

Controle de versão

Fábio José Ayres

Sumário

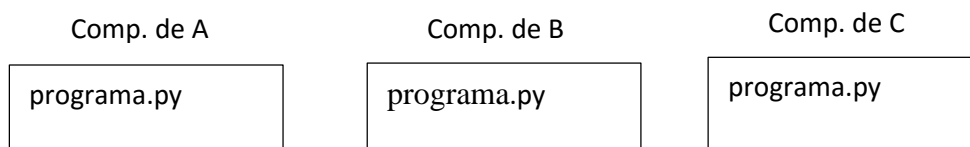
Conceitos	1
Porque usar controle de versão?	1
Sistemas de controle de versão	4
Gerenciamento de mudanças (<i>tracking changes</i>)	5
Tutorial	7
Preliminares	7
Registrando-se no GitHub	7
GitHub Desktop	9
Criar um repositório	9
Adicionando colaboradores	11
Clonando o repositório remoto na sua máquina	13
Ciclo regular de trabalho	15
Gerenciando conflitos	17
Conclusão	20

Conceitos

Porque usar controle de versão?

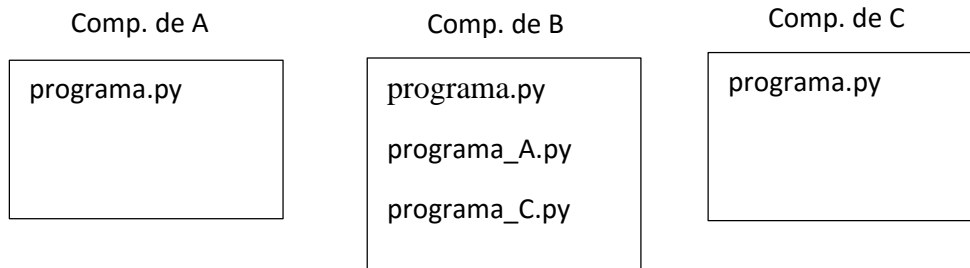
Imagine que um grupo de alunos vai desenvolver um projeto de software em grupo para a matéria de Design de Software (por exemplo, um exercício programa? Um projeto final? #ficadica).

O grupo consiste de três alunos: Astrogibalda Afobada, Baracobama Boêmio, e Cacá Confuso. Após dividir as tarefas do *sprint* entre si, o grupo começa a desenvolver o código. Afobada já começou a escrever um módulo e manda por e-mail para Boêmio e Confuso:



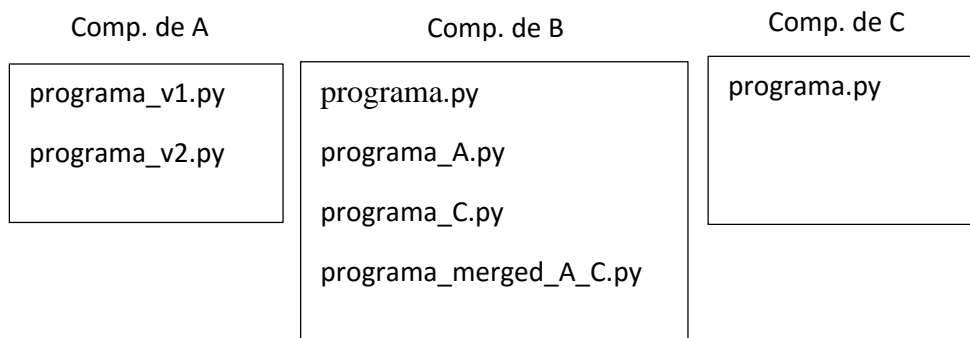
Afobada e Confuso continuam avançando no desenvolvimento, Boêmio vai trabalhar mais tarde. Quando chega a hora de Boêmio trabalhar, ele manda e-mail para Afobada e Confuso

para pedir a versão mais atual do programa:

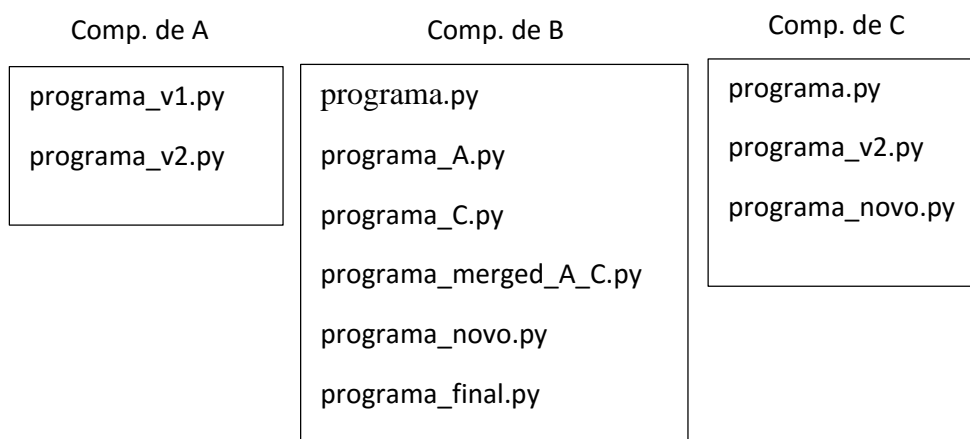


Boêmio tem que juntar as duas versões. Ele percebe que Afobada e Confuso acabaram desenvolvendo algumas coisas em duplicata (*codaram* a mesma função, por exemplo, culpa do Confuso!). E agora? Passa da meia noite e seus colegas estão dormindo! Ele emenda os dois códigos da maneira que ele acha melhor e continua trabalhando.

Enquanto isso, Afobada não espera o *merge* do código para continuar desenvolvendo: ela faz uma cópia do código original e continua trabalhando.



Agora, Confuso recebe a cópia v2 de Afobada e percebe a duplicação de código, fica bravo com Afobada, e resolve manter a sua própria versão do código, apenas adicionando a parte nova que Afobada escreveu. Ele manda sua nova versão do programa para Boêmio, que tenta juntar todas as versões.



Porém o grupo desenvolveu todo o trabalho sem testes adequados, e agora foi descoberto um erro. Durante o *debug*, Afobada e Boêmio trabalharam juntos para consertar o erro e acabaram gerando várias versões. Confuso trabalhou num requisito novo do projeto.

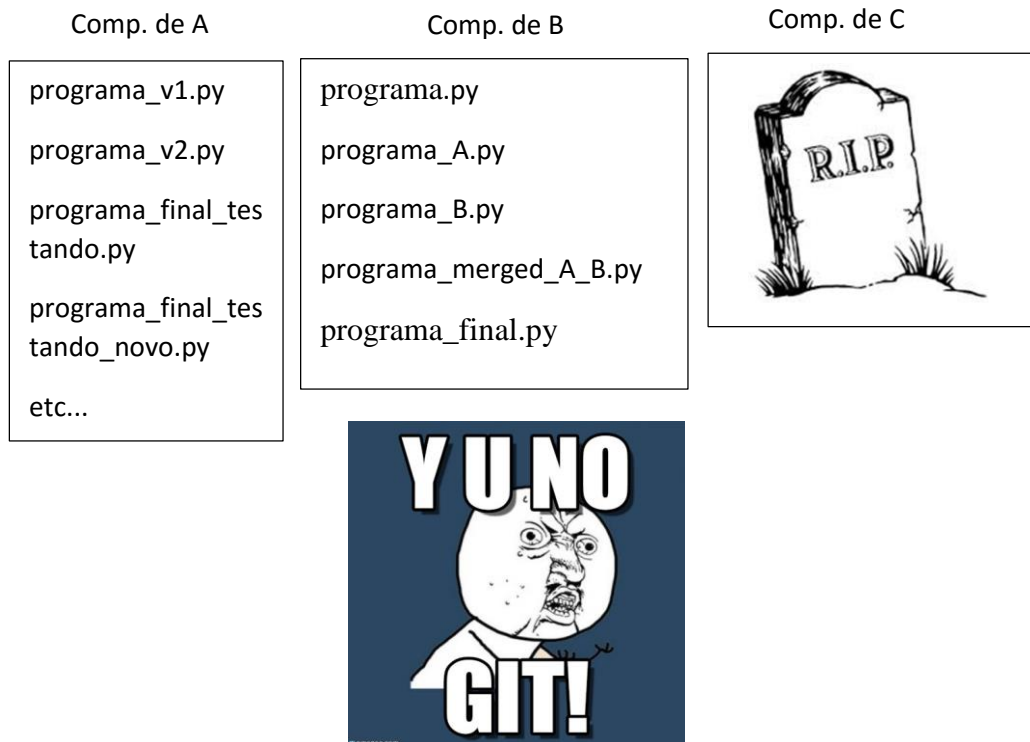
Comp. de A	Comp. de B	Comp. de C
programa_v1.py programa_v2.py programa_final_testando.py programa_final_testando_novo.py etc...	programa.py programa_A.py programa_C.py programa_merged_A_C.py programa_novo.py programa_final.py	programa.py programa_v2.py programa_novo.py programa_20151203.py

Por fim Afobada e Boêmio mandam a versão corrigida para Confuso, que com muito esforço consegue juntar as versões. Olha a confusão de arquivos!

Comp. de A	Comp. de B	Comp. de C
programa_v1.py programa_v2.py programa_final_testando.py programa_final_testando_novo.py etc...	programa.py programa_A.py programa_C.py programa_merged_A_C.py programa_novo.py programa_final.py	programa.py programa_v2.py programa_novo.py programa_20151203.py programa_entrega.py



Mas no dia de entregar o projeto, o HD de Confuso queimou.



Muitos dos problemas deste cenário podem ser diminuídos usando um sistema de controle de versão. Tais sistemas facilitam o gerenciamento de documentos (principalmente código-fonte) que são construídos de modo incremental (como uma sequência de pequenas modificações), e muitas vezes em grupo.

Sistemas de controle de versão

Sistemas de controle de versão podem ser aplicados à gestão de documentos em geral. Você talvez já tenha usado controle de versão: por exemplo, o mecanismo de revisão do Microsoft Word. Podemos usar controle de versão para arquivos de desenho técnico, para documentos legais, para bulas de remédio, para qualquer coisa! Mas é no gerenciamento de código-fonte que o controle de versão se sobressai: é absolutamente impossível conceber o desenvolvimento moderno de software (e tem sido assim desde os anos 80) sem esta ferramenta.

O uso de sistemas de controle de versão é prática bem estabelecida no mercado onde quer que software seja desenvolvido, de start-ups à grandes empresas multinacionais, em todos os setores de tecnologia. Saber os conceitos de controle de versão e ser proficiente em algum sistema de controle de versão moderno é um requerimento básico para todos os engenheiros (seja Mecânico, Mecatrônico, ou de Computação).

Em nosso curso, usaremos o software de controle de versão chamado *Git*, criado por Linus Torvalds em 2005. Linus é o criador do *kernel* do sistema operacional aberto *Linux*, no qual se baseiam as várias distribuições de sistema operacional que emprestam o mesmo nome (como Ubuntu Linux, RedHat Linux, Debian Linux, Linux Mint, etc). O código-fonte do *kernel* do Linux é extenso, resultado da colaboração de vários indivíduos mundo afora que gostaram do sistema operacional criado por Linus, que na época era um aluno de computação finlandês de 21 anos apenas!

Saber usar o Git é uma habilidade bastante valorizada!

Linked in®O que é c



Software Development Engineer

Intel Corporation

Brazil-Brazil, Campinas

 Anunciada há 22 dias  462 visualizações

[Candidate-se no site da empresa](#)

Descrição da vaga

The Open Source Technology Center (OTC) is searching for software developers who are interested in working on innovative open source software solutions across several segments, including but not limited to Drones, Robots and cognitive capabilities including vision and speech

...

Qualifications

Minimum:

- Bachelor's Degree in Computer Science or other technical degree. Master Degree in Computer Science will be a plus.
- Proficiency in at least one of the following languages: C/C++/Python
- Knowledge of debugging techniques and tools
- English proficiency (advanced)

Preferred:

- Existing contributions to an open-source project
- Knowledge of low-level I/O protocols such as I2C and SPI
- Knowledge of a source code revision control system (e.g. Git)

- 1-2 years experience in any related field will be a plus

Setor

Hardware e Software

Tipo de trabalho

Tempo Integral

Nesta apostila discutiremos o Git de modo bastante superficial. Para conhecer melhor o Git, veja este tutorial em <https://www.atlassian.com/git/tutorials/>

Outros sistemas de controle de versão incluem *Subversion*, *Mercurial*, e *CVS*. Algumas empresas de grande porte, como o Google, desenvolveram seus próprios sistemas de controle de versão. Mas todos estes sistemas compartilham os mesmos princípios básicos de operação, do ponto de vista do desenvolvedor de software.

Gerenciamento de mudanças (*tracking changes*)

Um *repositório (repository)* é um conjunto de arquivos que está sob controle do sistema de controle de versão. Geralmente trata-se de um diretório e de todos os arquivos e subdiretórios abaixo do mesmo. O repositório pode ser local (um diretório no seu computador) ou remoto (o repositório existe em um computador remoto, em um servidor da sua empresa ou no *cloud*).

Para começar a desenvolver código usando um sistema de controle de versão você pode:

- Criar um repositório vazio;

- Criar um repositório a partir de um diretório pré-existente;
- *Clonar* um repositório existente;

Ao começar um projeto devemos criar o repositório. No caso de equipes de desenvolvimento, geralmente cabe ao líder da equipe a criação do repositório e a definição de quais indivíduos terão acesso ao mesmo. Este repositório inicial é criado em um servidor de rede, ou em serviços online de armazenamento de repositórios. Neste curso usaremos o serviço online GitHub (<https://github.com/>).

Os membros da equipe então irão clonar o repositório em suas respectivas máquinas. Estes repositórios criados por clonagem são tão centrais como o repositório inicial, é por mera convenção que usaremos o repositório inicial como o repositório oficial do grupo de trabalho. Chamaremos o repositório inicial de *repositório remoto*, e os repositórios clonados de *repositórios locais*.

Os vários repositórios agora serão mantidos em sincronia pelo sistema de controle de versão, através de atividades específicas que passarão a fazer parte da rotina dos desenvolvedores. Tais atividades, como *check-out* (ou *pull*, ou *fetch*), *commit*, *check-in* (ou *push*), *branch*, entre outras, serão discutidas mais tarde.

O ciclo regular de trabalho de um membro da equipe (incluindo o líder), após clonar o repositório inicial, será:

- *Fetch*: baixa as últimas mudanças (em jargão da área, os últimos *commits*).
- *Merge*: quando solicitado a fazer o merge, o Git tenta adicionar estas novas mudanças ao estado atual do seu repositório (ou seja, tenta adicionar os novos códigos escritos pelos seus colegas ao código que já está na sua máquina).
 - *Pull*: o Git oferece o comando *pull*, que é equivalente a um *fetch* seguido de um *merge*.
- Resolução de conflitos: o *merge* não será bem-sucedido enquanto houverem *conflitos* entre o seu código e os novos *commits*. Geralmente o Git é bem esperto e consegue integrar o novo código sozinho, mas às vezes ele fica confuso e não sabe como fazer a integração automática. Nestes casos, o Git mostrará onde não foi possível integrar o código automaticamente e pedirá que você conserte manualmente a integração.
 - *Pro-tip*: mesmo que o Git integre o novo código automaticamente, vale a pena rever manualmente todos os arquivos modificados que tenham algo a ver com o código no qual você estava trabalhando logo antes do merge!
- Enquanto você estiver desenvolvendo uma pequena nova *feature*:
 - Escrever novo código com uma pequena mudança.
 - *Commit*: Uma vez que tudo esteja funcionando bem, gravar no seu repositório Git local essa sua pequena mudança.
 - *Pro-tip*: Faça vários pequenos commits, é para o seu próprio bem!
 - De tempos em tempos, repita o ciclo “pull”-“resolver conflitos” para garantir que você não está escrevendo código novo em *codebase* desatualizada.
- Se tudo estiver bem, e faça um último ciclo “pull”-“resolver conflitos”
- *Push*: envia seus *commits* para o repositório remoto

Agora é celebrar, vai tomar um café, uma água! É agora que você começa a receber os e-mails dos seus colegas exaltando a beleza e praticidade da feature que você desenvolveu!

Ou então a sua equipe não tem testes automatizados, e a sua feature quebrou o trabalho dos seus colegas! Neste momento você pode reverter as suas mudanças (Lembra da dica de fazer pequenos commits? De nada!) e fazer um novo *push* com essa atividade. (No jargão da área: dar um *revert*).

MEGA-ULTRA-PRO-TIP: É por isso que não se faz push crítico na sexta-feira de noite! Não apenas você terá que ir trabalhar no final de semana para consertar a cag..., ops, atividade incorreta que você fez, como terá que fazer isso sem a ajuda dos colegas (ou mesmo do maldit..., er, do colaborador que te mandou á ..., emm..., levantou a questão do código defeituoso).

Vamos recapitular o que temos até o momento:

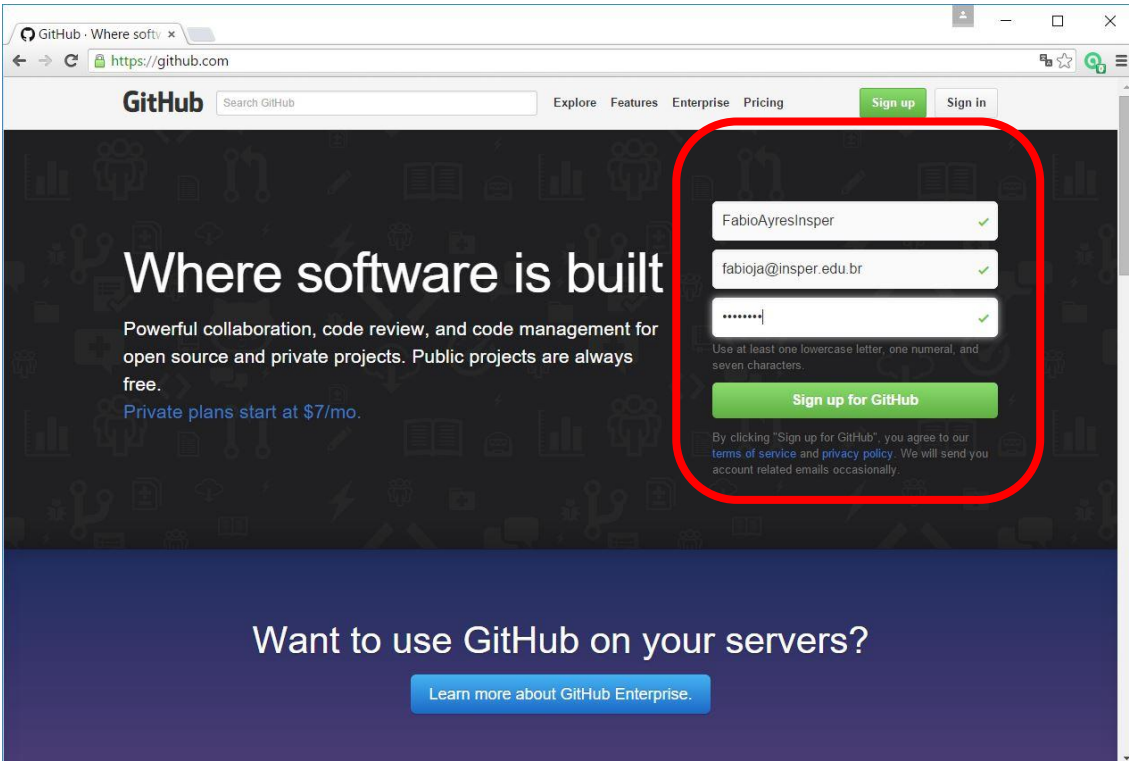
- O líder criou um repositório novo. Neste curso usaremos o serviço online *GitHub* para isso. Chamaremos este repositório inicial de repositório *remoto*.
- Todos clonam o repositório remoto, gerando os vários repositórios *locais*.
- Estabelece-se o ciclo regular de trabalho: *fetch-merge-resolve-commit-push*.

Tutorial

Atividades preliminares

Crie uma conta no GitHub: <https://github.com/>. Com isso você poderá criar seus próprios repositórios Git remotos, acessíveis de qualquer lugar, com backups e disponibilidade garantidos! Nada mal para um serviço gratuito, não?

Registrando-se no GitHub



The screenshot shows the GitHub homepage in a web browser. The URL bar displays 'https://github.com'. The page features the GitHub logo, a search bar, and navigation links for 'Explore', 'Features', 'Enterprise', and 'Pricing'. A 'Sign up' button is visible in the top right. The main content area has the heading 'Where software is built' and a description of GitHub's services. A sign-up form is highlighted with a red circle, containing three input fields: a username 'FabioAyresInsper', an email 'fabioja@insper.edu.br', and a password field with masked characters. Below the password field is a green 'Sign up for GitHub' button. A small disclaimer at the bottom of the form states: 'By clicking "Sign up for GitHub", you agree to our terms of service and privacy policy. We will send you account related emails occasionally.'

Observe que o site já selecionou para você a opção de conta gratuita:

The screenshot shows the GitHub 'Welcome to GitHub' page for user @FabioAyresInsper. The page is at the 'Step 2: Choose your plan' stage. A table titled 'Choose your personal plan' lists five options: Large (\$50/month, 50 private repositories), Medium (\$22/month, 20 private repositories), Small (\$12/month, 10 private repositories), Micro (\$7/month, 5 private repositories), and Free (\$0/month, 0 private repositories). The 'Free' plan is highlighted with a red box and its button is labeled 'Chosen'. To the right, a box titled 'Each plan includes:' lists features: Unlimited collaborators, Unlimited public repositories, Free setup, HTTPS Protection, Email support, and Wikis, Issues, Pages, & more. Below the table, there is a checkbox for 'Help me set up an organization next' and a 'Finish sign up' button, which is also highlighted with a red box.

Plan	Cost (view in BRL)	Private repositories	
Large	\$50/month	50	Choose
Medium	\$22/month	20	Choose
Small	\$12/month	10	Choose
Micro	\$7/month	5	Choose
Free	\$0/month	0	Chosen

Charges to your account will be made in US Dollars. Converted prices are provided as a convenience and are only an estimate based on current exchange rates. Local prices will change as the exchange rate fluctuates. Don't worry, you can cancel or upgrade at any time.

☐ Help me set up an organization next
Organizations are separate from personal accounts and are best suited for businesses who need to manage permissions for many employees. [Learn more about organizations.](#)

[Finish sign up](#)

Vá ao seu leitor de e-mails e confirme seu email:

The screenshot shows an email from GitHub to Fabio Jose Ayres. The email header includes the GitHub logo and the time 13:56. The subject is '[GitHub] Please verify your email address.' The body of the email starts with 'Hi @FabioAyresInsper!' and asks the user to verify their email address (fabioja@insper.edu.br) to access all of GitHub's features. A red box highlights the 'Verify email address' link. Below this, there is a link to the user's email settings page and a note about receiving the email. The GitHub logo is at the bottom.

Hi @FabioAyresInsper!

Help us secure your GitHub account by verifying your email address (fabioja@insper.edu.br). This lets you access all of GitHub's features.

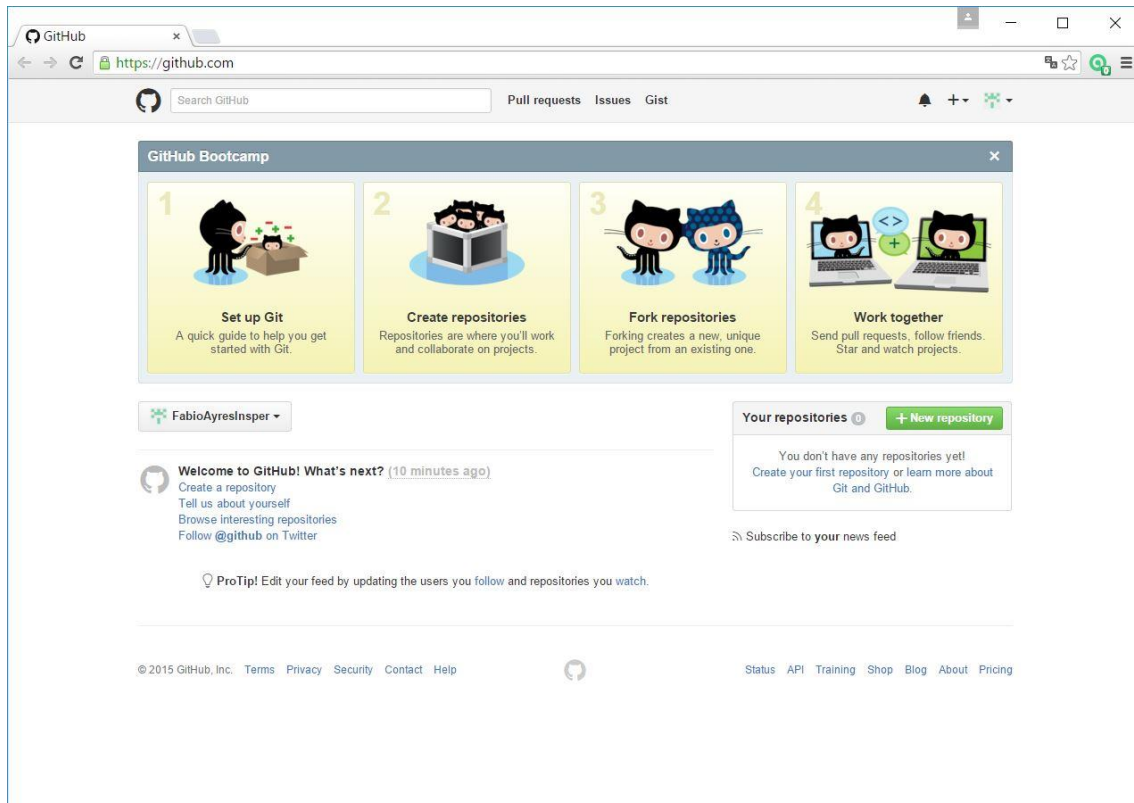
[Verify email address](#)

Button not working? Paste the following link into your browser:
<https://github.com/users/FabioAyresInsper/emails>

You're receiving this email because you recently created a new GitHub account or added a new email address. If this wasn't you, please ignore this email.

GitHub

Pronto, você agora tem o seu próprio espaço no GitHub!



GitHub Desktop

Windows:

Instale “GitHub Desktop”: <https://desktop.github.com/> . Trata-se de um programa Git completo, com o qual você poderá usar todos os benefícios de controle de versão em seu próprio desktop, off-line, flexível, mais uma interface de usuário que permite trabalhar com o seu repositório Git local e também interagir com seu repositório Git remoto

Mac:

Instale o programa “git” na sua máquina. Como eu não tenho Mac, não sei como fazer isso exatamente. Segundo este site – <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git#Installing-on-Mac> – parece que basta tentar rodar git na linha de comando que o sistema já percebe se tem o programa ou não, e passa instruções de instalação.

Linux:

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

Criar um repositório

Voltemos à página do github. Na barra superior, clique no ‘+’ e crie um novo repositório.

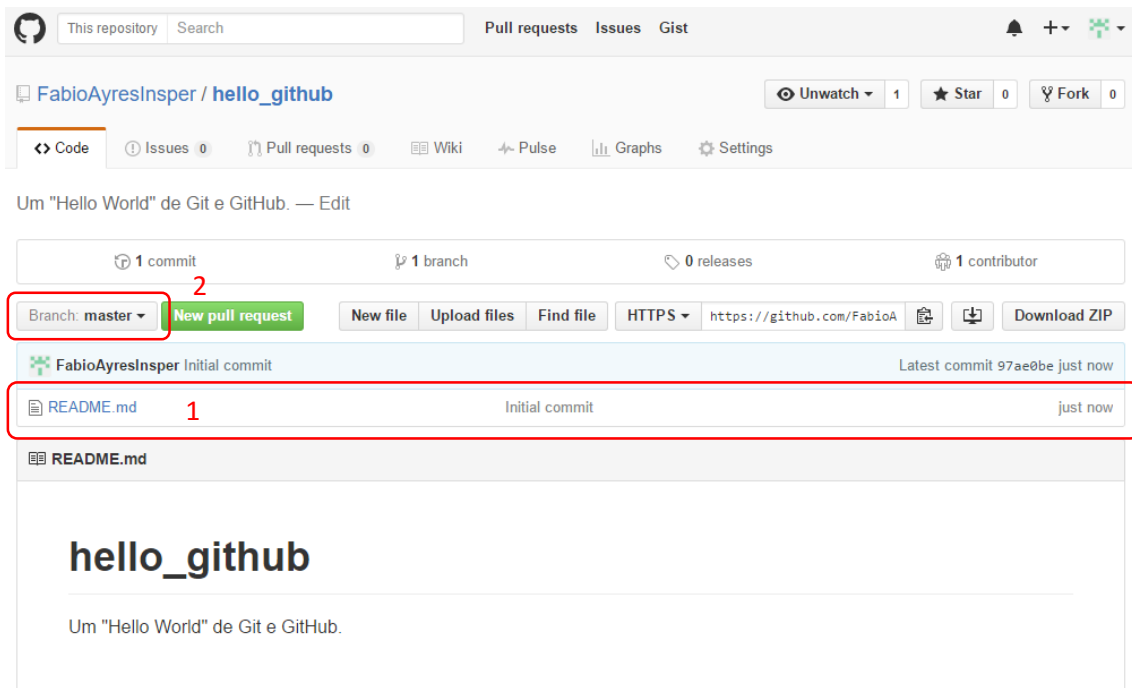


Você será redirecionado para a tela de criação de repositório.

The image shows the 'Create a new repository' page on GitHub. It includes a header with the GitHub logo, search bar, and navigation links. The main content area is titled 'Create a new repository' and contains a form with several fields and options. The form is annotated with red boxes and numbers 1 through 4. 1. A red box around the 'Repository name' input field. 2. A red box around the 'Description (optional)' input field. 3. A red box around the 'Public' radio button and its description. 4. A red box around the 'Initialize this repository with a README' checkbox and its description. Below the form, there are two dropdown menus for 'Add .gitignore' and 'Add a license', and a green 'Create repository' button.

1. Dê um nome ao seu repositório. Este nome será usado por todos que quiserem baixar o seu repositório (incluindo você mesmo!).
2. Escreva uma descrição breve do seu repositório, para ajudar outros desenvolvedores a localizar o seu trabalho no site do GitHub.
3. Marque o seu repositório como “Public”. Trata-se de uma característica do GitHub (e não do Git *per se*): repositórios públicos são visíveis a todos, enquanto repositórios privados são visíveis apenas a convidados. O GitHub disponibiliza apenas repositórios públicos para usuários de contas gratuitas. Se você tem a necessidade de repositórios privados (por exemplo, para guardar o código-fonte da sua *start up*), deve obter uma conta paga.
4. Marque este *checkbox* para criar um arquivo *README*. Ter um arquivo cujo nome convida o usuário à leitura do mesmo é uma prática antiga em computação. Tal arquivo tem caráter documental e, por estar no mesmo diretório do projeto, serve como um ponto de entrada de documentação para aqueles usuários que preferem abortar um novo projeto diretamente pela raiz do código-fonte.

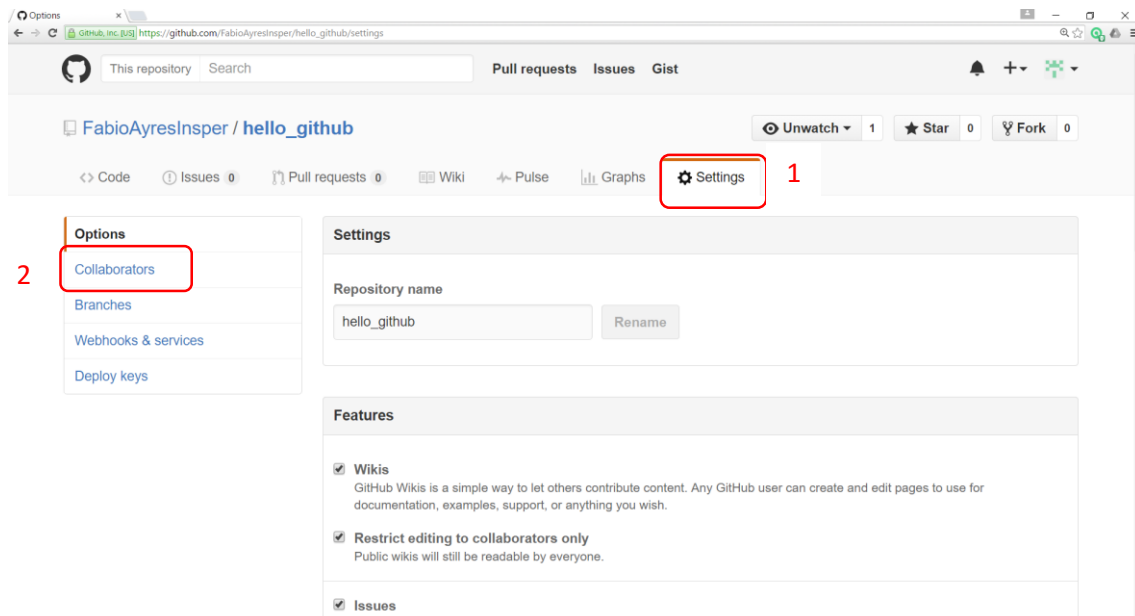
Pronto, seu repositório está criado no GitHub!



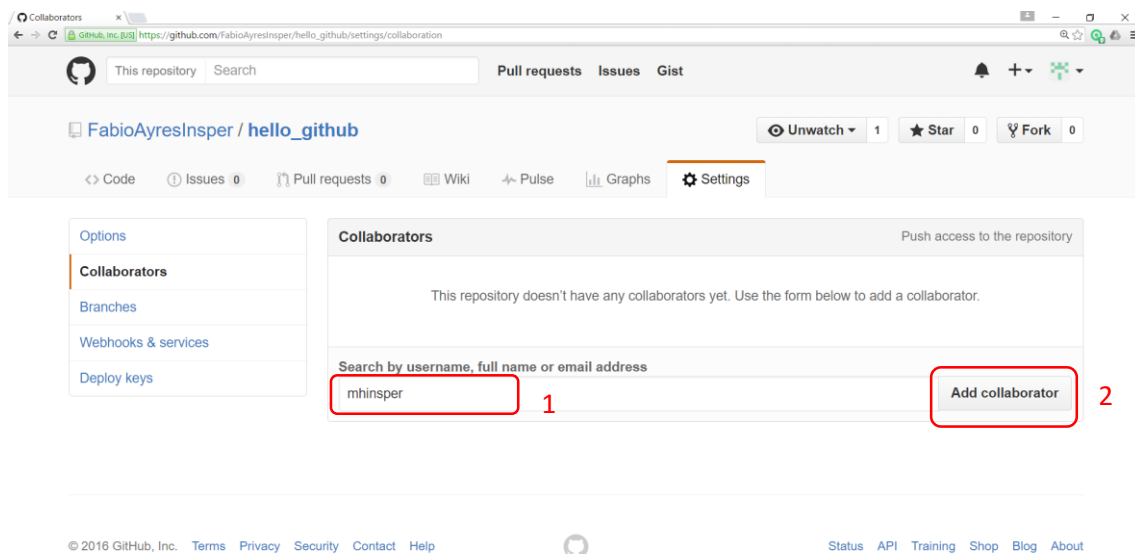
1. Observe que o repositório já vem com um *commit* feito, onde foi criado com um arquivo para você, o 'README.md'. Este tipo de arquivo está escrito em linguagem Markdown, um formato de arquivo texto que representa formatação de modo simples. Você já usou Markdown em ModSim!
2. Estamos observando o estado atual do *branch* "master". Branches são linhas paralelas de desenvolvimento de código, que eventualmente podem ser reintegradas. A linha principal é sempre chamada de master. Não abordaremos branching nesta apostila, infelizmente, pois este texto é uma introdução básica apenas.

Adicionando colaboradores

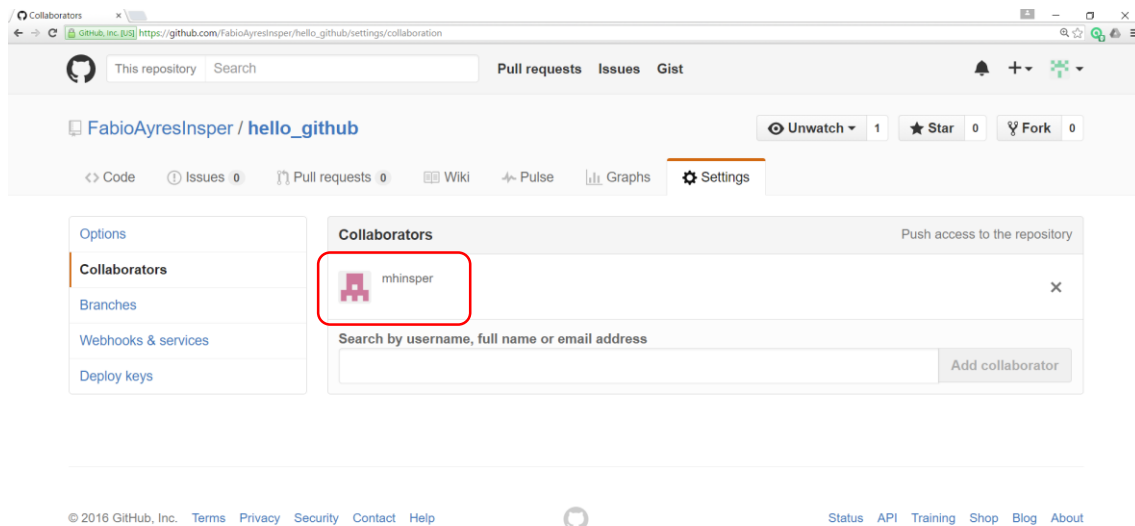
Agora que temos o repositório criado no GitHub, vamos adicionar um colaborador:



1. Clique na aba "Settings"
2. Clique em "Collaborators"



1. Adicione o username GitHub ou o email do seu colaborador
2. Clique em "Add collaborator"



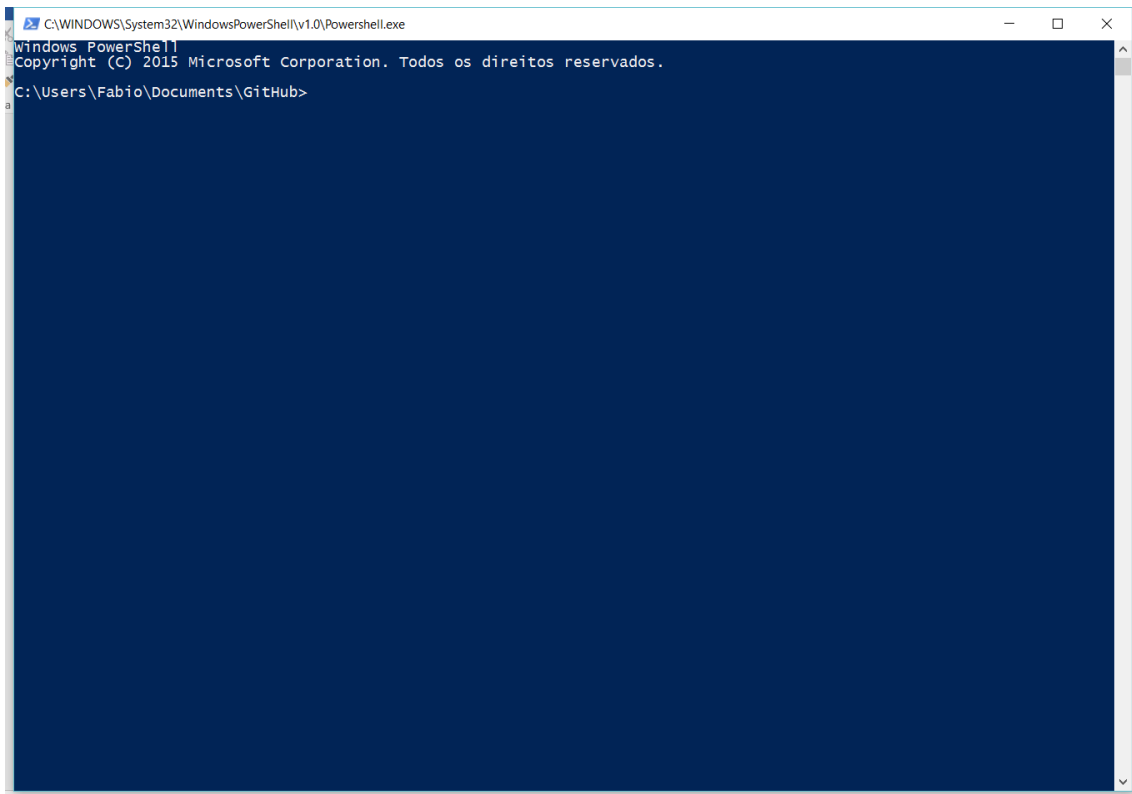
O colaborador foi adicionado, e ele poderá fazer *commits* neste repositório remoto!

Nota: como os repositórios são públicos, qualquer pessoa pode *clonar* ou fazer um *fork* do seu repositório remoto. Porém, somente os colaboradores podem fazer commits neste repositório.

Clonando o repositório remoto na sua máquina

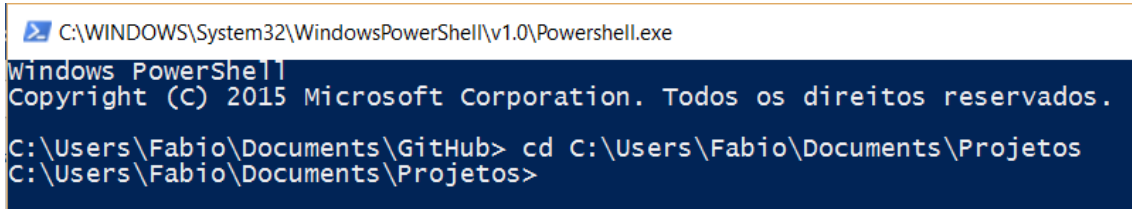
Neste momento, o que temos é um repositório no GitHub, com um commit apenas, no qual um arquivo foi adicionado. Agora precisamos clonar o repositório no seu próprio computador para que você possa trabalhar com os arquivos!

No Windows, abra o programa “Git Shell”. No Mac e Linux, abra uma terminal de linha de comando.



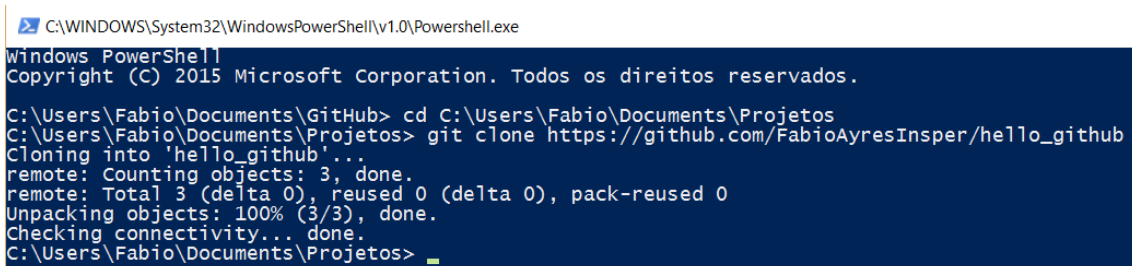
```
C:\WINDOWS\System32\WindowsPowerShell\v1.0\PowerShell.exe
Windows PowerShell
Copyright (C) 2015 Microsoft Corporation. Todos os direitos reservados.
C:\Users\Fabio\Documents\GitHub>
```

Com o comando `cd`, vá para o diretório onde você desenvolverá seu projeto.



```
C:\WINDOWS\System32\WindowsPowerShell\v1.0\PowerShell.exe
Windows PowerShell
Copyright (C) 2015 Microsoft Corporation. Todos os direitos reservados.
C:\Users\Fabio\Documents\GitHub> cd C:\Users\Fabio\Documents\Projetos
C:\Users\Fabio\Documents\Projetos>
```

Agora vamos clonar o repositório remoto aqui:



```
C:\WINDOWS\System32\WindowsPowerShell\v1.0\PowerShell.exe
Windows PowerShell
Copyright (C) 2015 Microsoft Corporation. Todos os direitos reservados.
C:\Users\Fabio\Documents\GitHub> cd C:\Users\Fabio\Documents\Projetos
C:\Users\Fabio\Documents\Projetos> git clone https://github.com/FabioAyresInesper/hello_github
Cloning into 'hello_github'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
Checking connectivity... done.
C:\Users\Fabio\Documents\Projetos>
```

Vamos entrar no diretório do projeto e verificar seu conteúdo:

```

C:\Users\Fabio\Documents\Projetos> cd .\hello_github
C:\Users\Fabio\Documents\Projetos\hello_github [master]> dir

Diretório: C:\Users\Fabio\Documents\Projetos\hello_github

Mode                LastWriteTime         Length Name
----                -
-a-----         14/04/2016      15:42             51 README.md

C:\Users\Fabio\Documents\Projetos\hello_github [master]>

```

Agora vamos começar a trabalhar!

Ciclo regular de trabalho

Vamos falar do ciclo regular de trabalho:

```

# Inicio da sessão de trabalho.

git pull

corrigir problemas de merge


# Parte principal da sessão de trabalho.

enquanto não acabei:

    trabalhar

    git commit


# Fechamento da sessão.

git pull

corrigir problemas de merge


git push

```

Neste ponto do tutorial nós acabamos de clonar um repositório, logo é evidente que nossa cópia está atualizada! Mas se esse não for o caso (ou seja, no caso geral), devemos antes de mais nada puxar (em inglês: *pull*) as últimas mudanças do repositório remoto.

```

C:\Users\Fabio\Documents\Projetos\hello_github [master]> git pull
Already up-to-date.
C:\Users\Fabio\Documents\Projetos\hello_github [master]> _

```

Se houverem mudanças no repositório remoto que entram em conflito com as mudanças locais, o git irá pedir que você resolva os conflitos antes de continuar. Retornaremos a esse ponto mais tarde.

Agora vamos trabalhar! Vamos fazer um arquivo hello.py que é um script Python que escreve "Hello World!" na tela. Eis a primeira versão:

```
print("Hello World!")
```

Vamos informar ao git que existe um novo arquivo que deve ser gerenciado por ele:

```
C:\Users\Fabio\Documents\Projetos\hello_github [master]> git add .\hello.py
C:\Users\Fabio\Documents\Projetos\hello_github [master +1 -0 -0]>
```

Note que o prompt agora indica que um arquivo foi adicionado ao *staging area*. Tudo aquilo que é adicionado ou modificado deve ser colocado nessa área fictícia de mudanças, até que o commit seja feito.

Vamos então fazer o commit.

```
C:\Users\Fabio\Documents\Projetos\hello_github [master +1 -0 -0]> git commit -a -m "Adicionei impressão da mensagem de s
audação."
[master c494c56] Adicionei impressão da mensagem de saudação.
1 file changed, 1 insertion(+)
 create mode 100644 hello.py
C:\Users\Fabio\Documents\Projetos\hello_github [master]>
```

Neste comando estamos dizendo ao git que registre as últimas mudanças. Note que temos que dar uma mensagem informativa sobre o que está contido nesse commit. Vamos ver agora o log de commits já feito.

```
C:\Users\Fabio\Documents\Projetos\hello_github [master]> git log
commit c494c56efa666884393c9f845d16e92a47c96a08
Author: Fabio Ayres <fabioja@insper.edu.br>
Date: Thu Apr 14 17:51:53 2016 -0300

    Adicionei impressão da mensagem de saudação.

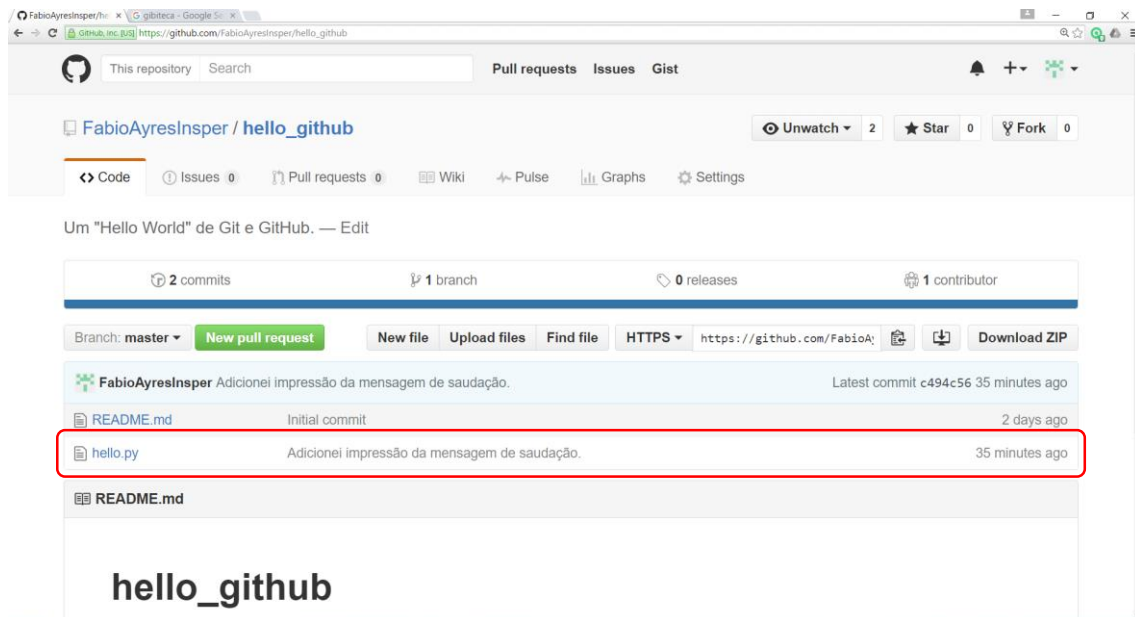
commit 97ae0be7674c9ccbeed7c3c76939bf997087de68
Author: FabioAyresInsper <fabioja@insper.edu.br>
Date: Tue Apr 12 17:52:20 2016 -0300

    Initial commit
C:\Users\Fabio\Documents\Projetos\hello_github [master]> _
```

Sem mais mudanças para o momento, vamos enviar esses novos commits para o repositório remoto. Primeiro fazemos um git pull para atualizar tudo, e depois fazemos um git push:


```
C:\Users\Fabio\Documents\Projetos\hello_github [master]> git pull
Already up-to-date.
C:\Users\Fabio\Documents\Projetos\hello_github [master]> git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 328 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/FabioAyresInspirer/hello_github
   97ae0be..c494c56  master -> master
C:\Users\Fabio\Documents\Projetos\hello_github [master]> _
```

Pronto, fizemos nosso primeiro round de trabalho no git! Eis o estado do repositório remoto:



Gerenciando conflitos

Vamos agora continuar nosso trabalho. Após o git pull inicial, alteramos o arquivo hello.py para o seguinte:

```
nome = input("Como você se chama? ")
print("Hello {0}!".format(nome))
print("Tenha um bom dia!")
```

Vamos ver como está nosso diretório de trabalho até o momento:

```
C:\Users\Fabio\Documents\Projetos\hello_github [master +0 ~1 ~0]> git status
On branch master
Your branch is up-to-date with 'origin/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:   hello.py

no changes added to commit (use "git add" and/or "git commit -a")
C:\Users\Fabio\Documents\Projetos\hello_github [master +0 ~1 ~0]> _
```

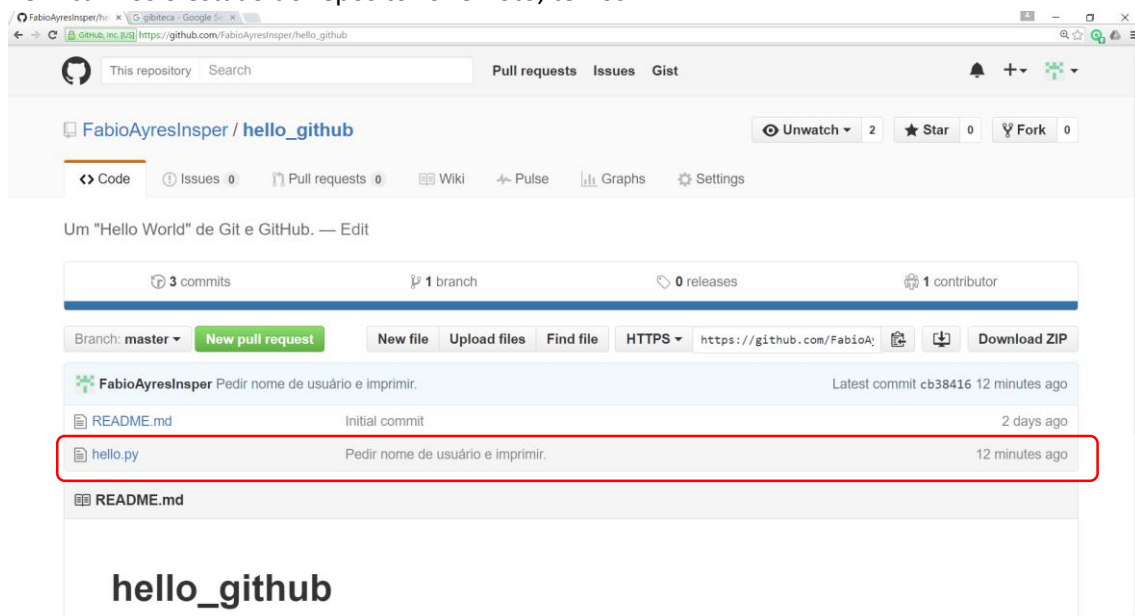
Podemos adicionar a ultima mudança ao *staging area* com um comando git add, ou simplesmente adicionar todas as modificações ao staging area no momento do commit com a flag "-a" do git commit, fica a seu critério o que fazer. Vamos então fazer o commit:

```
C:\Users\Fabio\Documents\Projetos\hello_github [master +0 ~1 ~0]> git commit -a -m "Ler nome de usuário e saudar adequadamente."
[master dcaecf2] Ler nome de usuário e saudar adequadamente.
1 file changed, 3 insertions(+), 1 deletion(-)
C:\Users\Fabio\Documents\Projetos\hello_github [master]> _
```

Pronto, agora vamos fazer um novo pull antes do push:

```
C:\Users\Fabio\Documents\Projetos\hello_github [master]> git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/FabioAyresInspirer/hello_github
c494c56..cb38416 master -> origin/master
Auto-merging hello.py
CONFLICT (content): Merge conflict in hello.py
Automatic merge failed; fix conflicts and then commit the result.
C:\Users\Fabio\Documents\Projetos\hello_github [master +0 ~0 ~0 !1 | +0 ~0 ~0 !2]>
```

Ops! Parece que o git não conseguiu fazer o merge desta vez! Temos um conflito em hello.py! Acho que alguém modificou o arquivo e fez um commit antes da gente! De fato, se verificarmos o estado do repositório remoto, temos:



Ao abrir o arquivo hello.py verificamos que o git marcou as regiões de conflito para nós:

```
<<<<<< HEAD
nome = input("Como você se chama? ")
print("Hello {0}!".format(nome))
print("Tenha um bom dia!")
=====
nome = input("Qual é o seu nome? ")

print("Hello {0}!".format(nome))
>>>>>> cb38416965ce3d14aa83477db1a4035aab9f1471
```

Temos que consertar o conflito e fazer um novo commit. Neste caso, para consertar o conflito, decidimos que vamos manter a linha de input() do código remoto (parte de baixo) e manter a nossa saudação (parte de cima). O código agora ficou assim:

```
nome = input("Qual é o seu nome? ")
print("Hello {0}!".format(nome))
print("Tenha um bom dia!")
```

Conforme recomendado na mensagem de erro anterior, consertamos o conflito e devemos fazer um novo commit:

```
C:\Users\Fabio\Documents\Projetos\hello_github [master +0 -0 -0 11 | +0 -0 +0 11]> git commit -a -m "Resolvendo o conflito."
[master b7dc62c] Resolvendo o conflito.
C:\Users\Fabio\Documents\Projetos\hello_github [master]>
```

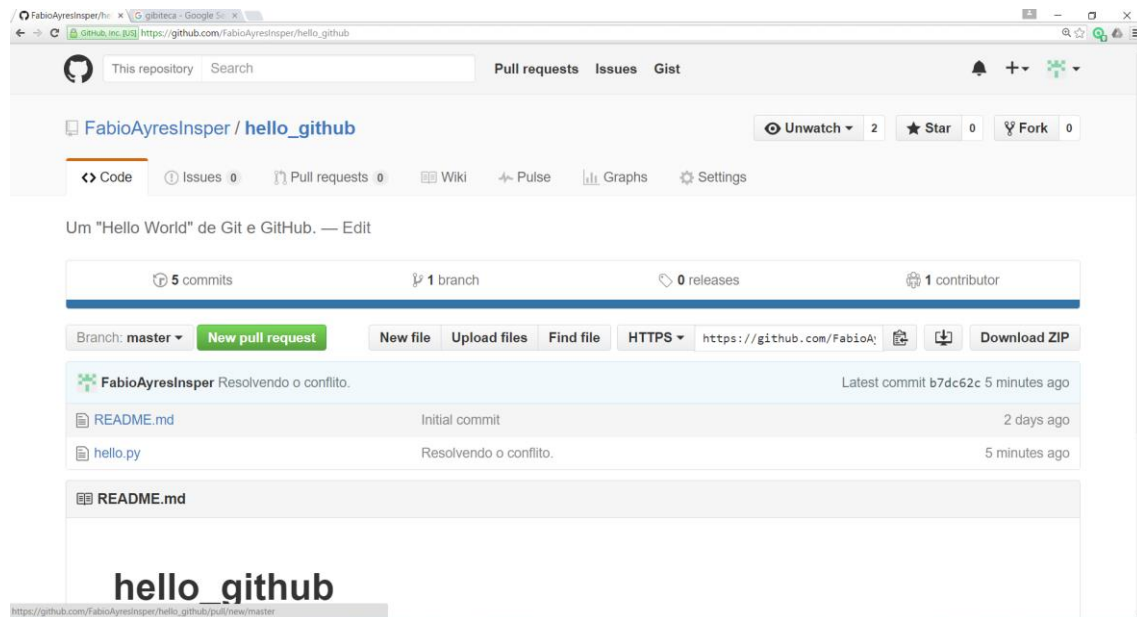
Agora podemos fazer o git pull de novo:

```
C:\Users\Fabio\Documents\Projetos\hello_github [master]> git pull
Already up-to-date.
C:\Users\Fabio\Documents\Projetos\hello_github [master]>
```

Funcionou! Podemos finalmente fazer o git push:

```
C:\Users\Fabio\Documents\Projetos\hello_github [master]> git push
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 767 bytes | 0 bytes/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To https://github.com/FabioAyresInspere/hello_github
   cb38416..b7dc62c  master -> master
C:\Users\Fabio\Documents\Projetos\hello_github [master]>
```

Pronto, consertamos o conflito e conseguimos fazer o push! Tudo certo! Vejamos o estado do repositório remoto:



Conclusão

Nesta apostila fizemos uma introdução bem superficial à sistemas de controle de versão, e em particular ao Git. Vários tópicos importantes não foram abordados, tais como:

- Fazer checkouts de versões diferentes
- “Reverter” commits
- *Branching*
- *Forking*, *pull requests* e arquiteturas de desenvolvimento colaborativo de software
- *Tagging*

Existem vários bons livros de Git no mercado para quem estiver interessado em aprender mais.

Além disso, eis uma tarefa para os mais interessados: faça um *fork* de algum projeto open-source de seu interesse, desenvolva uma nova *feature* e, se ficou legal, você pode pedir ao dono do código original que inclua seu desenvolvimento no código comunitário (ou seja, você pode fazer um *pull request*)! Parabéns, você acabou de contribuir concretamente em um software open-source, e pode adicionar este feito ao seu currículo!