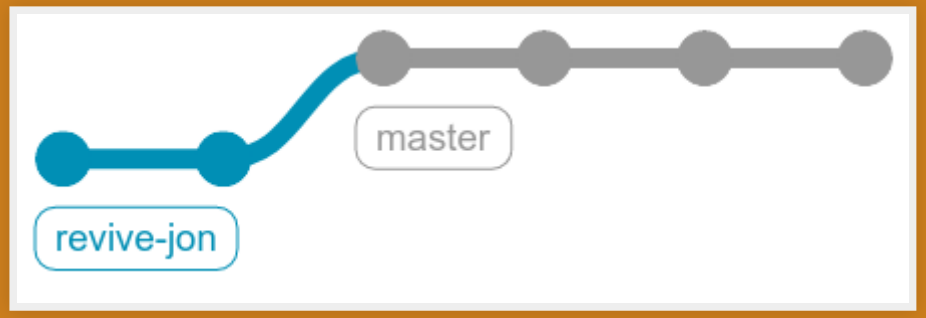
```
$ git merge revive-jon
```

Ok... O que o `git merge` fez...?

Aqui pode ser visto que há uma mensagem mencionando um *fast-forward*

```
Updating 964d3c2..7829a8f
Fast-forward
 aliados.html | 10 ++++++++++
 index.html   |  7 ++++++-
 2 files changed, 16 insertions(+), 1 deletion(-)
 create mode 100644 aliados.html
```

Como funciona um merge *fast-forward*:



Speaker notes

isso quer dizer que os commits da branch que está sendo mesclada foram feitos diretamente após a head da master, então o que foi feito foi apenas apontar a HEAD da master para a head da branch que está sendo mesclada

E se eu quiser mesclar a dany-rainha-resto-nadinha agora?

```
$ git merge dany-rainha-resto-nadinha
```

Agora pode ser visto que não foi possível fazer um *fast-forward*

```
Merge made by the 'recursive' strategy.
 dany.html | 10 ++++++++++
 1 file changed, 10 insertions(+)
 create mode 100644 dany.html
```

O que foi feito na verdade foi um merge `three-way`

Speaker notes

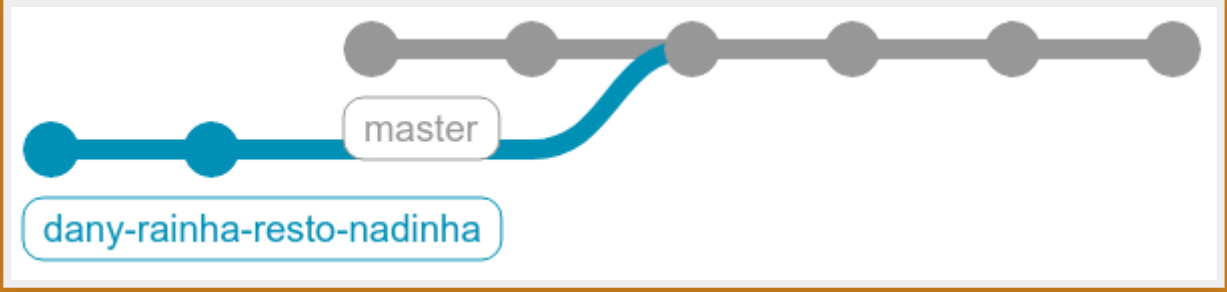
como os commits dessa branch não estavam diretamente após os commits da master, então não podemos simplesmente mover a HEAD da master para a HEAD desta branch sem perder commits. então o que é feito é que um "merge commit" é criado

- **merge commit**:
  - É um *commit object* com uma propriedade interessante a mais: ele possui mais de um ancestral, que são os commits head das branches que estão sendo mescladas

```
commit bbf06aada4ff5e79e01ecfa10941f34df779d95b
tree 8252ab52a2250ea9fb16cbcaf627979d88435248
parent 7829a8f0c01a81d20cbc428b0ee86019ba61db16
parent 2d6c47a2dbfdce00c31a7bec93542a8fdf525b09
author Giulio Carvalho <gcc@cin.ufpe.br> 1571808327 -0300
committer Giulio Carvalho <gcc@cin.ufpe.br> 1571808327 -0300

    Merge branch 'dany-rainha-resto-nadinha'
```

Como funciona um merge *three-way*



Speaker notes

Depois de o merge commit ser criado, a head da master é movida para o merge commit criado

E se modificações fossem feitas no mesmo trecho do mesmo arquivo nas branches que estou tentando mesclar?

CONFLITOS!

```
$ git merge night-king
```

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

Quando o git tem problemas para decidir qual modificação deve persistir, ele joga a bola pra o desenvolvedor

Speaker notes

mensagem de conflito aparecem e os arquivos conflituosos são mostrados, então o merge commit além de apontar pros dois ancestrais também carrega consigo a resolução dos conflitos

Queremos manter as duas modificações, e depois temos que dar o commit

```
$ unzip -o ../git-tut-pybr15-files/10files.zip
$ git add index.html
$ git commit
```

Pronto, agora vamos para o temido REBASE

git rebase

Se formos para a última branch que criamos e formos dar um `git merge` nela, um merge commit seria criado, mas podemos evitar isso usando `git rebase`

```
$ git checkout bran
$ git rebase master
```

Como funciona o rebase:

Speaker notes

A ideia do `git rebase` é mudar o ponto de divergência de uma branch para outro ponto. usando `git rebase master` o que estamos fazendo é tentando mudar o ponto de divergência para o commit que a HEAD da master aponta

com isso, todos os commits que fizemos na branch atual serão objetos diferentes por efeito dominó

após aplicar o rebase, um git merge pode ser aplicado e um fast-forward será realizado com sucesso

Agora podemos usar o `git merge` e teremos um *fast-forward* :)

```
$ git checkout master
$ git merge bran
```

Como aplicar todo esse jiu gitsu num ambiente colaborativo??

dica: muitos merges

Vamos criar um repositório no github e adicioná-lo como remoto

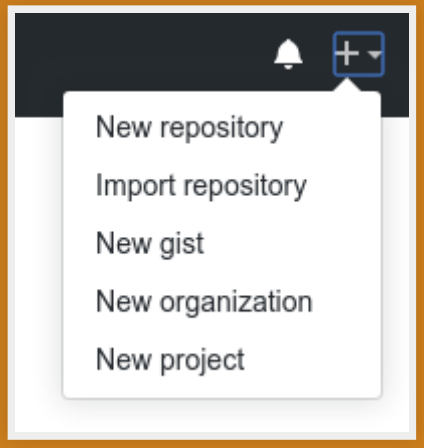
Um remoto é apenas outro repositório git.

Nada mais, nada menos

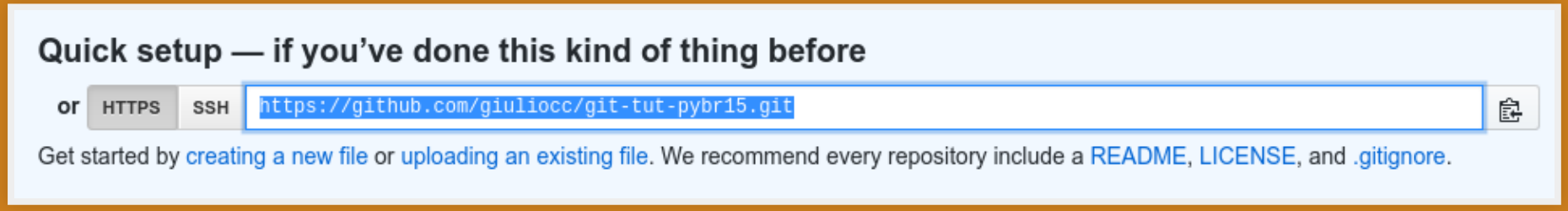
Speaker notes

é possível interagir com outros repositórios git através de dois comandos que iremos ver, mas antes vamos ver como adicionamos o remoto

Vamos ao Github criar um repositório novo


New repository
Import repository
New gist
New organization
New project

Vamos copiar o endereço HTTPS do repositório



E agora vamos criar o remoto

```
$ git remote add origin <endereço do repositório>
```

Speaker notes

origin é um apelido que damos ao remoto, podia ser qualquer outro, mas é meio que uma convenção, igual a "master"

Para vermos o remoto que adicionamos, podemos executar

```
$ git remote
```

```
origin
```

Speaker notes

antes de adicionar o remoto, esse comando iria mostrar uma tela vazia, que é meio sem graça de ver, né?

Temos um remoto, e ele está vazio, o que podemos fazer com ele?

Podemos criar uma master no remoto

git push

O `git push` cria uma branch local no repositório remoto

Então o que acontece quando executamos o `git push`?

```
$ git push origin master
```

O git irá verificar a diferença entre a história da branch local e da branch remota e concatená-la

Speaker notes

Se as histórias divergirem, um erro será levantado e o push não será completado

E pra pegar algo do remoto?

git fetch

Para vermos como o `git fetch` funciona, vamos simular uma colaboração

```
$ git checkout -b relaxa
$ unzip -o ../git-tut-pybr15-files/11files.zip
$ rm aliados.html inverno.html
$ git add index.html aliados.html inverno.html
$ git commit -m "Remove existência do Príncipe Prometido"
$ git push origin relaxa
$ git checkout master
$ git branch -D relaxa
```

Agora o remoto tem uma branch nova, e nós "não a temos localmente" por conta do `git branch -D`

Speaker notes

não a temos? foi tudo excluído? perdemos o commit? não, ele continua lá, sem pertencer a nenhuma branch, o chamado "dangling commit"

Então, vamos adicionar essa "colaboração" ao nosso projeto

```
$ git fetch origin relaxa
```

O `git fetch` criou uma branch no nosso repositório chamada origin/relaxa, podemos vê-la executando

```
$ git branch -r
```

```
  origin/master
  origin/relaxa
```

Speaker notes

`git branch -r`, diferentemente do `git branch` lista as branches ligadas a repositórios remotos

Mas ela não aparece lá se executarmos

`git branch`

Para isso, temos que criar a nova branch e executar um `git merge`

```
$ git checkout -b relaxa
$ git merge origin/relaxa
```

Pronto, agora temos a colaboração como uma branch local

Mas pera, temos um atalho pra isso

```
git pull
```

O `git pull` faria os mesmo passos que executamos agora em apenas um comando

Ao invés de

```
$ git fetch origin relaxa
$ git checkout -b relaxa
$ git merge origin/relaxa
```

Fazemos

```
$ git pull origin relaxa
```

E o resultado seria o mesmo

Por hoje é isso pessoal :)

Esse material foi baseado em...

Slides feitos com:

github.com/hakimel/reveal.js

Gráficos feitos com:

github.com/nicoespeon/gitgraph.js

Alguma pergunta?

Curiosidade?

Dúvida existencial?

Contatos:

@giuliocc

GitHub ou Telegram

PERGUNTAS EXTRAS!!

Como funciona uma release do GitHub?

O que acontece quando alguém modifica os commits no repositório remoto central que outros desenvolvedores estão trabalhando?