

Angelo Medeiros Nóbrega

Git, uma abordagem pragmática

João Pessoa - PB

2017 - V0.2.0

Angelo Medeiros Nóbrega

Git, uma abordagem pragmática

Introdução ao controle de versão e boas práticas com o Git e o Git-flow, usando como repositório remoto o Gitlab.

João Pessoa - PB

2017 - V0.2.0

Lista de ilustrações

Figura 1 – Estratégia de ramificação	8
Figura 2 – O <i>branch develop</i>	9
Figura 3 – O <i>branch feature</i>	10
Figura 4 – Tipos de merges	11
Figura 5 – O <i>branch hotfix</i>	12
Figura 6 – Uma alternativa para o MS-DOS, o <i>Cmder</i>	13
Figura 7 – Verificando a instalação do git	14
Figura 8 – Configuração inicial do git	14
Figura 9 – Criando repositório git	15
Figura 10 – O primeiro estágio	16
Figura 11 – O segundo estágio	17
Figura 12 – O terceiro estágio	18
Figura 13 – Outra maneira de realizar um commit	18
Figura 14 – Visualizando o log básico	19
Figura 15 – Outra maneira de visualizar o log	20
Figura 16 – Mais uma maneira de visualizar o log	21

Lista de abreviaturas e siglas

CV Controle de versão

1	ANTES DE TUDO, O QUE É CONTROLE DE VERSÃO?	5
1.1	O Git	5
1.2	O GitLab	6
1.3	O Git-flow	6
2	A ESTRATÉGIA DE RAMIFICAÇÃO	7
2.1	Os branches	7
2.1.1	<i>O branch master</i>	7
2.1.2	<i>O branch develop</i>	7
2.1.3	<i>O branch feature</i>	9
2.1.4	<i>O branch release</i>	10
2.1.5	<i>Os branches hotfixes e bugfixes</i>	11
3	PRIMEIROS PASSOS COM O GIT	13
3.1	Configuração inicial	13
4	OS 3 ESTÁGIOS	15
4.1	Criando um repositório	15
4.2	O primeiro estágio	16
4.3	O segundo estágio	16
4.4	O terceiro estágio	17
5	COMANDOS MAIS USADOS NO GIT	19
5.1	Visualizando o log	19

Antes de tudo, o que é controle de versão?

Antes de começar a utilizar o *git* para versionar seus trabalhos (códigos, imagens, layouts...) você deve entender o que é controle de versão. Após entender esse conceito você estará apto a usar todo o poder que o *git* proporciona.

O controle de versão(CV) é um sistema usado para ter controle sobre todas(isso se for usada corretamente) as mudanças feitas em um determinado arquivo. O CV permite você reverter sua aplicação que se encontra em um estado que está apresentando um *bug*, para um estado anterior onde o *bug* não havia se manifestado. Permite você também descobrir quem introduziu um problema, quando foi introduzido e onde foi introduzido. Se estiver usando um repositório remoto, você não correrá o risco de perder seu arquivos e melhor ainda, você também não perderá o controle sobre as mudanças feitas localmente.

O controle de versão ajudará na organização, facilitará na hora de trabalhar em equipe, sem aquela história de dois desenvolvedores alterarem um mesmo arquivo, ao mesmo tempo, por estarem desenvolvendo um mesmo projeto. Segurança, vocês desenvolverão, todos os projetos, sem medo de perder código ou acabar errando alguma atualização, sem ter como voltar. E você verá que existem muitas outras razões para usar um sistema para controle de versão.

Muitas vezes o controle de versão é confundido com backup, lembre-se que no controle de versão você terá acesso ao arquivo atual e todas alterações ligadas ao arquivo, e no backup você terá acesso apenas a última versão do arquivo.

Lembre-se também que todas essas *features* que o controle de versão oferece só existirão se forem usadas corretamente.

1.1 O Git

O Git é a ferramenta que você utilizará para fazer todo o controle de versão. O Git surgiu quando Linus Torvalds, o criador do Linux, começou a enfrentar problemas quando desenvolvia o kernel do linux (projeto open source, ou seja, o Linus trabalhava com apoio de uma comunidade para seu desenvolvimento) com as ferramentas de versionamento da época havendo a necessidade da criação de uma nova ferramenta. A proposta para o Git era apresentar algumas *features* que o sistema antigo não oferecia:

1. Velocidade;
2. Projeto simples;
3. Forte suporte para desenvolvimento não-linear (milhares de ramos paralelos);
4. Completamente distribuído;
5. Capaz de lidar com projetos grandes como o núcleo o Linux com eficiência (velocidade e tamanho dos dados).

Desde 2005 quando o Git foi criado, ele passou por um longo período de evolução e ainda continua. Hoje ele está em uma versão estável oferecendo todas as *features* citadas acima.

1.2 O GitLab

O GitLab nada mais é que um gerenciador de repositório *git* remoto. O *Gitlab* apresenta algumas vantagens em relação ao *Github*:

1. Numero de Repositórios ilimitados;
2. Espaço ilimitado (futuramente será cobrado por projetos maiores que 5Gb), atualmente o *GitHub* limita em 1GB por projeto;
3. Integração continua integrada (*GitLab CI*);
4. Importação projetos do *GitHub*, *BitBucket* e *Gitorious*;
5. Armazenamento de repositórios em servidores privados.

A integração continua integrada funciona apenas para sistemas operacionais baseados no Linux.

1.3 O Git-flow

O *git-flow* é uma extensão do *git* para auxiliar o controle de versão usando comandos pré-definidos como boas práticas nesse quesito. Na minha opinião o *git-flow* é muito mais do que uma simples extensão, é uma filosofia, é uma nova maneira de pensar sobre o controle de versão.

Você pode aplicar a metodologia do *git-flow* sem a necessidade de ter ele instalado, usando apenas comandos nativos do *git*.

A estratégia de ramificação

A estratégia de ramificação assemelha-se muito a estrutura de uma árvore, por isso alguns comandos do *git* usam opções como *branch* (que significa ramo ou galho), ferramentas como o *SourceTree* que serve para visualizar toda a ramificação do projeto em forma de grafos (a título de informação, *tree* significa árvore).

A estratégia que está representada na figura 1, já é algo que grandes e pequenas empresas usam a bastante tempo. Explicarei como ela é construída, como iremos trabalhar em cima dessa estratégia usando o *git*, e como usar a poderosa extensão do *git*, o *git-flow*, para facilitar ainda mais nosso trabalho.

2.1 Os branches

Na estratégia que iremos adotar vamos trabalhar com seis tipos de ramos, são eles, dois ramos principais e quatro tipos de ramos de apoio.

Os ramos principais são os ramos *master* e o *develop*. E os ramos de apoio serão os ramos, *feature*, *release*, *hotfix* e *bugfix*. A seguir irei descrever cada um desses ramos.

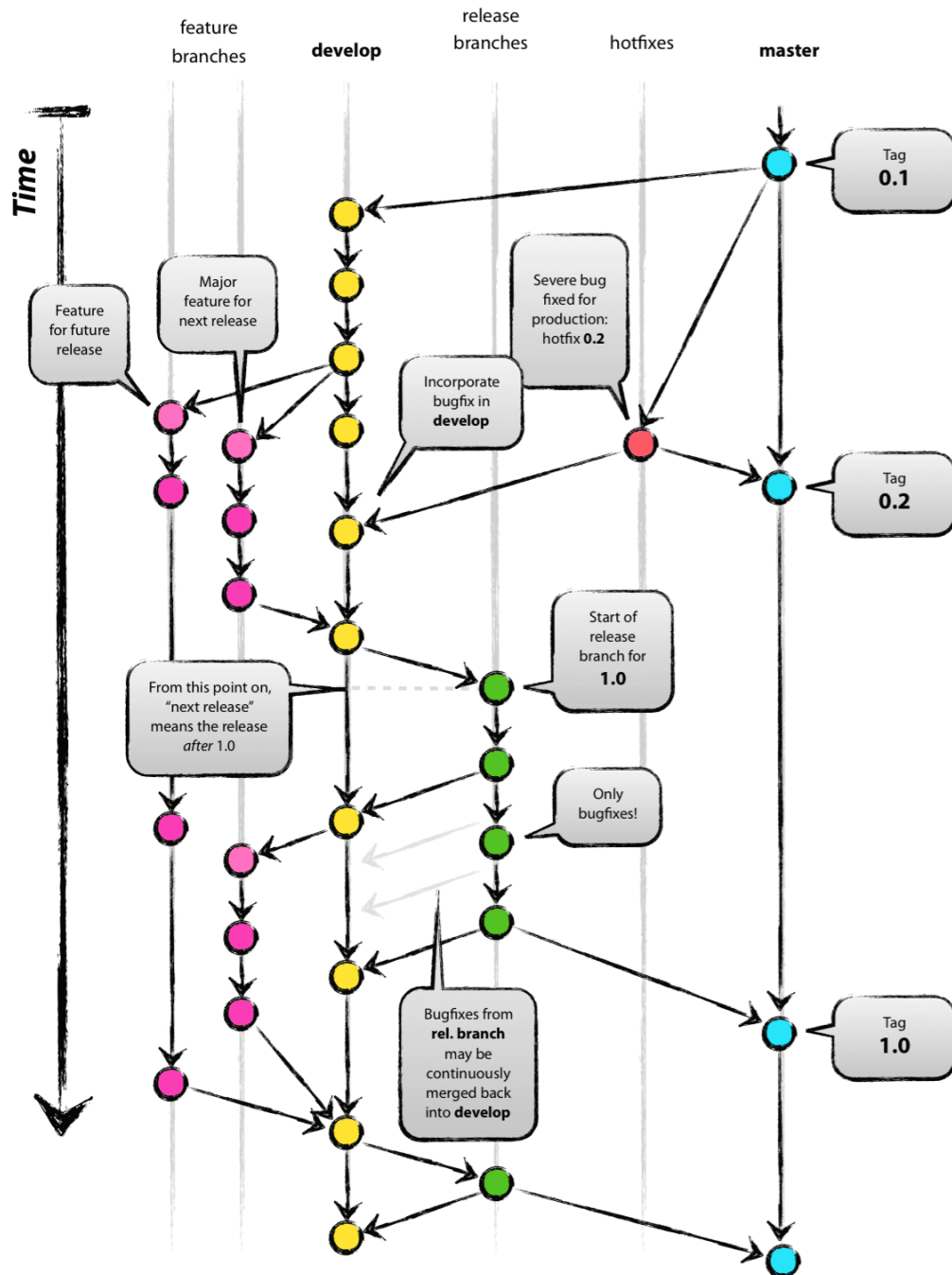
2.1.1 O *branch master*

O ramo *master* é o ramo que irá abrigar os códigos em suas versões mais estáveis, esse é o branch de produção. É o ramo que dará origem a aplicação final. Em algum momento do desenvolvimento todos os códigos produzidos em outros ramos farão um *merge* (ato de mesclar os ramos, colocar os arquivos de um *branch* em outro) com o *branch master*, de forma direta ou indireta.

2.1.2 O *branch develop*

Este ramo será responsável por conter os códigos em nível de desenvolvimento para o próximo *deploy* (significa implementar, mas pode mudar de significado de acordo com o contexto). Lembre-se que os códigos não serão criados nesse *branch*, esse é responsável apenas em abrigar os códigos que estão em desenvolvimento. Após os códigos serem devidamente testados, o *branch develop* fará um *merge* com o *branch master*, isso se nem

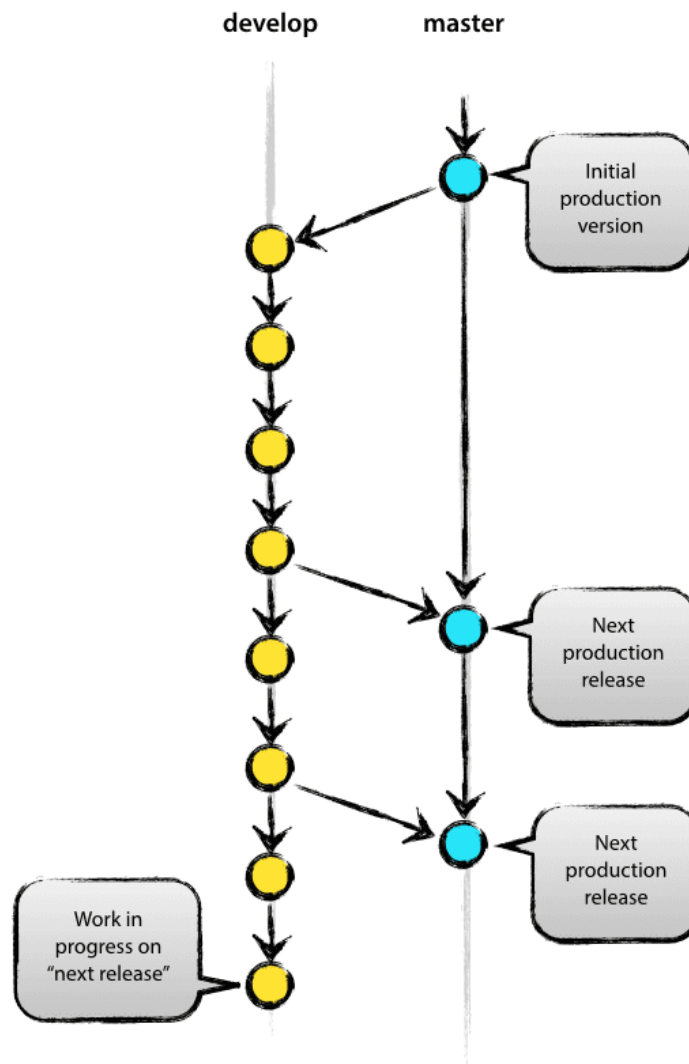
Figura 1 – Estratégia de ramificação



Fonte: (<http://nvie.com/posts/a-successful-git-branching-model/>)

um *bug* for encontrado no processo de testes, esse processo está representado na figura 2. Se o código apresentar *bug* será criado outro branch a partir do *branch develop* para a correção dos *bugs* e posteriormente mesclado com o *branch master*.

Os branches *master* e *develop*, possuem vida infinita, ou seja, eles sempre estarão presentes durante o desenvolvimento, e serão criados sempre que o git for inicializado. Os

Figura 2 – O *branch develop*

Fonte: (<http://nvie.com/posts/a-successful-git-branching-model/>)

branches a seguir terão vida curta, eles serão criados e após suas utilizações eles serão finalizados e excluídos, esse processo firará mais claro na prática.

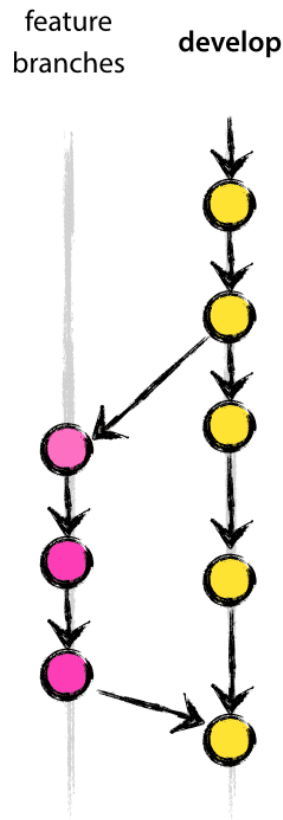
2.1.3 O *branch feature*

O branch feature representado na figura 3, será utilizado sempre que uma nova funcionalidade precisar ser criada. Ele sempre será criado a partir do *branch develop* e finalizado no *branch develop*, independente em qual ramo você esteja, isso se você estiver utilizando o *git-flow*.

Ao contrário do ramo *master* e *develop*, o ramo *feature* pode ser criado múltiplas vezes, uma para cada nova funcionalidade. Esse branch possui como prefixo *feature/**,

onde o asterisco(*) será substituído pelo nome do "sub-ramo" digamos assim, por exemplo, *feature/cadaastrodeclientes*, *feature/teladelogin*.

Figura 3 – O *branch feature*



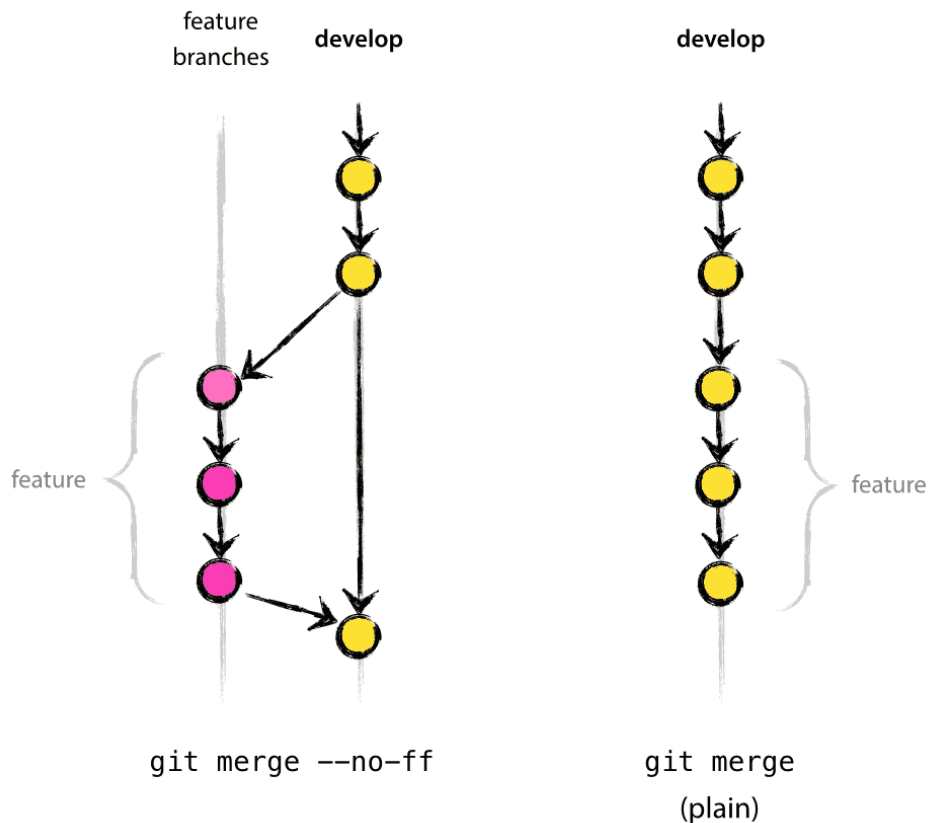
Fonte: (<http://nvie.com/posts/a-successful-git-branching-model/>)

São duas as principais maneiras de mesclar branches, figura 4. A primeira é fazendo a mesclagem criando um novo *commit* com as alterações dentro do ramo *develop*, sem adicionar os *commits* criados durante o desenvolvimento do ramo *feature* e, a outra é adicionando os *commits* criados no ramo *feature* dentro do ramo *develop* e, mais um novo *commit* indicando a mesclagem, o git-flow usa como padrão esse último. Novamente, esse processo ficará mais claro durante a prática.

2.1.4 O *branch release*

Esse branch é o ramo intermediário entre o *develop* e o *master*. O objetivo desse branch é a criação de tags (são os números que indicam a versão da aplicação, 1.0.0, por exemplo). Ele inicia no *branch develop* e termina no *branch master* e no *develop*. Você deve estar se perguntando porquê um *branch* que inicia-se no *develop* tem que terminar no *develop*. Isso acontece porquê o *branch develop* tem que ter a mesma *tag* do *branch master*.

Figura 4 – Tipos de merges



Fonte: (<http://nvie.com/posts/a-successful-git-branching-model/>)

O prefixo usado pelo *branch release* é *release/**, onde o asterisco(*) deve ser substituído pela tag da *release*, por exemplo, *release/0.1.0*. Mais na frente iremos aprender um conjunto de regras para a criação dessas tags, conhecido como versionamento semântico.

Sempre quando falo que um branch "termina" ou "finaliza", me refiro a mesclagem de um branch em outro, usado por padrão pelo git-flow.

2.1.5 Os *branches hotfixes* e *bugfixes*

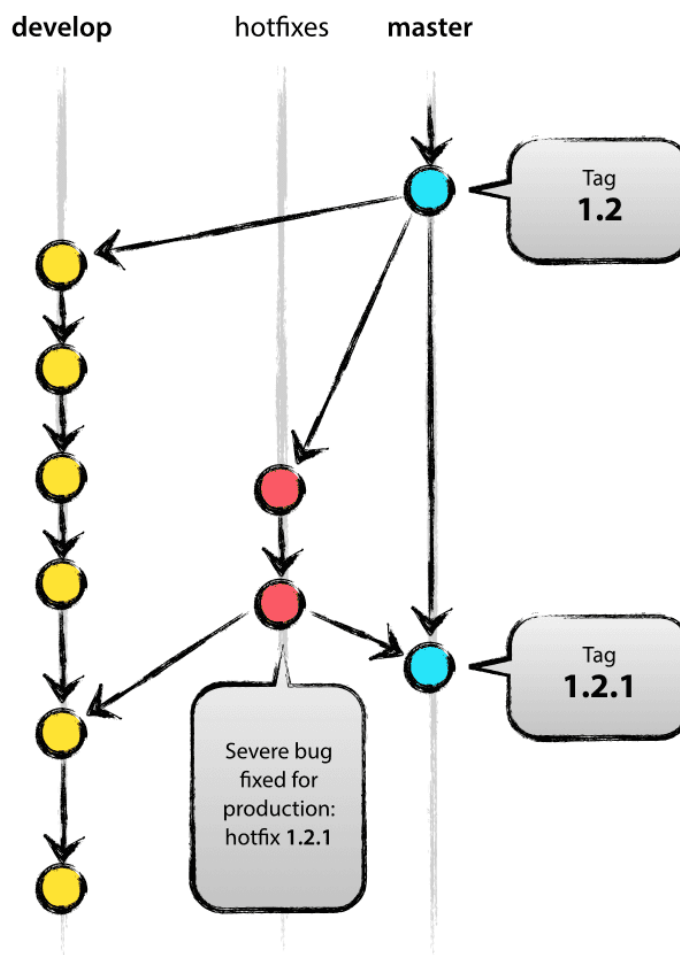
O *branch hotfix* e *bugfix*, tem o objetivo de correção de erros. A diferença entre eles é onde cada um é inicializado.

Se o *bug* for encontrado no ramo de produção(*branch master*), um *branch hotfix*, ver figura 5, deve ser criado para tratar o erro a partir do ramo *master*. Após a correção do erro, uma nova *tag* será criada automaticamente e, o *branch hotfix* será mesclado com o ramo *master* e também ao ramo *develop* para que esse não apresente o erro em futuras versões.

Análogo ao *branch release*, o prefixo do *branch hotfix* é *hotfix/**, onde o asterisco(*)

deve ser substituído pela nova tag, por exemplo, *hotfix/0.1.1*.

Figura 5 – O *branch hotfix*



Fonte: (<http://nvie.com/posts/a-successful-git-branching-model/>)

Já o *branch bugfix*, ele deve ser criado quando um *bug* é encontrado durante o desenvolvimento. Ele é iniciado e finalizado no *branch develop*. Seu prefixo é semelhante ao do *branch feature*, por exemplo, *bugfix/erro-cadastramento*.

instalado corretamente. Se ele tiver sido instalado corretamente, irá aparecer algo semelhante a figura 7.

Figura 7 – Verificando a instalação do git

```
u-angelo@AngeloUbuntu:~$ git
usage: git [--version] [--help] [-C <path>] [-c name=value]
       [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
       [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
       [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
       <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
```

Fonte: (Do autor)

Com o git devidamente instalado, vamos começar a configuração. Para isso insira os seguintes comandos, o primeiro será para o git identificar seu nome, e o segundo seu email(figura 8):

```
git config --global user.name "Angelo Medeiros"
git config --global user.email "angelo@email.com"
```

Caso queira visualizar suas configuração globais, digite o comando:

```
git config --global --list
```

O próximo comando serve para facilitar o entendimento visual:

```
git config --global color.ui true
```

Figura 8 – Configuração inicial do git

```
u-angelo@AngeloUbuntu:~$ git config --global user.name "Angelo Medeiros"
u-angelo@AngeloUbuntu:~$ git config --global user.email "angelo@email.com"
u-angelo@AngeloUbuntu:~$ git config --global --list
user.name=Angelo Medeiros
user.email=angelo@email.com
color.ui=true
```

Fonte: (Do autor)

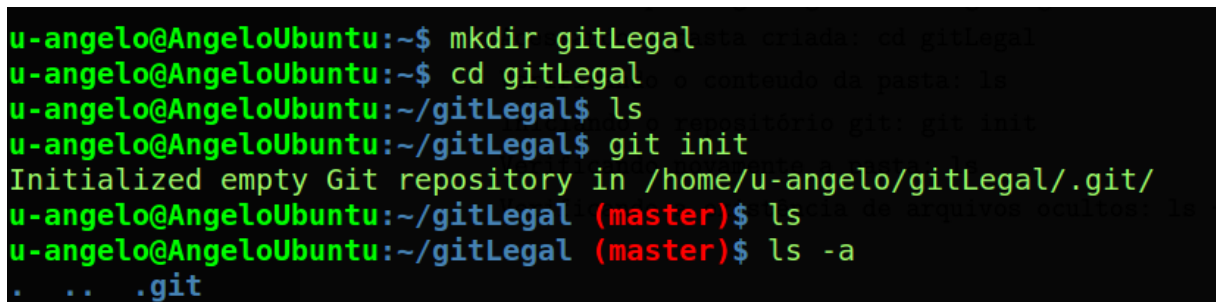
Nesse capítulo ensinarei a criar seu primeiro repositório git. E quais são os três principais estágios do processo para o versionamento usado pelo git.

4.1 Criando um repositório

O primeiro passo é acessar a pasta que você quer iniciar o repositório git. Para exemplificar eu criei uma pasta chamada gitLegal, e usei os seguintes comandos(ver figura 9):

```
Criando a pasta gitLegal: mkdir gitLegal
Acessando a pasta criada: cd gitLegal
Verificando o conteudo da pasta: ls
Iniciando o repositório git: git init
Verificando novamente a pasta: ls
Verificando a existência de arquivos ocultos: ls -a
```

Figura 9 – Criando repositório git



```
u-angelo@AngeloUbuntu:~$ mkdir gitLegal
u-angelo@AngeloUbuntu:~$ cd gitLegal
u-angelo@AngeloUbuntu:~/gitLegal$ ls
u-angelo@AngeloUbuntu:~/gitLegal$ git init
Initialized empty Git repository in /home/u-angelo/gitLegal/.git/
u-angelo@AngeloUbuntu:~/gitLegal (master)$ ls
u-angelo@AngeloUbuntu:~/gitLegal (master)$ ls -a
.  ..  .git
```

Fonte: (Do autor)

O processo de verificação da pasta foi feito apenas para mostrar que quando o repositório é inicializado o git cria uma pasta oculta. Nessa pasta oculta estão todos os arquivos necessário para o git gerenciar seu repositório. Na prática você usará apenas o comando *git init*.

4.2 O primeiro estágio

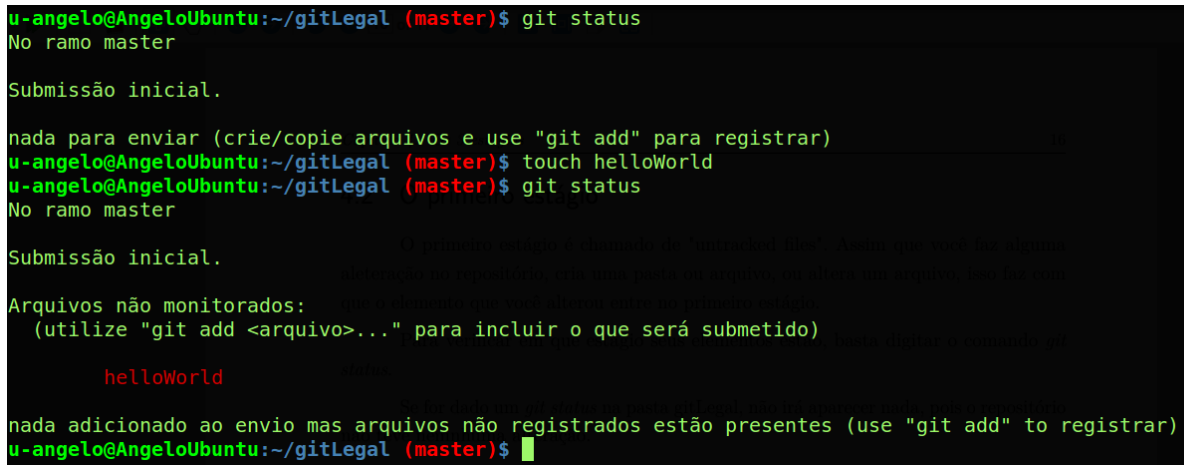
O primeiro estágio é chamado de "untracked files". Assim que você faz alguma alteração no repositório, cria uma pasta ou arquivo, ou altera um arquivo, isso faz com que o elemento que você alterou entre no primeiro estágio.

Para verificar em que estágio seus elementos estão, basta digitar o comando *git status*.

Se for dado um *git status* na pasta gitLegal, não irá aparecer nada, pois o repositório não teve nenhuma alteração.

Para demonstrar o primeiro estágio irei criar um arquivo em branco chamado *helloWorld* e, em seguida, executarei o comando para verificar o estágio(ver figura 10).

Figura 10 – O primeiro estágio



```
u-angelo@AngeloUbuntu:~/gitLegal (master)$ git status
No ramo master

Submissão inicial.

nada para enviar (crie/copie arquivos e use "git add" para registrar)
u-angelo@AngeloUbuntu:~/gitLegal (master)$ touch helloWorld
u-angelo@AngeloUbuntu:~/gitLegal (master)$ git status
No ramo master

Submissão inicial.

Arquivos não monitorados:
  (utilize "git add <arquivo>..." para incluir o que será submetido)

    helloWorld

nada adicionado ao envio mas arquivos não registrados estão presentes (use "git add" to registrar)
u-angelo@AngeloUbuntu:~/gitLegal (master)$
```

Fonte: (Do autor)

O comando *touch* foi usado para a criação do arquivo *helloWorld*.

4.3 O segundo estágio

O segundo estágio é chamado de "Changes to be committed". Nessa etapa você irá adicionar os elementos do primeiro estágio para serem "commitados" na próxima etapa. Irei criar outro arquivo em branco, chamado *helloWorld2*, apenas para ficar mais claro a importância dessa etapa(ver figura 11).

Agora vamos entender o que foi feito. Com a adição do arquivo *helloWorld2*, o primeiro estágio agora tem dois elementos. Vamos supor que você queira adicionar apenas o primeiro *helloWorld* para o último estágio. Para fazer isso, foi necessário apenas usar o comando *git add helloWorld*.

Figura 11 – O segundo estágio

```

u-angelo@AngeloUbuntu:~/gitLegal (master)$ touch helloWorld2
u-angelo@AngeloUbuntu:~/gitLegal (master)$ git status
No ramo master

Submissão inicial.

Arquivos não monitorados:
  (utilize "git add <arquivo>..." para incluir o que será submetido)

    helloWorld
    helloWorld2

```

O comando `touch` foi usado para a criação do arquivo `helloWorld2`. Nada adicionado ao envio mas arquivos não registrados estão presentes (use `git add` para adicioná-los).

```

u-angelo@AngeloUbuntu:~/gitLegal (master)$ git add helloWorld
u-angelo@AngeloUbuntu:~/gitLegal (master)$ git status
No ramo master

Submissão inicial.

Mudanças a serem submetidas:
  (utilize "git rm --cached <arquivo>..." para não apresentar)

    new file:   helloWorld

```

O segundo estágio é chamado de "Changes to be committed". Adicione os elementos do primeiro estágio para serem commitados.

```

Arquivos não monitorados:
  (utilize "git add <arquivo>..." para incluir o que será submetido)

    helloWorld2

```

```

u-angelo@AngeloUbuntu:~/gitLegal (master)$

```

Fonte: (Do autor)

Como mostra a figura 11, no segundo estágio encontra-se apenas o primeiro *helloWorld*. Peço desculpas pela falta de criatividade ao nomear os elementos.

4.4 O terceiro estágio

Nesse estágio é onde acontece o famoso *commit*. Para quem não sabe o que é um *commit*, ele nada mais é que um *snapshot* do estado de sua aplicação. Na figura 12 mostra o arquivo *helloWorld* sendo "commitado". Observe que após o *commit*, o arquivo *helloWorld* não aparece mais quando é usado o comando *git status*. O arquivo só voltará a aparecer quando sofrer alguma alteração novamente.

Existem duas principais maneiras de realizar um *commit*, na figura 12 mostra uma delas. Na figura 13 mostrarei a outra maneira realizando um *commit* do *helloWorld2*.

A primeira maneira deve ser usada quando você fez poucas alterações, e você

Figura 12 – O terceiro estágio

```

u-angelo@AngeloUbuntu:~/gitLegal (master)$ git commit -m "Commitando o arquivo helloWorld"
[master (root-commit) d3fbdd7] Commitando o arquivo helloWorld
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 helloWorld
u-angelo@AngeloUbuntu:~/gitLegal (master)$ git status
No ramo master
Arquivos não monitorados:
  (utilize "git add <arquivo>..." para incluir o que será submetido)

    helloWorld2

nada adicionado ao envio mas arquivos não registrados estão presentes (use "git add" to reg
u-angelo@AngeloUbuntu:~/gitLegal (master)$

```

Fonte: (Do autor)

Figura 13 – Outra maneira de realizar um commit

```

u-angelo@AngeloUbuntu:~/gitLegal (master)$ git add helloWorld2
u-angelo@AngeloUbuntu:~/gitLegal (master)$ git commit
1 - Adicionando o arquivo helloWorld2;
2 - Esse e um exemplo mostrando outra maneira de fazer um commit.
3
4 # Please enter the commit message for your changes. Lines starting
5 # with '#' will be ignored, and an empty message aborts the commit.
6 # No ramo master
7 # Mudanças a serem submetidas:
8 #   new file:   helloWorld2
9

```

Fonte: (Do autor)

consegue descrever todas essas mudanças em apenas uma linha usando o comando `git commit -m "comentario"`. A segunda maneira deve ser usada quando for preciso descrever diversas alterações e, apenas uma linha não basta. Uma dica é, evite usar acentuação dentro dos comentários, quando a acentuação é usada, algumas vezes quando você for visualizar os commits, a acentuação poderá não ser reconhecida pelo console.

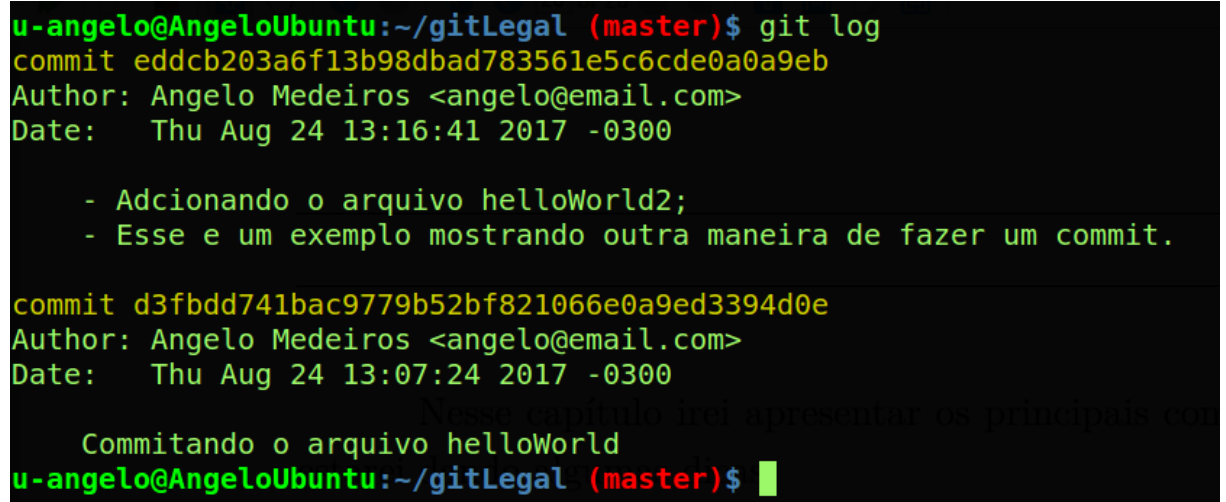
Comandos mais usados no git

Nesse capítulo irei apresentar os principais comandos usados pelo git, e sempre estarei dando algumas dicas.

5.1 Visualizando o log

Até agora vocês aprenderam a visualizar em que estágio está sua aplicação, e a criar os *commits*. Agora ensinarei como visualizar os *commits* criados, o *log*. O comando básico para visualizar um *log* é o *git log* (ver figura 14), mas ele também pode vir com outras opções, ensinarei os mais usados.

Figura 14 – Visualizando o log básico



```
u-angelo@AngeloUbuntu:~/gitLegal (master)$ git log
commit eddcb203a6f13b98dbad783561e5c6cde0a0a9eb
Author: Angelo Medeiros <angelo@email.com>
Date: Thu Aug 24 13:16:41 2017 -0300

    - Adicionando o arquivo helloWorld2;
    - Esse e um exemplo mostrando outra maneira de fazer um commit.

commit d3fbdd741bac9779b52bf821066e0a9ed3394d0e
Author: Angelo Medeiros <angelo@email.com>
Date: Thu Aug 24 13:07:24 2017 -0300

    Commitando o arquivo helloWorld
u-angelo@AngeloUbuntu:~/gitLegal (master)$
```

Fonte: (Do autor)

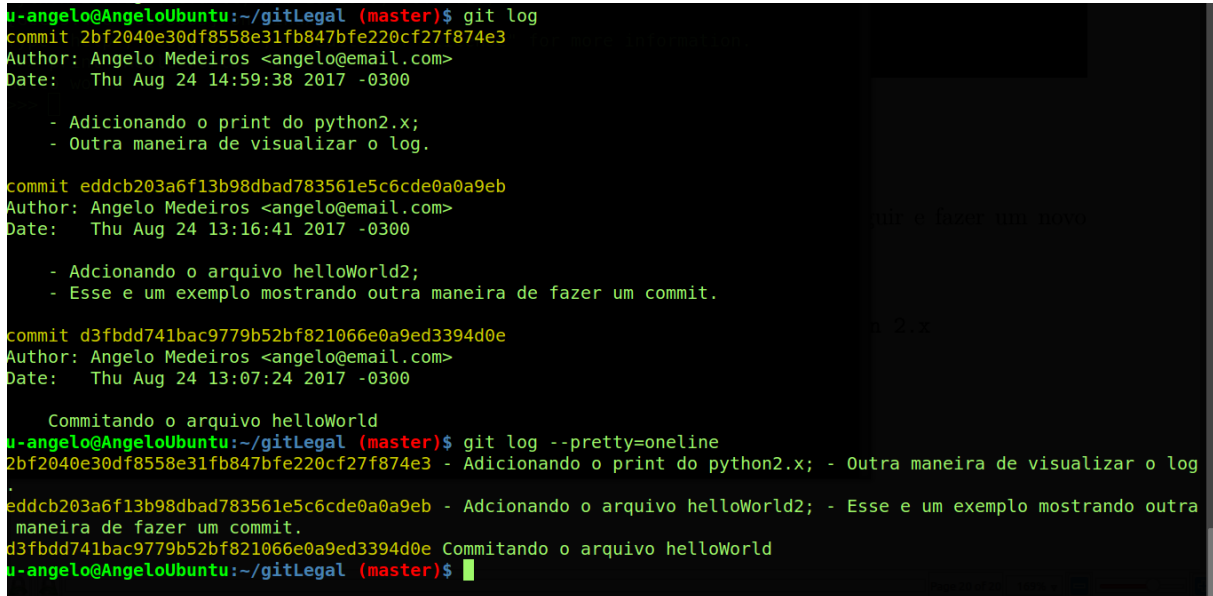
Irei adicionar dentro do arquivo helloWorld, o texto a seguir e fazer um novo commit.

```
# Comando para imprimir na tela usando o python 2.x

print "Hello world!"
```

A figura 15 mostra a execução do comando `git log --pretty=oneline`. Esse comando exibe o log de forma reduzida, mostrando apenas o *hash* e os comentários em apenas uma linha.

Figura 15 – Outra maneira de visualizar o log

A terminal window with a dark background and light green text. The prompt is 'u-angelo@AngeloUbuntu:~/gitLegal (master)\$'. The command 'git log' is entered, and the output shows three commits in oneline format. Each commit line includes the hash, author, date, and a list of changes. The first commit is '2bf2040e30df8558e31fb847bfe220cf27f874e3' with changes '- Adicionando o print do python2.x;' and '- Outra maneira de visualizar o log.'. The second is 'eddcdb203a6f13b98dbad783561e5c6cde0a0a9eb' with changes '- Adicionando o arquivo helloWorld2;' and '- Esse e um exemplo mostrando outra maneira de fazer um commit.'. The third is 'd3fbdd741bac9779b52bf821066e0a9ed3394d0e' with the change 'Commitando o arquivo helloWorld'. After the third commit, the command 'git log --pretty=oneline' is entered, and the same three commits are displayed in a more compact oneline format.

Fonte: (Do autor)

Outra opção para o mesmo comando é:

- `git log --pretty=oneline -2`, o parâmetro `-2`, faz com que o comando exiba apenas os dois últimos commits(você pode alterar o parâmetro por qualquer outro valor).

Abaixo seguem outros comandos para vocês experimentarem:

- `git log -p`, exibe as alterações em cada arquivo de todos os commits;
- `git log -p -1`, exibe as alterações do último commit(ver figura 16);
- `git log -stat`, exibe um resumo das alterações;
- `git log -stat -2`, exibe um resumo das alterações dos últimos dois commits;
- `git log --since=10.minutes`, exibe os commits dos últimos 10 minutos;
- `git log --since=2.hours`, exibe os commits das últimas duas horas;
- `git log --since=1.days`, exibe os commits no intervalo de tempo de um dia.

Figura 16 – Mais uma maneira de visualizar o log

```
u-angelo@AngeloUbuntu:~/gitLegal (master)$ git log -p -1
commit 2bf2040e30df8558e31fb847bfe220cf27f874e3
Author: Angelo Medeiros <angelo@email.com>
Date: Thu Aug 24 14:59:38 2017 -0300
- Adicionando o print do python2.x;
- Outra maneira de visualizar o log.

diff --git a/helloWorld b/helloWorld
index e69de29..3a5b3da 100644
--- a/helloWorld
+++ b/helloWorld
@@ -0,0 +1,3 @@
+# Comando para imprimir na tela usando o python 2.x
+
+print "Hello world!"
u-angelo@AngeloUbuntu:~/gitLegal (master)$
```

Fonte: (Do autor)

Na figura 16 as linhas que começam com um símbolo de mais(+) indica algo que foi adicionado. Se começar com o símbolo de menos(−) significa que algo foi retirado. Os números entre os símbolos de (@), indicam as posições das linhas, onde houve alterações. Quando vocês estiverem praticando, isso ficará mais claro.

5.2 Criando branches