

Angelo Medeiros Nóbrega

Git, uma abordagem pragmática

João Pessoa - PB

2017 - V1.0.0

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 - V1.0.0

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
Figura 17 – Criando um branch	21
Figura 18 – Mesclando usando o merge	22
Figura 19 – Mesclando usando o rebase	23
Figura 20 – Voltando versão	24
Figura 21 – Ignorando arquivos	25
Figura 22 – Criando repositório - 1ª etapa	27
Figura 23 – Criando repositório - 2ª etapa	28
Figura 24 – Criando repositório - 3ª etapa	29
Figura 25 – Criando repositório - 4ª etapa	30
Figura 26 – Resolvendo conflito - parte 1	34
Figura 27 – Resolvendo conflito - parte 2	34
Figura 28 – Resolvendo conflito - parte 3	35
Figura 29 – Resolvendo conflito - parte 4	35
Figura 30 – Iniciando repositório com o <i>Git-flow</i>	37
Figura 31 – Resumo dos comandos do <i>Git-flow</i>	39

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	O <i>branch master</i>	7
2.1.2	O <i>branch develop</i>	7
2.1.3	O <i>branch feature</i>	9
2.1.4	O <i>branch release</i>	10
2.1.5	Os <i>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
5.2	Criando branches	21
5.3	Mesclando branches	22
5.3.1	Mesclando usando o merge	22
5.3.2	Mesclando usando o rebase	22
5.4	Voltando versões	23
5.5	Algumas dicas	24
5.5.1	Ignorando arquivos com o Git	24
5.5.2	Alterando o proxy	25
6	TRABALHANDO COM REPOSITÓRIO REMOTO	27
6.1	Criando seu primeiro repositório remoto	27
6.2	Realizando seu primeiro <i>push</i>	30

6.3	Realizando seu primeiro clone	31
6.4	Criando um branch a partir do repositório remoto	31
6.5	Realizando seu primeiro <i>pull</i>	31
6.6	Trabalhando com <i>tags</i>	32
6.6.1	Versionamento Semântico	32
6.6.2	Criando <i>tags</i> com o <i>git</i>	33
6.7	Resolvendo conflitos	33
7	TRABALHANDO COM O GIT-FLOW	36
7.1	Criando o repositório usando o <i>git-flow</i>	36
7.2	Criando features	36
7.3	Criando releases	38
7.4	Criando hotfixes	38
7.5	Criando bugfixes	38
7.6	Resumo dos comandos do <i>git-flow</i>	39
	Finalizando	40

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 o mesmo arquivo ao mesmo tempo por estarem desenvolvendo um mesmo projeto. Segurança, vocês desenvolverão todos os projetos sem medo de perder algum código ou acabar errando alguma atualização sem ter como voltar. 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 termos 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, é uma estratégia que grandes e pequenas empresas usam a bastante tempo. Explicarei como ela é construída e 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*. Os ramos de apoio serão os branches *feature*, *release*, *hotfix* e *bugfix*. A seguir irei descrever cada um desses ramos.

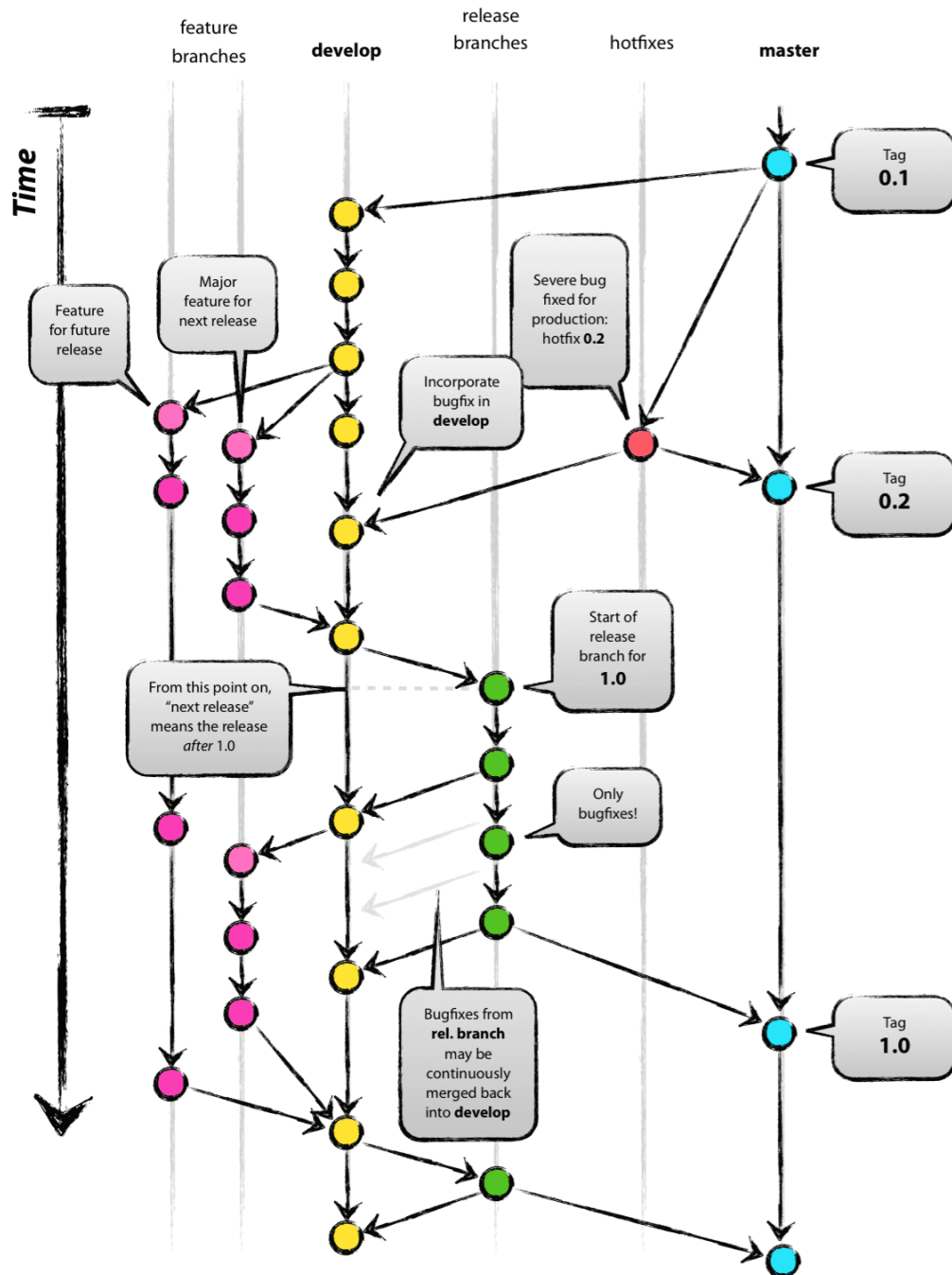
2.1.1 O *branch master*

O branch *master* é o ramo que irá abrigar os códigos em suas versões mais estáveis, ele será 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*, ele 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

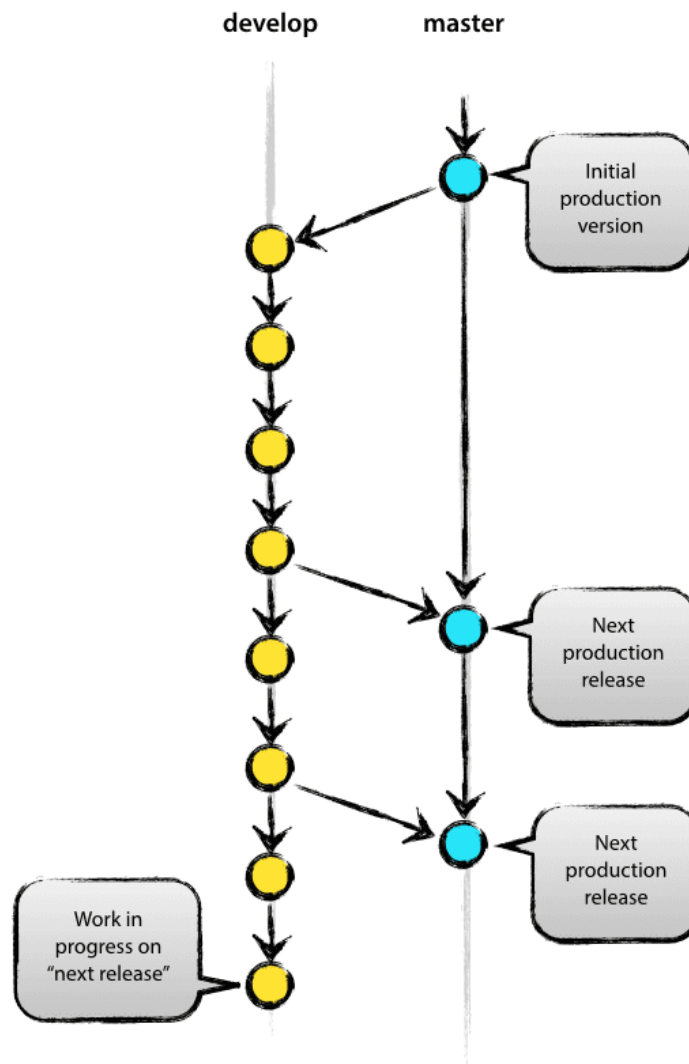
Figura 1 – Estratégia de ramificação



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 sempre serão criados quando o repositório for

Figura 2 – O *branch develop*

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

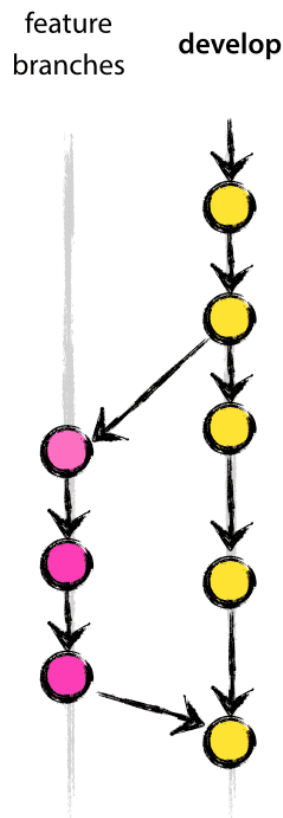
inicializado (isso se você estiver usando o *git-flow*, por padrão o *git* cria apenas o *master*). Os branches a seguir terão vida curta, eles serão criados durante a evolução do projeto e, após suas utilizações, eles serão finalizados e excluídos, esse processo ficará mais claro na prática.

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



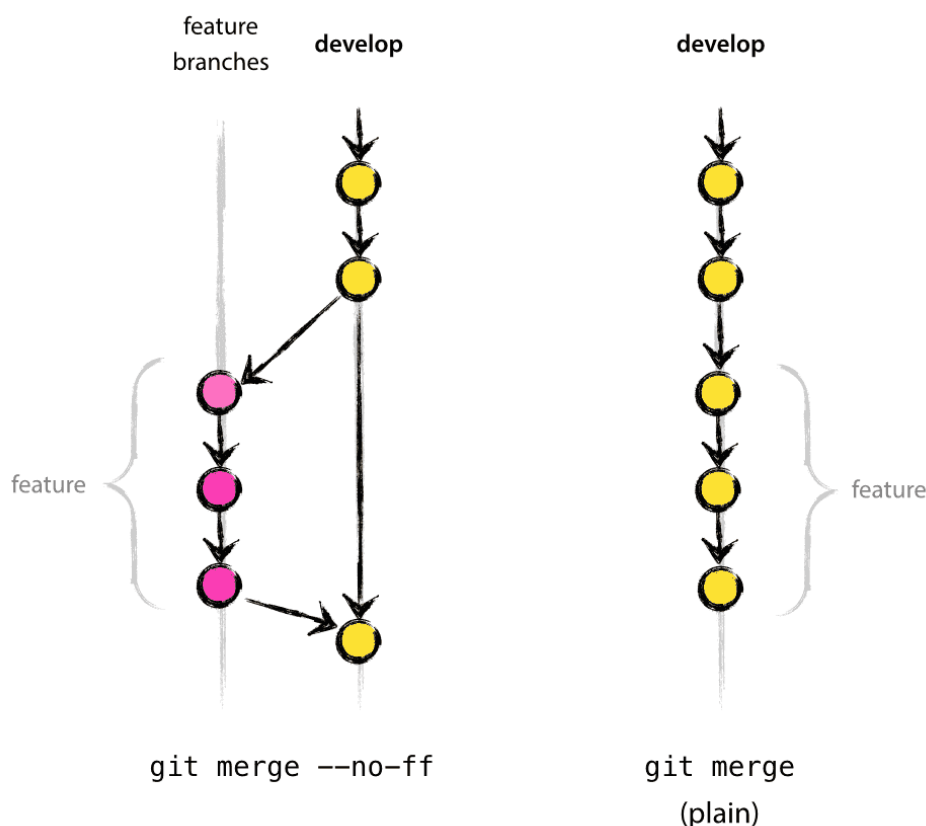
Fonte: (<http://nvie.com/posts/a-successful-git-branching-model/>)

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

Figura 4 – Tipos de merges



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

deve está 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*.

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*

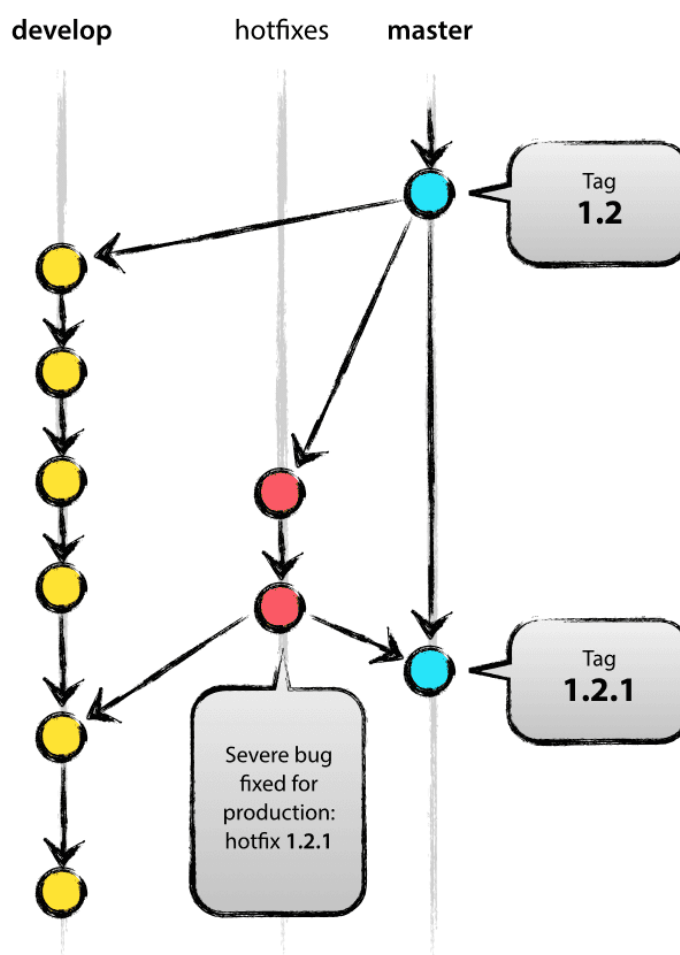
O *branch hotfix* e *bugfix* tem o objetivo de correção de erros. A diferença entre eles é onde cada um é inicializado.

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/>)

O *branch bugfix* 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*.

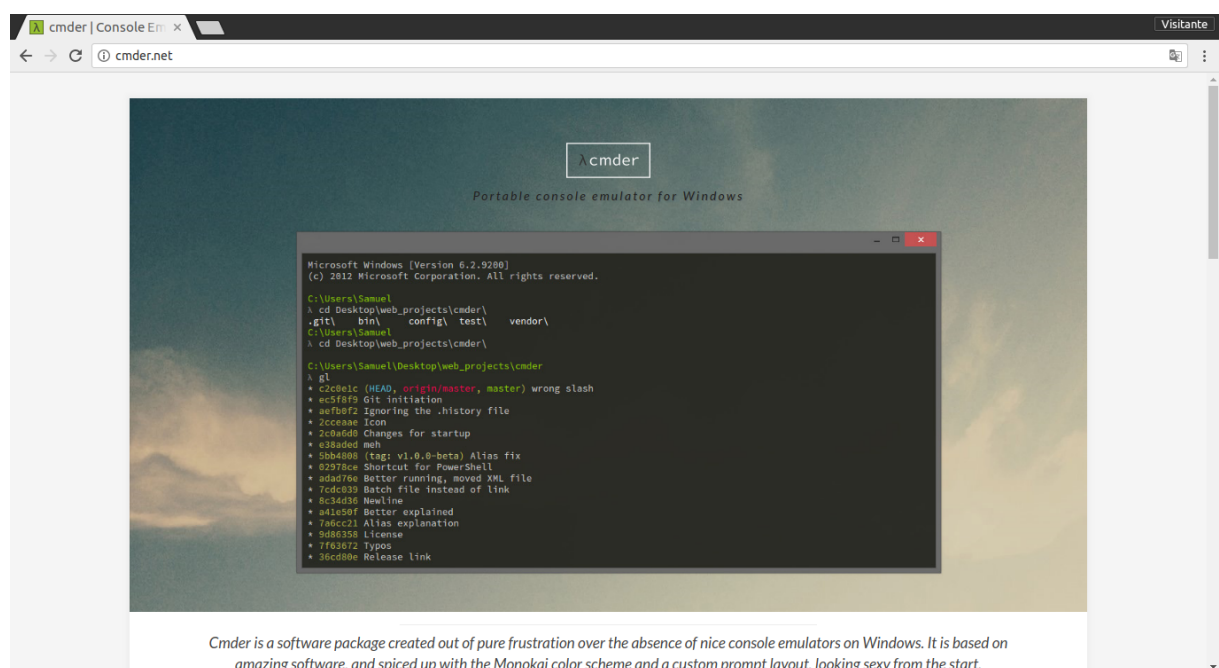
Primeiros passos com o Git

Nessa etapa vamos supor que todos estejam com o Git e o Git-flow instalados, caso haja a necessidade de um guia para as instalações, esse será adicionado como apêndice no final livro em futuras versões.

3.1 Configuração inicial

O primeiro passo é abrir seu console preferido, se estiverem no Linux ou macOS, abram o terminal por exemplo. Se estiverem utilizando o Windows vocês podem usar o Git Bash que é instalado com o Git. Ainda para usuários Windows, uma dica que dou é usar o Cmder, é um console free e muito mais agradável de se trabalhar e não precisa ser instalado, apenas baixado, ver figura 6. Sugiro também não utilizar o MS-DOS, nem o PowerShell.

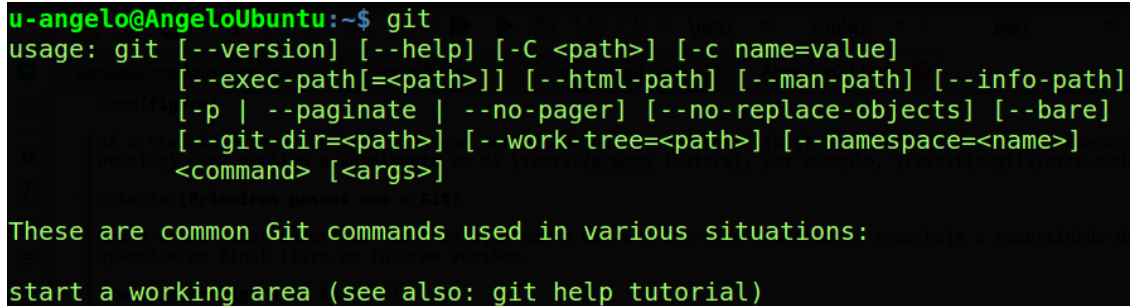
Figura 6 – Uma alternativa para o MS-DOS, o *Cmder*



Fonte: (<http://cmder.net/>)

Com o seu console aberto digite *git* e aperte enter, isso irá verificar se o git foi 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"
```

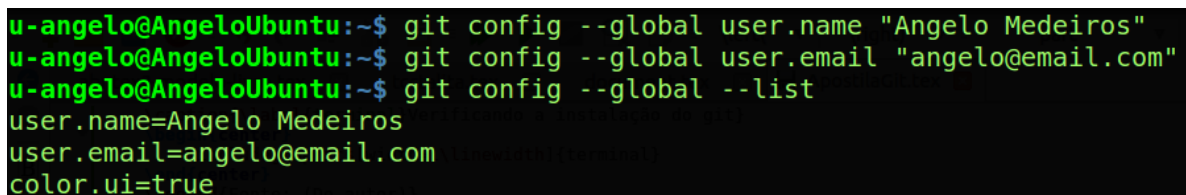
Caso queira visualizar suas configuração globais digite o comando:

```
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)

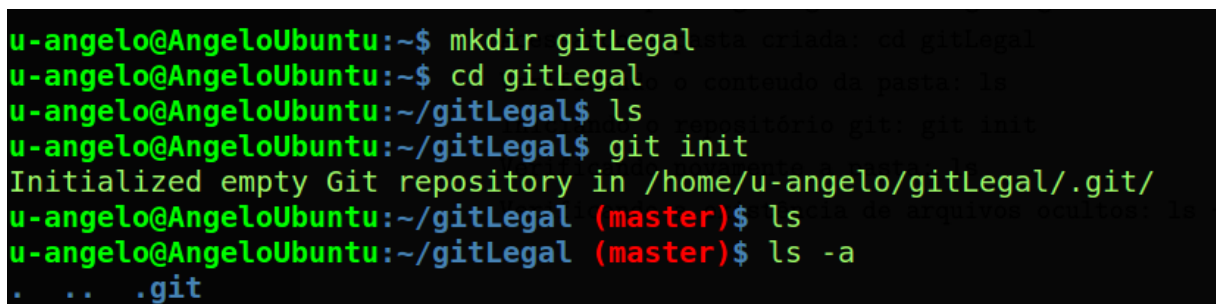
Nesse capítulo ensinarei a criar seu primeiro repositório git e, quais são os três principais estágios do processo para o versionamento usado pelo git.

4.1 Criando um repositório

O primeiro passo é acessar a pasta pelo terminal 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

A terminal window screenshot with a black background and green text. It shows the following commands and output: 1. 'mkdir gitLegal' is executed. 2. 'cd gitLegal' is executed. 3. 'ls' is executed, showing an empty directory. 4. 'git init' is executed, resulting in the message 'Initialized empty Git repository in /home/u-angelo/gitLegal/.git/'. 5. 'ls' is executed again, showing the new '.git' directory. 6. 'ls -a' is executed, also showing the '.git' directory. The prompt changes from '\$' to '(master)\$' after the 'git init' command.

```
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

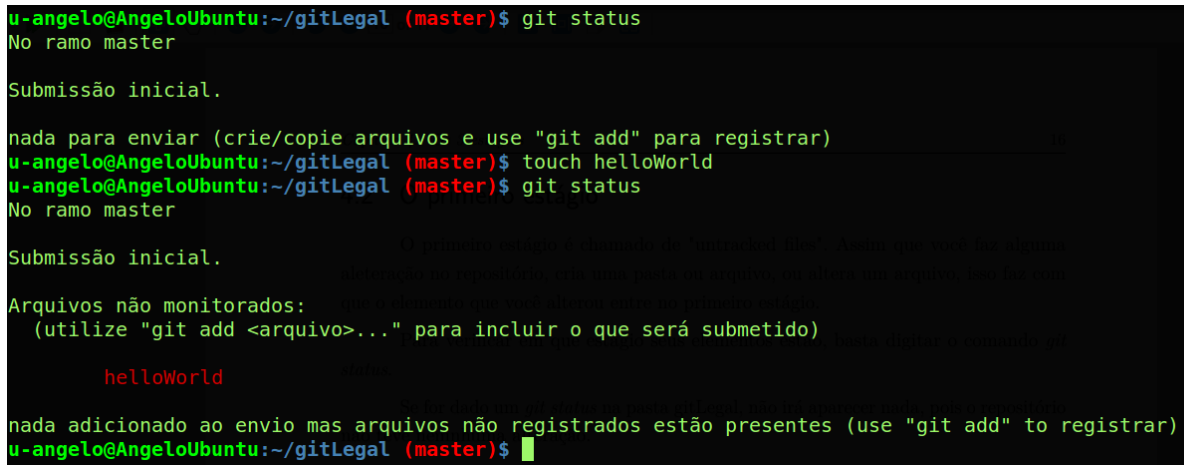
O primeiro estágio é chamado de "untracked files". Assim que você faz alguma alteração no repositório, criando uma pasta ou arquivo, ou alterando um arquivo existente, isso faz com que o elemento que você alterou entre no primeiro estágio.

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" no terceiro estágio. 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.

Na maioria das vezes é necessário adicionar mais de um arquivo ao terceiro estágio e não é viável adicionar um por um. Uma alternativa mais eficiente para essa situação é usar o comando “**git add .**”, ele irá adicionar todos os arquivos ao terceiro estágio. Preste atenção no ponto seguido do `add`, ele também faz parte do comando.

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.

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)

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*.

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)

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

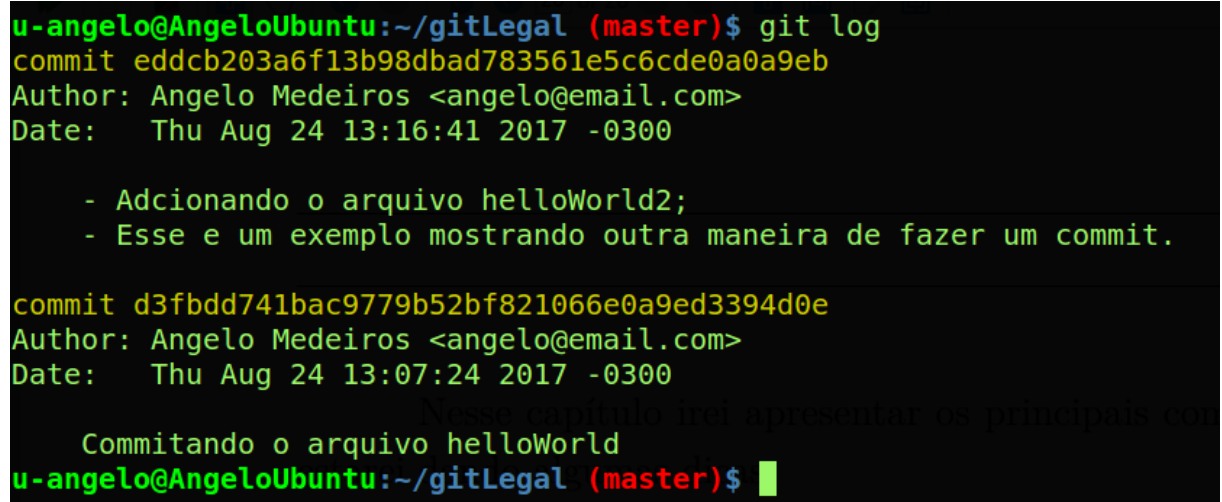
Comandos mais usados no git

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)

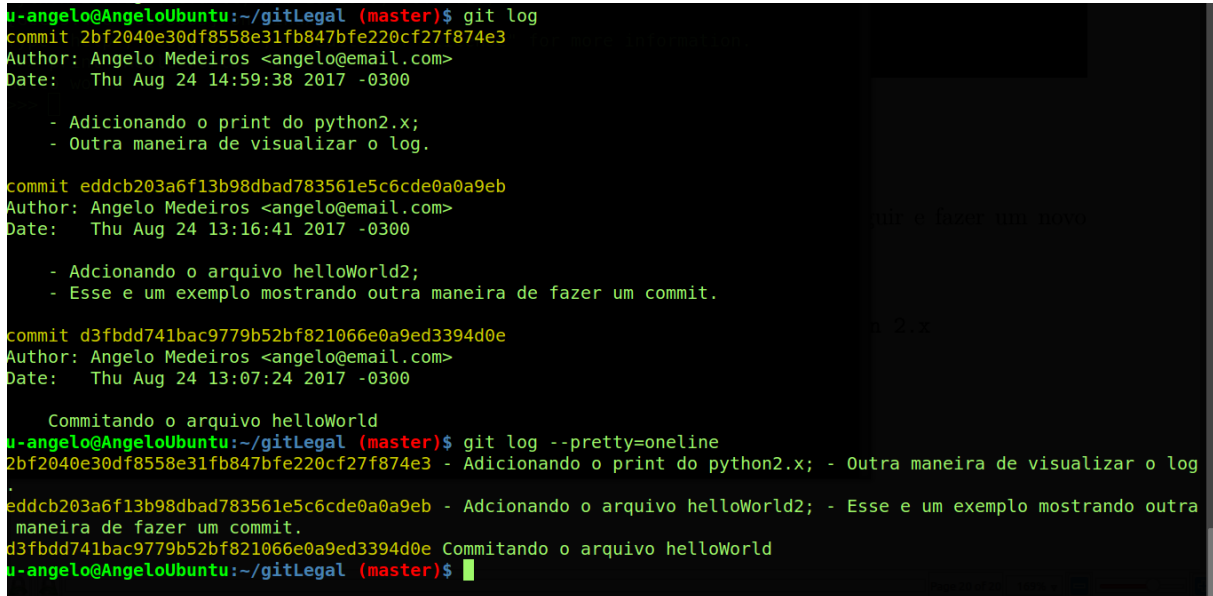
Irei adicionar dentro do arquivo helloWorld, o texto a seguir e fazer um novo commit.

```
# 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



```
u-angelo@AngeloUbuntu:~/gitLegal (master)$ git log
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.

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)$ git log --pretty=oneline
2bf2040e30df8558e31fb847bfe220cf27f874e3 - Adicionando o print do python2.x; - Outra maneira de visualizar o log
eddcb203a6f13b98dbad783561e5c6cde0a0a9eb - Adicionando o arquivo helloWorld2; - Esse e um exemplo mostrando outra
d3fbdd741bac9779b52bf821066e0a9ed3394d0e Commitando o arquivo helloWorld
u-angelo@AngeloUbuntu:~/gitLegal (master)$
```

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 houveram alterações. Quando vocês estiverem praticando, isso ficará mais claro.

5.2 Criando branches

Para criar um novo *branch*, usamos o comando `git checkout -b nomeBranch`. Onde *nomeBranch* deve ser substituído pelo nome do *branch* que você quer criar. Vale resaltar que um *branch* é sempre criado baseado no *branch* atual, ou seja, se você estiver no *branch master*, o novo *branch* será um clone do *branch master*, ver figura 17.

Figura 17 – Criando um branch

```

u-angelo@AngeloUbuntu:~/gitLegal (master)$ ls
helloWorld2 helloWorld test
u-angelo@AngeloUbuntu:~/gitLegal (master)$ git checkout -b meuPrimeiroBranch
Switched to a new branch 'meuPrimeiroBranch'
u-angelo@AngeloUbuntu:~/gitLegal (meuPrimeiroBranch)$ ls
helloWorld2 helloWorld test
u-angelo@AngeloUbuntu:~/gitLegal (meuPrimeiroBranch)$

```

Fonte: (Do autor)

Observe que quando você cria um *branch*, o *git* já inicia dentro do novo *branch*. Irei listar abaixo os principais comandos relacionados a manipulação dos branches.

- `git checkout nomeDoBranch`, troca do *branch* atual para o *branch* *nomeDoBranch*;
- `git branch`, exibe os branches criados localmente;
- `git branch -a`, exibe os branches locais e os remotos(se você tiver trabalhado com branches remotos);
- `git branch -d nomeDoBranch`, deleta o *branch* *nomeDoBranch*.

5.3 Mesclando branches

Mesclar branches na maioria das vezes é uma tarefa fácil. Porém algumas vezes ocorrem conflitos, esse assunto será tratado em uma seção mais na frente. Por enquanto, vamos nos preocupar apenas em mesclar os *branches*. São duas as principais maneiras de fazer a mesclagem usando o git, você pode usar o *merge* ou o *rebase*.

5.3.1 Mesclando usando o merge

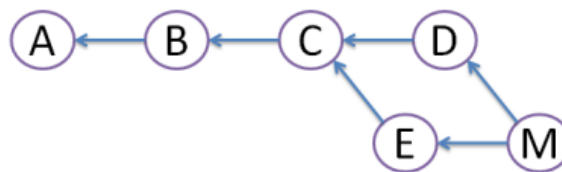
O comando

```
git merge funcionalidade1
```

é usado para realizar o merge. Ele irá mesclar o *branch* *funcionalidade1* com o *branch* *atual*.

A vantagem do merge é que ele preserva um histórico completo do seu projeto e, evita problemas quando está trabalhando em equipe(ver figura 18).

Figura 18 – Mesclando usando o merge



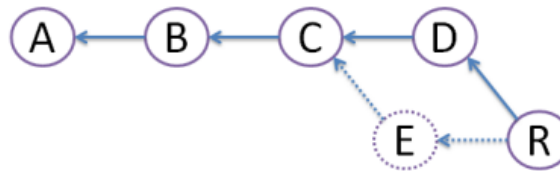
Fonte: (<https://stackoverflow.com/questions/16666089>)

5.3.2 Mesclando usando o rebase

O rebase preserva um histórico mais limpo e linear do seu projeto, porém ao trabalhar em grupo isso torna-se uma desvantagem, pois o rebase recria os commits. Na figura 19 quando o elemento E foi mesclado com o elemento D dando origem ao elemento R, o elemento E foi reescrito com outro commit, isto é, com outro hash. Então para alguém

de fora que tenha o elemento E vai ter um commit diferente do seu, apesar de ser o mesmo elemento, tornando algo confuso.

Figura 19 – Mesclando usando o rebase



Fonte: (<https://stackoverflow.com/questions/16666089>)

O rebase é aconselhado apenas quando você está trabalhando localmente, a não ser que você saiba o que está fazendo. Caso contrário, use sempre o merge. A documentação do git descreve diversos parâmetros que podem e devem ser usados, tanto para o rebase e para o merge.

5.4 Voltando versões

Na minha opinião essa é a principal função do git, o processo de voltar versões. Existem diversas maneiras de fazer isso. Algumas são simples, mas não oferecem muita segurança para quem está iniciando nesse novo mundo do controle de versões. Outras envolvem mais etapas, porém você sentirá segurança realizando o processo, o fato de ter mais etapas não o torna mais difícil.

A maioria dos comandos para voltar versões necessitam do *hash* do *commit*, você pode encontrar esse *hash* visualizando o *log*. Não é necessário o hash completo, apenas o começo. A seguir estão os principais comandos:

- `git reset d3fbdd741 --hard`, desfaz todas as alterações anteriores do commit d3fbdd741(ver figura 20);
- `git reset HEAD~2 --hard`, desfaz todas as alterações dos dois últimos commits;
- `git ckeckout d3fbdd741`, cria um branch temporário a partir do commit d3fbdd741.

Entendo a figura 20. Veja que antes do *reset* existia o arquivo *helloWorld2*. No final do processo além de apagar o arquivo *helloWorld2*, as alterações feitas no arquivo *helloWorld* também foram desfeitas, apesar de não mostrar na figura. Essas mudanças aconteceram porque o arquivo *heloWorld2* foi criado no *commit* eddcb203a, como pode ser observado nos comentários do *commit* mostrado no *log*, e o conteúdo do arquivo *helloWorld*

Figura 20 – Voltando versão

```
u-angelo@AngeloUbuntu:~/gitLegal (master)$ ls
helloWorld helloWorld2 test
u-angelo@AngeloUbuntu:~/gitLegal (master)$ git log
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.

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)$ git reset d3fbdd741 --hard
HEAD is now at d3fbdd7 Commitando o arquivo helloWorld
u-angelo@AngeloUbuntu:~/gitLegal (master)$ ls
helloWorld test
u-angelo@AngeloUbuntu:~/gitLegal (master)$
```

Fonte: (Do autor)

ocorreu no *commit* 2bf2040e30. Por isso a importância de documentar corretamente os *commits*.

Você pode combinar diversas técnicas para voltar versões usando o git. Quando forem ganhando mais segurança do que estão fazendo, o processo de voltar versões irá se tornar algo natural. Por exemplo, você pode criar um novo *branch* a partir do qual queira voltar uma versão, e depois mesclar esse novo *branch* com o *branch* principal. Depende apenas da sua engenhosidade, então façam diversos experimentos usando o git para adquirir experiência.

5.5 Algumas dicas

Nas seções seguintes serão descritas algumas dicas.

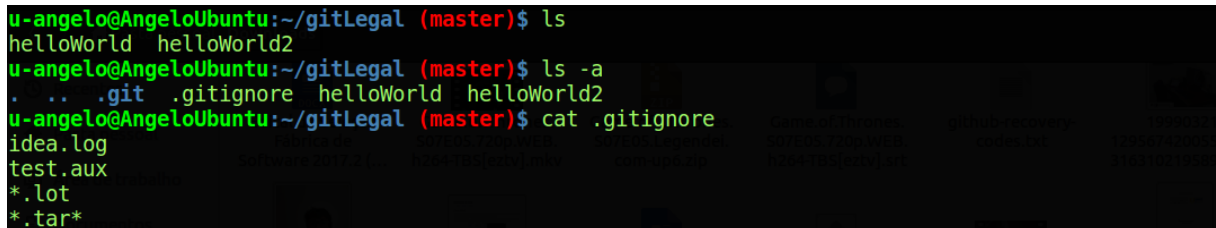
5.5.1 Ignorando arquivos com o Git

Geralmente tem arquivos que são criados automaticamente durante um projeto, por exemplo arquivos de log. Tais arquivos sempre ficam aparecendo no segundo estágio e

isso não é algo interessante.

Para impedir que alguns arquivos fiquem aparecendo ao visualizar o status, o git trabalha com o arquivo **.gitignore**. Esse arquivo é criado na raiz do projeto e, contém os nomes dos arquivos a serem ignorados, ver figura 21.

Figura 21 – Ignorando arquivos



```
u-angelo@AngeloUbuntu:~/gitLegal (master)$ ls
helloWorld helloWorld2
u-angelo@AngeloUbuntu:~/gitLegal (master)$ ls -a
. .. .git .gitignore helloWorld helloWorld2
u-angelo@AngeloUbuntu:~/gitLegal (master)$ cat .gitignore
idea.log
test.aux
*.lot
*.tar*
```

Fonte: (Do autor)

No exemplo da figura 21, os arquivos *idea.log* e *test.aux*, não aparecerão mais no status. Todos os arquivos que terminarem com a extensão **.lot** ou apresentarem a extensão **.tar**, por exemplo *ubuntuMobile.tar.xz*, também serão ignorados.

Observação: Se um arquivo já tiver sido adicionado para o segundo estágio em algum momento do desenvolvimento, ele não será ignorado, mesmo que o nome do arquivo esteja presente no **.gitignore**. Se esse for o caso, o comando

```
git rm --cached nomeDoArquivo
```

irá retirar o arquivo do segundo estágio, ou seja, do monitoramento e, a partir disso ele passará a ser ignorado normalmente, caso queira retirar uma pasta completa, use o comando

```
git rm -r --cached nomeDaPasta
```

experimente ambos como prática.

5.5.2 Alterando o proxy

Dependendo do local onde esteja trabalhando, é necessário uma configuração prévia do proxy para o git trabalhar com repositórios remotos.

Os principais comandos para alteração do proxy são:

- `git config --global http.proxy "proxy:port"`
- `git config --global http.proxy "10.10.32.1:3128"`, altera o proxy para 10.10.32.1 na porta 3128

- `git config --global http.proxy ""`, altera o proxy de *manual* para *nenhum*
- `git config --global --unset http.proxy`, outra maneira para desativar o proxy

Trabalhando com repositório remoto

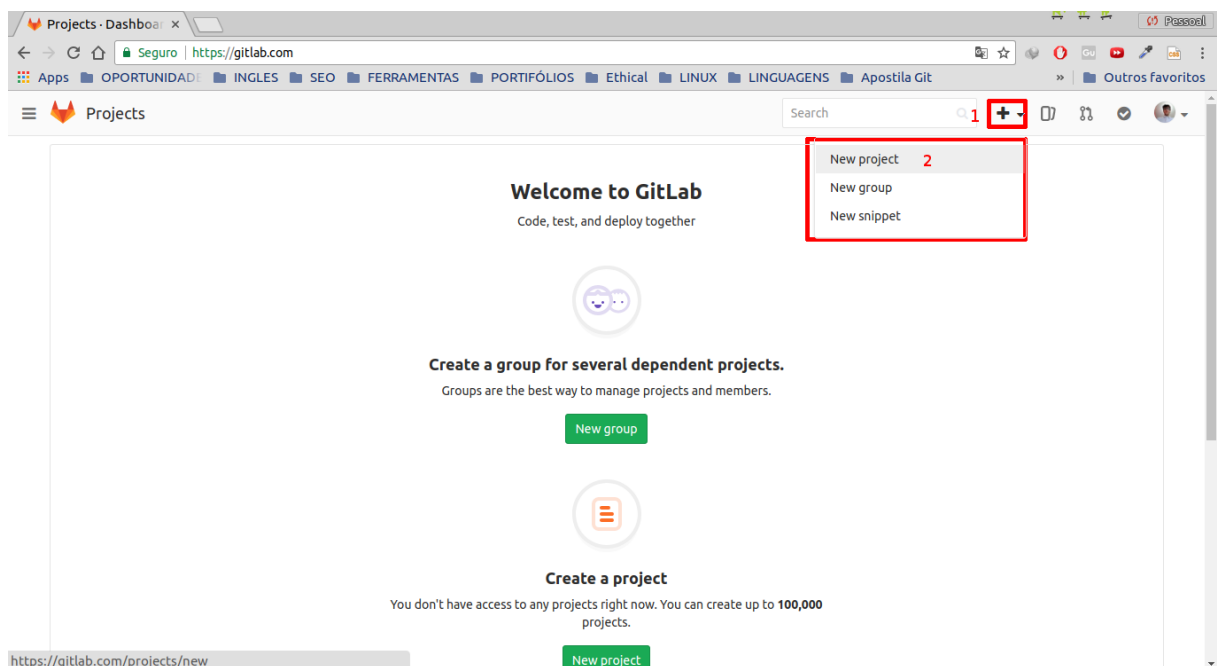
Na seção 1.2 falei brevemente sobre o *gitLab*, Nesse capítulo iremos estudar como o *git* se comporta trabalhando com repositórios remotos e falaremos sobre algumas características do *gitLab*.

Antes de começar a usar os comandos das próximas seções, será necessário a criação de uma conta no *gitLab*.

6.1 Criando seu primeiro repositório remoto

Com o *gitLab* aberto siga as instruções da figura 22 para iniciar o processo de criação do seu primeiro repositório. Em **1** você irá abrir as opções para a criação do novo repositório. Em **2** você irá selecionar *New project*, a continuação segue nas figuras subsequentes.

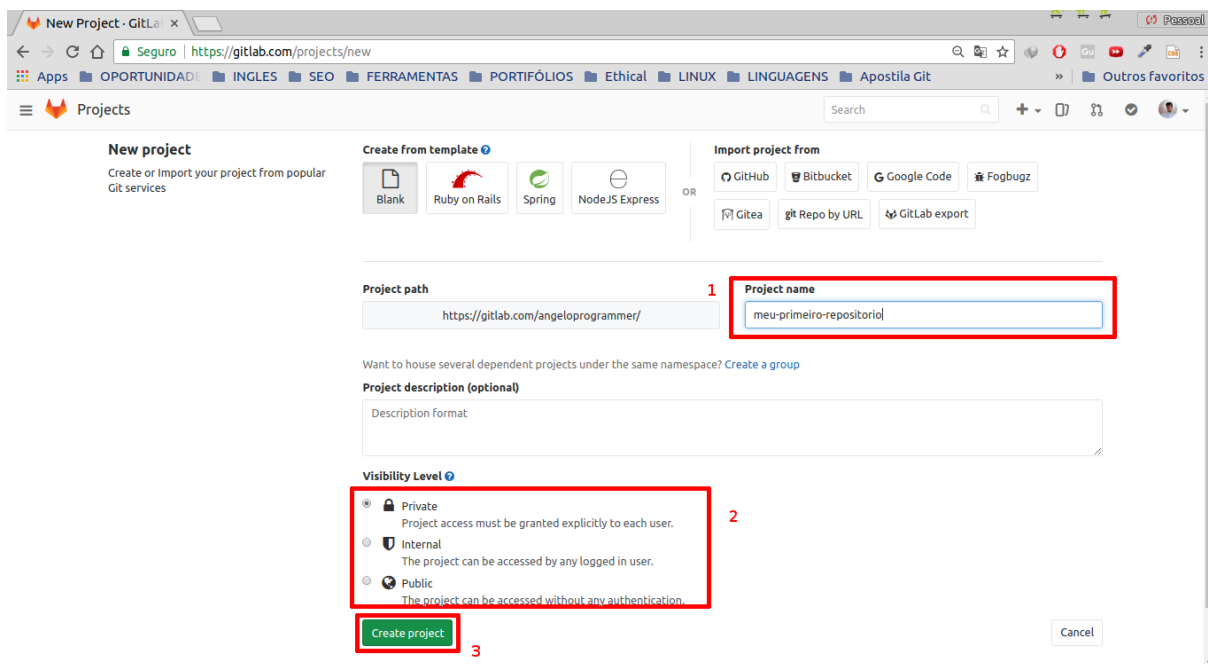
Figura 22 – Criando repositório - 1ª etapa



Fonte: (Do autor)

Continuando na figura 23, em **1** você irá adicionar o nome do repositório, não necessariamente sendo igual ao repositório local. Em **2** exibe as opções de visibilidade do projeto, não tendo relação alguma com as permissões do projeto. Em **3** aperte em *Create project*.

Figura 23 – Criando repositório - 2ª etapa



Fonte: (Do autor)

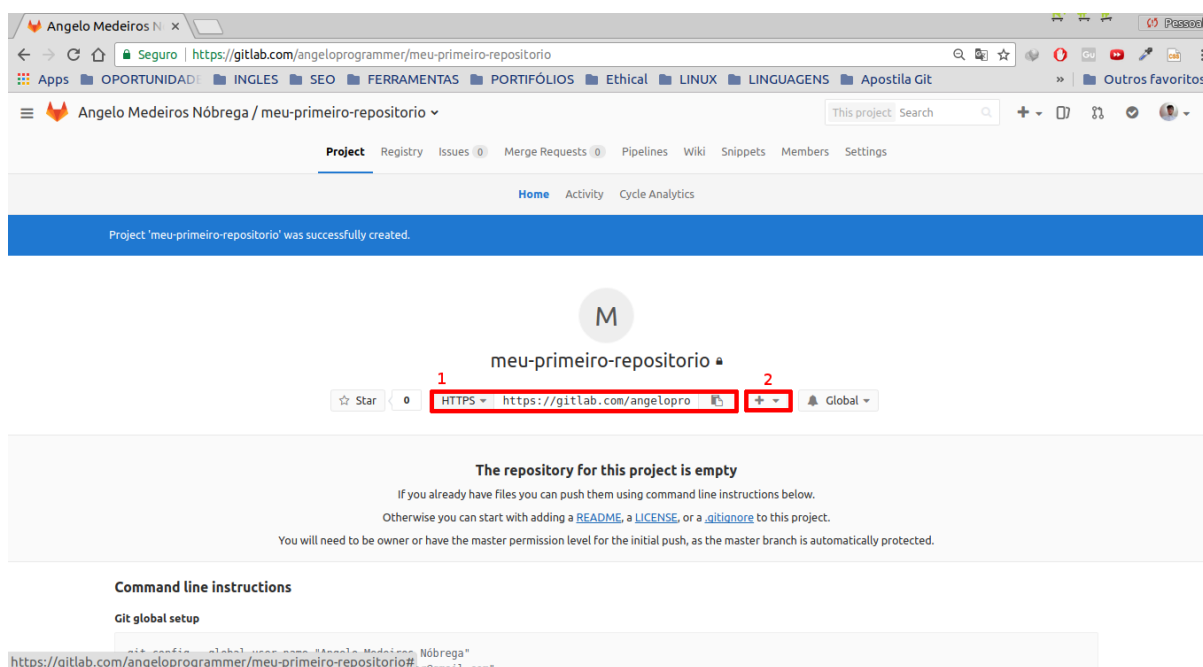
O *git* pode se comunicar com os repositórios remotos através de dois tipos de protocolos, podendo ser o *HTTPS* e o *SSH*. Com o *HTTPS* não é necessário uma configuração prévia (salvas algumas exceções, como o proxy, dependendo do caso) para iniciar seu uso, enquanto o *SSH* só irá funcionar se você adicionar sua chave pública *SSH* nas configurações do *gitLab*.

Em **1** na figura 24, você pode escolher o tipo de protocolo que você irá usar para conectar seu repositório local com o remoto. Em **2** você pode manipular seu repositório remotamente, criando novos arquivos, novos branches, informando problemas, entre outras opções. Lembre-se que as alterações feitas no repositório remoto não altera o conteúdo do repositório local, a não ser que você utilize comandos no *git* para isso, um *git pull* por exemplo (esse comando será descrito mais a frente).

Se você observar o *git* está sempre dando dicas. Com o *gitLab* não é diferente. Um exemplo disso está representado na figura 25. O *gitLab* exibe diversas opções para você conectar seu repositório remoto com o repositório local.

Na primeira etapa da figura 25 você irá realizar a configuração inicial do git, se

Figura 24 – Criando repositório - 3ª etapa



Fonte: (Do autor)

você vem seguindo a apostila desde os primeiros capítulos, esse processo de configuração já foi realizado na seção 3.1. As etapas 2, 3 e 4, serão casos situacionais.

A etapa 2 deve ser usada quando não existe um repositório local. A primeira linha da etapa 2 irá criar um repositório local, a partir do repositório remoto, na pasta atual em que seu console se encontra. A segunda linha serve para acessar a pasta do repositório. A terceira linha (opcional) criará um arquivo em branco com o nome *README.md*. A quarta linha como já devem saber adiciona o arquivo *README.md* para o terceiro estágio. A quinta linha realizará o commit. E a última linha irá adicionar o arquivo *README.md* que foi criado localmente para o repositório remoto, ao *branch master* remoto.

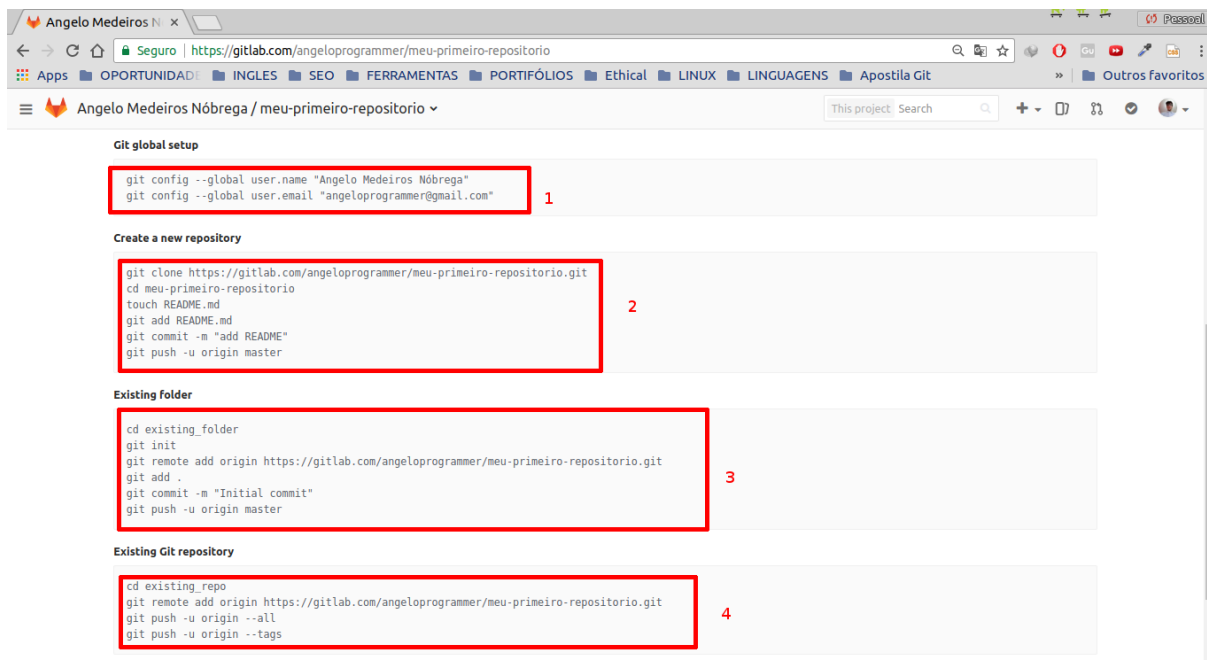
A etapa 3 deve ser utilizada quando você quer iniciar um repositório local e, em seguida fazer a conexão do repositório criado localmente com o repositório remoto. A primeira linha serve para acessar a pasta do repositório. A segunda é utilizada para criar o repositório local. A terceira linha será responsável em estabelecer a conexão entre o repositório local e o remoto. As linhas seguintes vocês já sabem suas devidas aplicações.

A etapa 4 deve ser usada quando já existe um repositório criado localmente. Novamente, a segunda linha irá estabelecer a conexão entre os repositórios. A terceira linha adicionará todos os branches locais ao repositório remoto. A quarta linha (opcional) irá adicionar todas as tags criadas, caso existam.

Lembre-se de verificar os tipos de protocolos que estão utilizando. Se tiver em

dúvida em qual utilizar opte pelo protocolo *HTTPS*. Lembre-se também de configurar o proxy do seu local de trabalho caso seja necessário.

Figura 25 – Criando repositório - 4ª etapa



Fonte: (Do autor)

6.2 Realizando seu primeiro *push*

O *push* (empurrar) é o ato de fazer o upload do seu projeto local para um repositório remoto. O *push* só funcionará corretamente se o seu repositório local estiver conectado com o repositório remoto. Essa conexão é realizada com um comando do tipo `git remote add origin ENDEREÇO_DO_REPOSITÓRIO_REMOTO`, esse endereço encontra-se na primeira etapa da figura 24. O *origin* não faz parte do comando, ele é o nome padrão dados aos repositórios remotos, mas nada impede de você utilizar outro nome, o aconselhado pela comunidade é manter como *origin*.

Os principais comandos para realizar um *push* serão descritos a seguir:

- `git push -u origin master`, sobe o *branch master* para o repositório remoto, usei como exemplo o *master*, mas você pode usar qualquer outro *branch* que tenha criado;
- `git push -u origin cadastroDeUsuarios`, sobe o *branch* cadastro de usuários;
- `git push -u origin --tags`, sobe as tags criadas (haverá uma seção apenas para tags).

6.3 Realizando seu primeiro clone

Realizar um clone é necessário quando você precisa duplicar um repositório remoto para sua máquina. Por exemplo, quando você deleta seu repositório local, ou quando você encontra um projeto opensource e quer utilizá-lo, ou ainda mesmo quando alguém rouba sua máquina e seu projeto está nele, mas nesse último caso só irá funcionar se você estiver usando o git corretamente.

Para fazer um clone é necessário apenas duas condições, o endereço do repositório e obviamente permissão para fazer tal processo. Se o projeto for público você pode fazer o clone do repositório sem pedir a permissão ao dono. Abaixo segue os dois principais comandos para realizar um clone:

- `git clone ENDEREÇO_DO_REPOSITÓRIO_REMOTO`
- `git clone https://gitlab.com/.../meu-primeiro-repositorio.git`, irá fazer um clone e copiar seus arquivos para a pasta com o nome do repositório, nesse caso *meu-primeiro-repositorio.git*;
- `git clone https://gitlab.com/.../meu-primeiro-repositorio.git Test1`, irá fazer o mesmo que o comando anterior, porém irá adicionar os arquivos para a pasta *Test1*.

6.4 Criando um branch a partir do repositório remoto

Vocês já sabem criar um *branch* a partir de um *branch* local. Nessa seção vamos supor que você tenha criado um branch remotamente, como mostra na segunda etapa da figura 24. Por padrão o git cria um *branch* baseado no seu *branch* atual com o comando `git checkout -b nomeDoBranch`, o comando para criar um branch local a partir de um branch remoto é muito semelhante, ele está descrito abaixo:

- `git checkout -b test1 origin/test2`, cria um branch localmente com o nome *test1* a partir do *branch* remoto *test2*, o comando para visualizar o nome do branch remoto foi descrito na seção 5.2.

6.5 Realizando seu primeiro *pull*

O *pull* faz o trabalho inverso do *push*, enquanto o *push* sobe seu arquivos do repositório local para o repositório remoto, o *pull* desce os arquivos que estão no repositório remoto para o repositório local.

Os principais comandos para realizar um pull serão descritos a seguir:

- `git pull origin master`, desce os arquivos do *branch master* localizado remotamente para o *branch* local atual;
- `git pull`, esse comando é usado quando para atualizar a lista de branches, por exemplo, vamos supor que você tenha criado um *branch* remotamente, esse novo *branch* criado não será exibido na sua máquina local até você atualizar a lista (o comando para visualizar os branches está descrito na seção 5.2).

6.6 Trabalhando com *tags*

As tags são os rótulos dados para as versões. Geralmente esses rótulos são numéricos, 1.2.1 por exemplo, vocês provavelmente já viram algumas tags de pré-lançamento (pre-release) usando nomes de letras gregas, versão 1.0.0-alpha ou versão 1.0.0-beta. Para manter um nível de organização, iremos adotar as regras do versionamento semântico para a criação das tags, tais regras são adotadas pela maioria das empresas, comunidades e desenvolvedores.

6.6.1 Versionamento Semântico

O versionamento semântico consiste em um conjunto de 11 regras bem especificadas. Eu irei falar a essência do versionamento semântico, e fica como pesquisa para vocês saberem mais na [documentação](#) completa.

Abaixo está descrito um resumo do que consiste o versionamento semântico. Dado um número da versão `MAJOR.MINOR.PATCH`:

1. versão Maior(MAJOR): quando fizer mudanças incompatíveis na API;
2. versão Menor(MINOR): quando adicionar funcionalidades mantendo compatibilidade;
3. versão de Correção(PATCH): quando corrigir falhas mantendo compatibilidade.

A versão MINOR tem que garantir compatibilidade apenas com a versão MAJOR. A versão de correção (PATCH) tem que garantir compatibilidade com todas as versões acima dela. Se por exemplo, ocorrer quebra de compatibilidade quando a correção de um *bug* for feita, deve ser criada uma nova versão MINOR. Esses e outros detalhes estão especificados na documentação.

Rótulos adicionais para pré-lançamento(pre-release) e metadados de construção(build) estão disponíveis como extensão ao formato `MAJOR.MINOR.PATCH`.

6.6.2 Criando *tags* com o *git*

Para criar uma *tag* usando o *git* basta usar o comando

```
git tag X.Y.Z
```

onde as letras X, Y e Z devem ser substituídas pelo número da versão seguindo as regras do versionamento semântico visto na seção 6.6.1.

A seguir estão descritos outros comandos relacionados com as tags:

- `git tag -l`, exibe as tags criadas;
- `git tag`, outra maneira de exibir as tags criadas;
- `git push --tags`, sobe as tags criadas para o repositório remoto.

6.7 Resolvendo conflitos

Nem tudo são flores quando falamos do controle de versões. As vezes acontece de ter mais de um desenvolvedor editando o mesmo código, e quando é necessário fazer um merge, o sistema de controle de versão não tem como advinhar qual código deve ser mantido. Existem diversos cenários para falar sobre conflitos.

Esse assunto é tão complexo que cada empresa tem uma maneira diferente de lidar. Para exemplificar isso, existem dois tipos de visões, uma é a visão pessimista, onde o código só pode ser editado por um desenvolvedor, e os outros podem apenas ler o arquivo, ou seja, não possuem permissão de escrita. A vantagem disso é que, nesse cenário não existe o risco de ocorrer tais conflitos. Já a visão otimista é aquela onde o código pode ser editado por diversos desenvolvedores e, quando chegar na hora de realizar o merge, o desenvolvedor irá comparar os códigos decidindo quais mudanças deverão permanecer. A vantagem nessa ótica otimista é que a evolução do projeto torna-se algo mais fluida.

Vamos imaginar o seguinte cenário, existem dois branches idênticos, o *branch exemplo0* e o *branch exemplo1*, ambos com um arquivo em branco chamado *wtf*. O desenvolvedor A tem acesso ao *branch exemplo0* e o desenvolvedor B ao outro *branch*.

O desenvolvedor A adiciona ao arquivo *wtf*, o seguinte conteúdo e depois faz um commit:

```
Java é a melhor linguagem de todas,  
ela é super rápida, fácil de aprender e leve.  
JAVA É VIDA! O NOME DO MEU FILHO VAI SER JAVA!
```

O outro desenvolvedor adiciona o seguinte conteúdo ao arquivo `wtf` e também faz um commit:

Python é legal.

Agora existem dois *branches* com o mesmo arquivo, mas com conteúdos diferentes (ver figura 26) e, em *commits* distintos.

Figura 26 – Resolvendo conflito - parte 1

```
u-angelo@AngeloUbuntu:~/Documentos/LINGUAGENS/Latex/Apostila de métodos (exemplo1)$ cat wtf
Python é legal.
u-angelo@AngeloUbuntu:~/Documentos/LINGUAGENS/Latex/Apostila de métodos (exemplo1)$ git checkout exemplo0
Switched to branch 'exemplo0'
u-angelo@AngeloUbuntu:~/Documentos/LINGUAGENS/Latex/Apostila de métodos (exemplo0)$ cat wtf
Java é a melhor linguagem de todas,
ela é super rápida, fácil de aprender e leve.
JAVA É VIDA! O NOME DO MEU FILHO VAI SER JAVA!
u-angelo@AngeloUbuntu:~/Documentos/LINGUAGENS/Latex/Apostila de métodos (exemplo0)$
```

Fonte: (Do autor)

O desenvolvedor A resolve fazer um *merge* do *branch* do desenvolvedor B, e acaba se deparando com a seguinte mensagem (ver figura 27):

CONFLITO (adicionar/adicionar): conflito de mesclagem em wtf

Figura 27 – Resolvendo conflito - parte 2

```
u-angelo@AngeloUbuntu:~/Documentos/LINGUAGENS/Latex/Apostila de métodos (exemplo0)$ git merge exemplo1
Mesclagem automática de wtf
CONFLITO (adicionar/adicionar): conflito de mesclagem em wtf
Automatic merge failed; fix conflicts and then commit the result.
u-angelo@AngeloUbuntu:~/Documentos/LINGUAGENS/Latex/Apostila de métodos (exemplo0|MERCING)$ git status
```

Fonte: (Do autor)

Se vocês observarem, o branch atual após o merge passou a ser *exemplo0/MERCING*. Nessa branch o conteúdo do arquivo `wtf` (ver figura 28), é a união dos dois conteúdos, e o desenvolvedor terá que decidir qual código irá permanecer.

O desenvolvedor A então edita o arquivo `wtf` e mantém o conteúdo do desenvolvedor B. Após alterar o conteúdo ele faz um novo *commit* e, todos os conflito são resolvidos, tanto do código, quanto o “conflito” do desenvolvedor A (piada muito boa, eu sei!), ver figura 29.

Figura 28 – Resolvendo conflito - parte 3

```
u-angelo@AngeloUbuntu:~/Documentos/LINGUAGENS/Latex/Apostila de métodos (exemplo0|MERCING)$ cat wtf
<<<<<<< HEAD
Java é a melhor linguagem de todas,
ela é super rápida, fácil de aprender e leve.
JAVA É VIDA! O NOME DO MEU FILHO VAI SER JAVA!
=====
Python é legal.
>>>>>>> exemplo1 (HEAD)
u-angelo@AngeloUbuntu:~/Documentos/LINGUAGENS/Latex/Apostila de métodos (exemplo0|MERCING)$
```

Fonte: (Do autor)

Figura 29 – Resolvendo conflito - parte 4

```
u-angelo@AngeloUbuntu:~/Documentos/LINGUAGENS/Latex/Apostila de métodos (exemplo0|MERCING)$ vim wtf
u-angelo@AngeloUbuntu:~/Documentos/LINGUAGENS/Latex/Apostila de métodos (exemplo0|MERCING)$
u-angelo@AngeloUbuntu:~/Documentos/LINGUAGENS/Latex/Apostila de métodos (exemplo0|MERCING)$ git add .
u-angelo@AngeloUbuntu:~/Documentos/LINGUAGENS/Latex/Apostila de métodos (exemplo0|MERCING)$ git commit -m "Resolvendo conf
lito no arquivo wtf"
[exemplo0 3eaf03b] Resolvendo conflito no arquivo wtf
u-angelo@AngeloUbuntu:~/Documentos/LINGUAGENS/Latex/Apostila de métodos (exemplo0)$ cat wtf
Python é legal.
u-angelo@AngeloUbuntu:~/Documentos/LINGUAGENS/Latex/Apostila de métodos (exemplo0)$
```

Fonte: (Do autor)

Trabalhando com o Git-flow

Na seção 1.3 falei brevemente do git-flow. Nesse capítulo entraremos em mais detalhes nessa incrível ferramenta que nos ajudará no desenvolvimento baseado na estratégia de ramificação apresentada no capítulo 1.

Como já foi dito, o *git-flow* é apenas uma ferramenta para facilitar nossas vidas, você não é obrigado a usá-la, desde que siga as boas práticas e, aplique a estratégia de ramificação proposta.

Lembre-se também que o *git* sempre estará presente, tarefas como fazer *commits*, visualizar o *status*, visualizar o *log*, trocar de *branch*, entre outras coisas, ainda serão realizadas apenas com o *git*. Também nada impede você usar os outros comandos do git que existem no git-flow.

Nas próximas seções irei descrever as mesmas funcionalidades que foram ensinadas com o git, mas usando comandos do git-flow, sempre mostrando as suas vantagens.

7.1 Criando o repositório usando o *git-flow*

O comando para criar um repositório usando o *git-flow* é quase idêntico ao do *git*. O comando

```
git flow init
```

iniciará o repositório, mas antes fará uma série de perguntas (ver figura 30), por padrão iremos deixar como estão, apertando enter para cada pergunta.

7.2 Criando features

Os *branches features* serão criados sempre que uma nova funcionalidade precise ser desenvolvida. Elas sempre serão iniciadas do branch develop. O comando para iniciar uma nova feature é:

- `git flow feature start nomeDaFuncionalidade`, esse comando irá criar um novo *branch* chamado *nomeDaFuncionalidade*;
- `git flow feature start telaDeLogin`, cria o *branch* *telaDeLogin*;

Figura 30 – Iniciando repositório com o *Git-flow*

```
u-angelo@AngeloUbuntu:~/Documentos/LINGUAGENS/JavaScript$ git flow init
Initialized empty Git repository in /home/u-angelo/Documentos/LINGUAGENS/JavaScript/.git/
No branches exist yet. Base branches must be created now.
Branch name for production releases: [master]
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
Hooks and filters directory? [/home/u-angelo/Documentos/LINGUAGENS/JavaScript/.git/hooks]
u-angelo@AngeloUbuntu:~/Documentos/LINGUAGENS/JavaScript (develop)$
```

Fonte: (Do autor)

Diferente do *git*, independente de qual *branch* você esteja, o *branch feature* sempre será iniciado do *branch develop*.

Para fazer um *push* de um *branch feature* você usará o seguinte comando:

- `git flow feature publish nomeDaFuncionalidade`, esse comando irá fazer o *push* do *branch* chamado *nomeDaFuncionalidade*;
- `git flow feature publish telaDeLogin`, faz o *push* do *branch telaDeLogin*;

Para fazer o *pull* aconselho usar o *git* de maneira usual. Mas pode ser feito usando o *git-flow* com o comando:

- `git flow feature pull origin nomeDaFuncionalidade`, esse comando irá fazer o *pull* do *branch* chamado *nomeDaFuncionalidade*;
- `git flow feature pull origin telaDeLogin`, faz o *pull* do *branch telaDeLogin*;

Após terminar a criação da funcionalidade *telaDeLogin*, é necessário fazer um merge para o *branch develop*, depois apagar o *branch local* e o *branch remoto*, já que você não vai mais precisar deles. Se fosse usar apenas o *git* você teria que fazer cada etapa dessas separadamente, mas o *git-flow* realiza todas essas tarefas com o seguinte comando:

```
git flow feature finish telaDeLogin
```

Perceba que as únicas coisas que mudaram nos comandos foram as partes centrais, *start*, *publish* e *finish*. Esse padrão irá se repetir para os outros tipos de *branches*.

7.3 Criando releases

O *branch release* deve ser criado quando for necessário enviar a aplicação para a produção. Nesse branch serão realizados testes e, após sua finalização ele será mesclado com o *branch master* e o *develop*.

Os comandos são semelhantes ao das features, a diferença está nos nomes dos branches, no caso das releases os nomes serão as tags discutidas na seção 6.6:

- `git flow release start X.Y.Z`, esse comando irá criar um novo *branch* chamado *X.Y.Z*;
- `git flow release publish 1.3.1`, faz o *push* do *branch 1.3.1*;
- `git flow release finish 1.3.1`, finaliza o *branch 1.3.1*;

7.4 Criando hotfixes

O *branch hotfix* deve ser criado quando um *bug* for encontrado no branch de produção, o *master*. Esse *branch* será iniciado no *master* e finalizado no *master* e no *develop*, já que o *branch develop* também precisa sofrer a correção.

Os comandos para o *branch hotfix* seguem os mesmos padrões dos *releases*:

- `git flow hotfix start X.Y.W`, esse comando irá criar um novo *branch* chamado *X.Y.W*;
- `git flow hotfix publish 1.3.2`, faz o *push* do *branch 1.3.2*;
- `git flow hotfix finish 1.3.2`, finaliza o *branch 1.3.2*;

7.5 Criando bugfixes

O *branch bugfix* tem a mesma função do *hotfix*, correção de *bugs*. Ele deve ser criado quando um *bug* for encontrado durante a fase de desenvolvimento, ou seja, no branch *develop*. O *branch bugfix* inicia-se no *develop* e é finalizado somente no *develop*.

Os comandos para o *branch bugfix* seguem os mesmos padrões dos *features*:

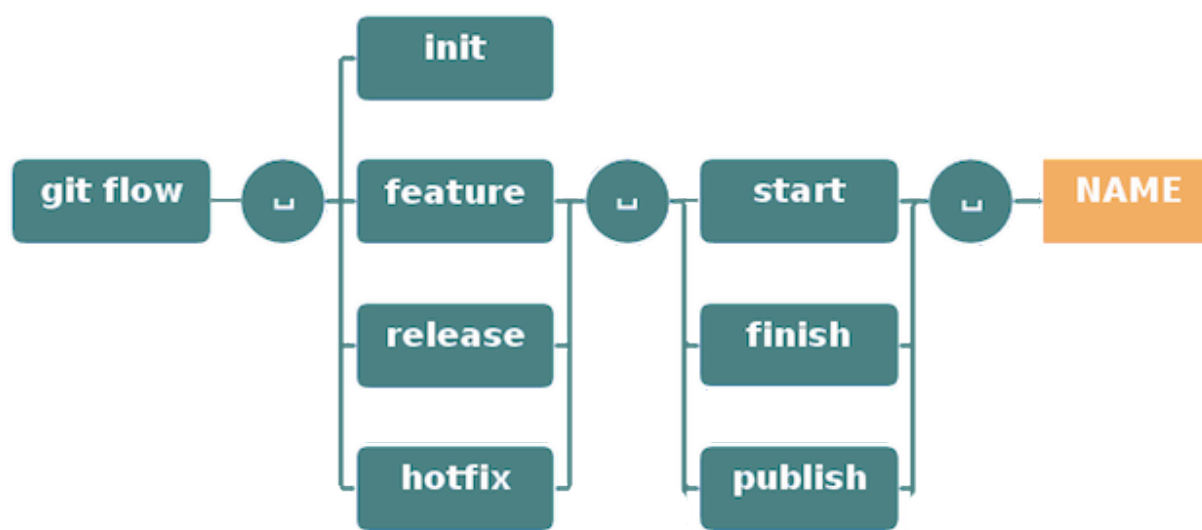
- `git flow bugfix start erroCarrinhoCompras`, esse comando irá criar um novo *branch* chamado *X.Y.W*;
- `git flow bugfix publish erroCarrinhoCompras`, faz o *push* do *branch 1.3.2*;

- `git flow bugfix finish erroCarrinhoCompras`, finaliza o *branch erroCarrinhoCompras*;

7.6 Resumo dos comandos do *git-flow*

A figura 31 abrange quase todos os comandos do *git-flow* de forma resumida, demonstrando a facilidade que é trabalhar com o *git-flow*.

Figura 31 – Resumo dos comandos do *Git-flow*



Fonte: (<https://danielkummer.github.io/git-flow-cheatsheet/index.html>)

Finalizando

Essa apostila teve como objetivo plantar uma sementinha na mente de vocês sobre o controle de versão, suas vantagens e como ele tornará seu desenvolvimento mais ágil e seguro.

Existem diversas outras questões a serem abordadas e outras ferramentas. Agora depende apenas de você, pesquisar mais sobre o assunto e praticar.