O Git, é um sistema de versionamento de software. Através dele, podemos desenvolver softwares versionados de acordo com cada nível de evolução do sistema possibilitando o desenvolvimento remoto e de maneira individual por cada um dos participantes do projeto de desenvolvimento do software, e ao termino do desenvolvimento de cada funcionalidade por cada um dos colaboradores, podemos sincronizar todos os trechos de código em um único software ou versão de software e caso haja alguma coisa errada o Git irá nos apontar onde está o erro, e nos possibilitar controlar e voltar versões anteriores.

A instalação do Git é feita facilmente em sistemas debian based via:

#apt-get install git

para verificar se a instalação foi feita corretamente, digitamos no terminal o comando:

$git

Caso esteja instalado corretamente, o git nos listará varias funcionalidades do programa.

Para gerarmos uma identidade para o novo usuário do git, podemos configurar da seginte forma:

Assim configuramos o nome do usuário:

$git config --global user.name "Nome do Usuario"

Assim configuramos o email do usuário git:

$git config --global user.email "seuemail@email.com"

Uma vez inseridos nome de usuário e e-mail, o git já está pronto para iniciar o versionamento de software

Podemos também configurar o git destacar em cores pontos que devem ser evidenciados para que tenhamos mais atenção em quais quer retornos do git.

$git config --global color.ui true

Por enquanto, vamos trabalhar apenas com essas três funcionalidades básicas para termos uma config inicial caso desejem ver as configurações setadas, liste-as conforme abaixo:

$git config --list

Para seguirmos com nossos exemplos, vamos criar nosso primeiro repositório que nada mais é do que um diretório onde iremos armazenar os arquivos do nosso projeto. Ao criar um controle de repositório do git, os arquivos contidos no diretório em questão iram entrar no nosso sistema de controle de versões. Ou seja, os arquivos serão versionados e tudo que vier a ser modificado desse ponto em diante estar sendo monitorado pelo nosso controle deste repositório do git. Para darmos continuidade, vamos criar um diretório chamado “aulagit” com o comando mkdir (linux).

$mkdir aulagit

Acesse o diretório através do comando cd (change directory).

$cd aulagit

Então, agora vamos gerar o nosso repositório git com o comando git init

$git init

Ao entrar com o comando "git init" o git irá criar um diretório de controle para o diretório corrente este diretório será o ".git" (diretório oculto Linux) nesse diretório o git irá gerar vários subdiretórios e arquivos, contendo a estrutura para que ele faça o nosso sistema de controle de versão ou repositório. Caso deseje listar o conteúdo do seu diretório do Git, basta digitar listar o conteúdo do diretório.

$ls .git

Agora que ja temos uma instancia do controle de versões no nosso diretório de trabalho "./aulagit"

vamos criar nosso primeiro arquivo no diretório para testarmos o nosso controle de versão do git.

$touch teste1.txt

Depois de criar nosso primeiro arquivo no diretório de trabalho, vamos verificar o status do nosso git.

$git status

No ramo master

Submissão inicial.

Arquivos não monitorados:

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

teste1.txt

Nada adicionado ao envio mas arquivos não registrados estão presentes (use "git add" to registrar)

Observe que agora temos um arquivo no primeiro estágio do controle de versão o "untracked files" ou seja, temos um arquivo novo no diretório gerenciado pelo controle de versão. Mas, o arquivo de fato ainda não está sendo monitorado pelo git.

Para adicionarmos o arquivo ao monitoramento do controle de versões, vamos usar o comando "git add teste1.txt" como sugerido pelo retorno do comando “git status”.

$git add teste1.txt

Então vamos novamente verificar o status do git.

$git status

No ramo master

Submissão inicial.

Mudanças a serem submetidas:

(utilize "git rm --cached <arquivo>..." para não apresentar)

new file: teste1.txt

Agora verifique que o nosso arquivo está no segundo estágio do controle de versão "changes to be commited". Ou seja, mudanças a serem submetidas ao controle de versão. Arquivos que estão neste estágio estão prontos para entrar efetivamente no controle de versão, onde o git irá gerar um hash que irá conter as informações sobre essa versão de edição do arquivo.

Assim, vamos enviar o arquivo para o nosso terceiro estágio do controle de versão do git. Este estágio manda de fato os arquivos para o controle de versão e podemos inserir um comentário no ato do commit para servir de descrição dessa etapa de evolução do desenvolvimento.

$git commit -m "meu primeiro commit"

[master (root-commit) 16c60be] meu primeiro commit

1 file changed, 1 insertion(+)

create mode 100644 teste1.txt

Então o git irá nos informar que temos um novo arquivo em nosso controle de versão; e ao verificar novamente o status do git através do comando "git status" ele irá retornar a mensagem “nothing to be commited” (nada a submeter). Pois o nosso arquivo já passou pelas três fazes do commit.

$git status

No ramo master

nada a submeter, diretório de trabalho vazio

Para conferirmos nosso log do git, basta digitar o comando "git log" e então o git irá nos retornar o histórico dos nossos commits mostrando detalhes como: Hash gerado para o commit, autor do commit, as datas dos mesmos e as mensagens que foram inseridas em cada commit. Agora vamos modificar o nosso teste1.txt e ver o que o git nos informa.

$vim teste1.txt

1 minha primeira mudança em um aquivo gerenciado pelo git

<esc>:wq!

Agora, ao digitarmos o comando git status, vamos ver que o teste1.txt voltou para o estagio “changes not staged for commit”.

$git status

No ramo master

Changes not staged for commit:

(utilize "git add <arquivo>..." para atualizar o que será submetido)

(utilize "git checkout -- <arquivo>..." para descartar mudanças no diretório de trabalho)

modified: teste1.txt

nenhuma modificação adicionada à submissão (utilize "git add" e/ou "git commit -a")

Teremos que usar novamente o comando git add teste1.txt. e depois git commit - m "Modificando o arquivo teste1.txt"

$git add teste1.txt

$git commit -m "Modificando o arquivo teste1.txt"

[master 92e3325] Modificando o arquivo teste1.txt

1 file changed, 1 insertion(+), 1 deletion(-)

Assim, teremos novamente o arquivo teste1.txt submetido ao nosso controle de versão e ao digitar o comando “git status”, vamos observar que o git irá nos retornar a mensagem de nada para ser submetido (commitado). E ao digitar o comando git log vamos verificar que agora os logs vão listar nosso primeiro e segundo commit. Ou seja, a segunda versão do nosso teste1.txt.

$git log

commit 92e332598e26cc00f9f9cd9cd9190f4533f2dee6

Author: neto.silva <alnetosilva@gmail.com>

Date: Sat Dec 12 21:59:14 2015 -0200

Modificando o arquivo teste1.txt

commit 16c60be6a7166101976a0fd6fc0a0afc76880b34

Author: neto.silva <alnetosilva@gmail.com>

Date: Sat Dec 12 21:37:24 2015 -0200

meu primeiro commit

Observem que nosso arquivo entrou no controle de versão localhost sem a necessidade de um repositório externo.

Vamos criar um novo arquivo para entrar no nosso controle de versão, vamos chamar-lo de teste2.txt

$vim teste2.txt

segundo arquivo.txt

<esc>:wq!

Agora verifique o status do repositório git.

$git status

No ramo master

Arquivos não monitorados:

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

teste2.txt

nada adicionado ao envio mas arquivos não registrados estão presentes (use "git add" to registrar)

Ele irá informar que não ha nada para ser adicionado ao envio (commit). No entanto, temos um novo arquivo para ser versionado; então, vamos adicionar ele ao controle de versão.

$git add teste2.txt

$git status

No ramo master

Mudanças a serem submetidas:

(use "git reset HEAD <file>..." to unstage)

new file: teste2.txt

Então receberemos a mensagem de que há mudanças para ser enviadas (commitadas).

Neste estágio, temos formas diferentes de enviar as alterações ao controle de versão:

A primeira é através do comando abaixo git commit -m "descricao breve do commit" para fazermos uma breve descrição do commit. Ou seja, trabalharmos de forma rápida como já fizemos anteriormente.

A segunda nos permite ser mais detalhistas ao descrever as alterações enviadas (commit).

$git commit

1 - inserindo o arquivo teste2.txt ao controle de versão

2 - Meu segundo commit

3 # Please enter the commit message for your changes. Lines starting

4 # with '#' will be ignored, and an empty message aborts the commit.

5 # No ramo master

6 # Mudanças a serem submetidas:

7 # new file: teste2.txt

8 #

<Esc>:wq!

Dessa forma, o git irá abrir o seu aditor padrão (no meu caso o vim) já informando quais arquivos foram enviados. Assim, poderemos descrever de forma mais detalhada as alterações efetuadas nesse commit (tente ser simples e objetivo). Então, ao inserir o comando “git log” vamos verificar os logs do nosso git; que no retorno do commando irá descrever de forma básica desde o commit mais atual até o mais antigo. Que serão exibidos do modo como descrevemos através do editor padrão.

$git log

commit d6edb736ca0c9b926637559e654793d909034589

Author: neto.silva <alnetosilva@gmail.com>

Date: Sat Dec 12 23:43:48 2015 -0200

- inserindo o arquivo teste2.txt ao controle de versão

- Meu segundo commit

commit 92e332598e26cc00f9f9cd9cd9190f4533f2dee6

Author: neto.silva <alnetosilva@gmail.com>

Date: Sat Dec 12 21:59:14 2015 -0200

Modificando o arquivo teste1.txt

commit 16c60be6a7166101976a0fd6fc0a0afc76880b34

Author: neto.silva <alnetosilva@gmail.com>

Date: Sat Dec 12 21:37:24 2015 -0200

meu primeiro commit

Edite os arquivos teste1.txt e teste2.txt e verifique o status do git.

$vim teste1.txt

1 Alterando minha primeira mudança em um aquivo gerenciado pelo git

<Esc>:wq!

$vim teste2.txt

1 segundo arquivo txt

2 incluindo uma linha ao arquivo2.txt

$git status

No ramo master

Changes not staged for commit:

(utilize "git add <arquivo>..." para atualizar o que será submetido)

(utilize "git checkout -- <arquivo>..." para descartar mudanças no diretório de trabalho)

modified: teste1.txt

modified: teste2.txt

nenhuma modificação adicionada à submissão (utilize "git add" e/ou "git commit -a")

Caso deseje ignorar e desfazer as mudanças feitas no arquivo teste2.txt e commitar somente as mudanças feitas no arquivo teste1.txt execute o commando “git checkout -- <arquivo>” como sugerido no retorno do “git status”. Então o arquivo2.txt irá voltar ao seu estado antes da edição e não entrará no próximo commit.

Caso deseje somente ignorar as mudanças feitas no arquivo teste2.txt para este commit. No entanto, preservar as alterações feitas para enviar em um commit posterior, faça da seguinte forma.

Primeiro adicione todos os arquivos editados.

$git add .

Depois de adicionar, volte somente o arquivo teste2.txt à unstaged.

$ git reset HEAD teste2.txt

Unstaged changes after reset:

M teste2.txt

Ao verificar o status do git, veremos que temos dois arquivos em duas etapas diferentes.

$ git status

No ramo master

Mudanças a serem submetidas:

(use "git reset HEAD <file>..." to unstage)

modified: teste1.txt

Changes not staged for commit:

(utilize "git add <arquivo>..." para atualizar o que será submetido)

(utilize "git checkout -- <arquivo>..." para descartar mudanças no diretório de trabalho)

modified: teste2.txt

Desta forma, somente o arquivo teste1.txt será enviado ao executarmos o commit.

Devolva o arquivo teste1.txt para o status de untaged com o comando “reset HEAD” e veja que agora, o git irá nos mostrar que temos 2 arquivos modificados. Agora vamos enviar nosso commit de forma resumida passando o parametro “-a = git add” que irá adicionar os arquivos ao controle de versão no ato do commit com nossas modificações no diretório. No entanto, se houver arquivos novos eles devem ser adicionados primeiramente com o “git add .”

$git commit -a -m "mensagem”

ou somente

$git commit -a

e digitamos no nosso editor padrão a descrição do nosso commit.

A partir de agora, vamos trabalhar com a ferramenta git log. Ao usar o comando “git log” sem parametros, temos uma lista com todos os nossos commits, desde o mais atual até o mais remoto. No entanto, para verificarmos o que foi feito em cada arquivo detalhadamente para cada commit, executamos o commando:

$git log -p

commit 621a22adc202b453dbd468cd5710bfed393cc56a

Author: neto.silva <alnetosilva@gmail.com>

Date: Sun Dec 13 00:32:37 2015 -0200

Modificando teste1.txt e teste2.txt

diff --git a/teste1.txt b/teste1.txt

index 7d1312d..eed3ad3 100644

--- a/teste1.txt

+++ b/teste1.txt

@@ -1 +1,2 @@

minha primeira mudança em um aquivo gerenciado pelo git

+adicionando uma linha

diff --git a/teste2.txt b/teste2.txt

index 62a4935..9ada19e 100644

--- a/teste2.txt

+++ b/teste2.txt

@@ -1 +1,2 @@

segundo arquivo txt

+incluindo uma linha ao arquivo2.txt

commit d6edb736ca0c9b926637559e654793d909034589

Author: neto.silva <alnetosilva@gmail.com>

Date: Sat Dec 12 23:43:48 2015 -0200

- inserindo o arquivo teste2.txt ao controle de versão

- Meu segundo commit

diff --git a/teste2.txt b/teste2.txt

new file mode 100644

index 0000000..62a4935

--- /dev/null

+++ b/teste2.txt

@@ -0,0 +1 @@

+segundo arquivo txt

commit 92e332598e26cc00f9f9cd9cd9190f4533f2dee6

Author: neto.silva <alnetosilva@gmail.com>

Date: Sat Dec 12 21:59:14 2015 -0200

Modificando o arquivo teste1.txt

diff --git a/teste1.txt b/teste1.txt

index 0298ce8..7d1312d 100644

--- a/teste1.txt

+++ b/teste1.txt

@@ -1 +1 @@

-teste 1

+minha primeira mudança em um aquivo gerenciado pelo git

commit 16c60be6a7166101976a0fd6fc0a0afc76880b34

Author: neto.silva <alnetosilva@gmail.com>

Date: Sat Dec 12 21:37:24 2015 -0200

meu primeiro commit

diff --git a/teste1.txt b/teste1.txt

new file mode 100644

index 0000000..0298ce8

--- /dev/null

+++ b/teste1.txt

@@ -0,0 +1 @@

+teste 1

Assim, veremos nas linhas iniciadas com "-" as linhas que foram removidas do nosso arquivo e com "+" as linhas que foram adicionadas em nossos arquivos

caso queira mostrar apenas os ultimos commits, vamos passar a quantidade de ultimos commits que queremos ver

$git log -p -2

No comando acima mostramos apenas os dois ultimos commits.

Caso deseje verificar de forma resumida apenas as estatisticas de alterações em nosso repositorio, vamos digitar o comando

$git log --stat

Assim, veremos apenas a quantidade de alterações, inserções feitas em cada arquivo de forma resumida

Podemos fazer a visualização de logs de forma mais resumida ainda onde cada commit será exibido de maneira resumida em uma unica linha

$git log --pretty=oneline

Podemos trazer tambem as estatisticas do nosso log de forma mais resumida ainda e formatada de acordo com a nossa necessidade

$git log --pretty=format:"%h - %an, %ar : %s"

Assim veremos a seguinte formatação:

"rash - autor do commit, tempo desde o commit : descrição inserida pelo usuário na hora do commit"

Podemos também, recuperar os logs de um determinado período. Por exemplo para recuperarmos os logs dos últimos 2 dias

$git log --since=2.days

Podemos usar "days, weeks, mounts" etc...

Normalmente, para desenvolvermos grandes projetos usamos IDE's como netbeans, eclipse entre outras. E essas IDE's criam seus arquivos de controle dentro do nosso diretório de códigos fontes (o próprio fonte do projeto). Ao enviar esses projetos para serem desenvolvidos em equipe, podemos ignorar os arquivos da ide ou quaisquer outros do controle de versão. Assim, vamos criar um arquivo no diretório chamado .gitignore e nele inserir os arquivos ou diretórios que devem ser ignorados, cada um em uma linha.

$vim .gitignore

veja que agora ao executar o commando "git status" veremos que o git mostrou que temos o arquivo .gitignore como untrackd files. Podemos inserir o .gitignore no controle de versão.

$git add .

assim inserimos todas as ultimas alterações para o controle de versão.

$git commit -m "adicionando .gitignore" ou controle de versão.

e ao mandar um $git log veremos que o log irá mostrar que o ultimo commit inseriu o arquivo .gitignore

assim, vamos inserir no nosso gitignore o nosso gitignore

$vim gitignore

./.gitignore

<Esc>:wq!

$git status

e observe que agora não temos mais arquivos como untracked files!

Para fixarmos ainda mais o conhecimento, vamos agora praticar o exercício abaixo

Como podemos ignorar um arquivo no controle de versão?

A-.git

b-.ignore

c-.profile

d-.gitignore

Qual o comando para seguir no processo de um COMMIT quando o arquivo esta no estagio untracked files?

a - git reset HEAD

B - git commit -m

c - git add

d - git checkout --

Qual o comando para verificar o detalhamento completo dos commits?

a - git log -p

b - git log --stat

c - git log

d - git log --pretty=online

Qual a função do parâmetro -a no momento de um commit?

a - adicionar automaticamente os arquivos no momento do commit

b - fazer um commit automático sem mensagens

c - adicionar uma descrição no commit

d - nem uma das anteriores

Agora vamos desfazer as alterações do git. Para isso, vamos criar um arquivo chamado teste2.txt

$vim teste2.txt

teste2

<ctrl>wq!

$git status

observe que agora o git mostrou que temos um arquivo no status de untracked files o nosso teste2.txt; vamos adicionar ele ao controle de versão, porem não vamos enviar ainda.

$git add teste2.txt

$git status

observe que ao verificar o status do git ele mostra que existem mudanças a serem enviadas. No entanto, caso desejemos remover o arquivo que acabou de ser adicionado ao controle de versões mas não foi enviado ainda, podemos mandar o comando:

$git reset HEAD arquivo2.txt

$git status

e observar que o arquivo teste2.txt saiu do status changes to be commited e voltou ao status de untracked files.

para termos uma noção mais a fundo da funcionalidade do commando reset HEAD, vamos criar dois novos arquivos

$touch teste{3,4}.txt

$git status

observe agora, que temos vários arquivos no status de untracked files. e para não necessitarmos enviar um a um, vamos incluir todos de uma vez no nosso controle de versão e depois remover somente o arquivo que desejarmos

$git add .

$git status

observe que todos os novos arquivos estão no status changes to be commited. No entanto, desejamos commitar somente os teste2.txt e teste3.txt e ignorar o arquivo teste4.txt. Para isso, vamos remover o arquivo teste4.txt do status changes to be commited:

$git reset HEAD teste4.txt

$git status

Observe agora, que temos o arquivo teste4.txt como untracked files e os arquivos teste2.txt e teste3.txt com mudanças para ser enviadas (changes to be commited).

Vamos voltar também os status dos teste 2 e 3.txt e serguir para nossa próxima etapa onde vamos voltar commits.

$git reset HEAD teste2.txt

$git reset HEAD teste3.txt

Vamos então commitar todos os nossos arquivos de uma vez atraves do -a:

$git commit -a -m “commitando tudo para voltar versões”

Uma vez feito isso, vamos usar o commando git checkout para voltar a versão anterior a criação dos arquivos teste3.txt e teste4.txt. Para isso vamos executar o comando git log para recuperarmos o rach do commit que queremos restaurar.

$git log --pretty=oneline -p -2

706191aae243ba36f2e5233f546c7b5ad6cb1259 Inserindo teste3 e 4 .txt

**diff --git a/teste3.txt b/teste3.txt**

**new file mode 100644**

**index 0000000..e69de29**

**diff --git a/teste4.txt b/teste4.txt**

**new file mode 100644**

**index 0000000..e69de29**

d03b619467cc2ef9413604a70a16a0ff467d8cd6 editando inserindo a terceira quarta e quinta linha

**diff --git a/teste2.txt b/teste2.txt**

**index ae22f21..c6a7830 100644**

**--- a/teste2.txt**

**+++ b/teste2.txt**

@@ -1,5 +1,5 @@

Modificando completamente teste2.txt

Editando teste2.txt apos a segunda modificação

-Inserindo terceira linha

-quarta linha

-quinta linha

+Inserindo terceira linha!

+quarta linha!

+quinta linha!

E então voltaremos a nossa penultima versão do nosso projeto:

$git checkout d03b6

Warning: you are leaving 1 commit behind, not connected to

any of your branches:

706191a Inserindo teste3 e 4 .txt

If you want to keep it by creating a new branch, this may be a good time

to do so with:

git branch <new-branch-name> 706191a

HEAD is now at d03b619... editando inserindo a terceira quarta e quinta linha

$ls

teste1.txt teste2.txt

Observe que então o Head (cabeça do nosso projeto) voltou ao ponto do commit anterior e que o git criou um <new-branch-name> 706191a que contem a nossa utima versão do projeto. Veremos mais sobre isso mais adiante. Vamos voltar nosso projeto para a versão final. Passando o hash do branch ou somente “master” para voltar ao mais atual.

$git checkout 706191

Previous HEAD position was d03b619... editando inserindo a terceira quarta e quinta linha

HEAD is now at 706191a... Inserindo teste3 e 4 .txt

E então nosso projeto voltará para a ultima versão de desenvolvimento.

$ ls

teste1.txt teste2.txt teste3.txt teste4.txt

Podemos usar o git reset HEAD também para remover commits do nosso log sem descartar as alterações feitas no arquivos seguinte forma.

$git reset HEAD~2 --soft

Remove os dois últimos commits.

No entanto se quisermos remover os commits sem criar branchs e descartar as alterações feitas depois dos níveis passados, usamos o parâmetro --hard.

Vamos commitar novamente os arquivos.

$git add .

$git commit -m “commitando tudo novamente”

$ git status

HEAD detached from 706191a

Mudanças a serem submetidas:

(use "git reset HEAD <file>..." to unstage)

new file: teste3.txt

new file: teste4.txt

Observe que o status do git nos informa que o HEAD está no commit 70611, que voltamos com o reset HEAD soft. Vamos criar o arquivo teste5.txt adicionar e commitar o mesmo.

$git log -p -1

commit b906227bcc9654d78764ba3c55021c165e71be6a

Author: neto.silva <alnetosilva@gmail.com>

Date: Fri Dec 18 00:30:34 2015 -0200

incluindo teste5.txt

diff --git a/teste5.txt b/teste5.txt

new file mode 100644

index 0000000..e69de29

Observe no log que ele foi commitado corretamente. No entanto, queremos remover esse commit e as alterações que foram feitas com ele.

$ git reset HEAD~1 --hard

HEAD is now at 524c533 commitando tudo novamente

$ls

teste1.txt teste2.txt teste3.txt teste4.txt

Assim, essa versão e tudo que havia nela foi removida permanentemente do nosso projeto.

$git log --pretty=oneline -1

524c53359cf1aa1c5c07333ce65c6f5bc76f71bd commitando tudo novamente

E nosso commit master voltou a ser o “commitando tudo novamente”

As vezes, fazemos tantas edições no código de um programa que acabamos esquecendo onde iniciamos e o que foi feito nesse código. Nesses casos, o git nos dá a possibilidade de voltarmos o arquivo ao estado original.

Vamos modificar algumas linhas no teste1.txt e depois voltar ao estado original.

Primeiro, vamos ver como está nosso arquivo1.txt.

$ cat teste1.txt

Modificando completamente teste1.txt

Agora vamos modificar:

$ vim teste1.txt

1 Modificando completamente teste1.txt

2 inserindo uma linha

<esc>:wq!

Agora que já modificamos o arquivo, vamos voltar ele a versão anterior a modificação. Passando o nome do arquivo como subparametro do checkout.

$ git checkout -- teste1.txt

$ cat teste1.txt

Modificando completamente teste1.txt

E então o nosso arquivo voltou para o estado em que estava no momento do ultimo commit. Tenha muito cuidado ao executar esse comando.

Entendendo branches (Ramificações / galhos).

O git, funciona com um software que faz fotocopias (snapshots) do estado atual do projeto. A cada commit, o git faz uma copia de tudo que está validado no momento atual e deixa guardado em sua estrutura para comparação e recuperação. Assim, o git segue uma linha do tempo para fazer o versionamento do projeto.

Uma vez que vamos fazer a divisão do nosso projeto em equipe onde cada desenvolvedor vá desenvolver um modulo de forma independente a partir de um ponto “x” do projeto já iniciado. Para que possamos dividir o projeto em varias partes a partir do ponto desejado e depois pegar todas as partes e tornar novamente um único projeto, vamos necessitar fazer branches do projeto a cada divisão e concatenação.

Ao criar um branche, temos a possibilidade de seguir linhas de desenvolvimento em paralelo onde cada programador vai desenvolvendo seu modulo de forma independente. Cada branch pode ter vários sub branches ou seja, um modulo pode ser desenvolvido em varias partes para ser desenvolvido por vários programadores. No entanto, ao fundir todos os módulos ou partes na versão final do projeto, cada commit irá ser alinhado conforme a linha do tempo no desenvolvimento.

Antes de iniciarmos a prática, vamos habilitar a visualização do branch atual no bash para poderemos trabalhar de modo mais claro. Edite seu ~.bashrc ou ~.bash\_profile na ultima linha conforme abaixo.

$vim + ~/.bashrc

### Git branch on terminal

function parse\_git\_branch () {

git branch 2> /dev/null | ( -e '/^[^\*]/d' -e 's/\* \(.\*\)/ (\1)/'

}

RED="\[\033[0;31m\]"

YELLOW="\[\033[0;33m\]"

GREEN="\[\033[01;32m\]"

NO\_COLOR="\[\033[0m\]"

PS1="$GREEN\u@\h$NO\_COLOR:\w$RED\$(parse\_git\_branch)$NO\_COLOR\$ "

force\_color\_prompt=yes

<esc>:wq!

Pronto! Agora que nosso bash já está configurado para mostrar o branch atual, vamos aos exemplos práticos.

Para criar um ramo do nosso projeto vamos usar o parâmetro -b no git checkout.

$ git checkout -b branch1

Switched to a new branch 'branch1'

Observe que ao criar o novo branch, o git já nos mudou para o branch criado e nossa modificação no bash que mostra o branch atual, agora mostra o “branch1” que acabamos de criar.

(branch1)$ git branch

\* branch1

master

A partir desse momento, todas as modificações feitas serão válidas somente no branch “branch1”. Vamos criar um novo arquivo no branch chamado branch1.txt.

$vim branch1.txt

1 novo branch

<esc>:wq!

Agora vamos adicionar e enviar o nosso arquivo ao controle de versão.

$ git add .

$ git commit -m "Incluindo branch1.txt"

[branch1 9feb57a] Incluindo branch1.txt

1 file changed, 1 insertion(+)

create mode 100644 branch1.txt

Uma vez que já temos uma modificação valida no “branch1” vamos verificar os últimos logs do nosso git.

$ git log --pretty=oneline -2

9feb57a19aa760858c3508da71fa1b7cbb29fa25 Incluindo branch1.txt

86ed89e733f3ad5f40c4eafa387b1d4640212564 comitando o diretorio ignorar

Observe que o nosso ultimo commit está sendo exibido conforme criado anteriormente.

Agora vamos voltar para o branch master.

$git checkout master

Switched to branch 'master'

(master)$

Observe que nosso console passa a exibir o branch master como branch atual.

Agora verifique os logs do git.

$ git log --pretty=oneline -2

86ed89e733f3ad5f40c4eafa387b1d4640212564 comitando o diretorio ignorar

d03b619467cc2ef9413604a70a16a0ff467d8cd6 editando inserindo a terceira quarta e quinta linha

Oh e agora onde foi parar nosso commit? Observe que nosso commit anterior não está sendo exibido. Isso acontece pois não estamos no branch onde as modificações foram feitas. Observe que o arquivo criado também não existe no branch master pois ele foi criado no “branch1”.

$ ls

teste1.txt teste2.txt

Agora, vamos modificar o teste1.txt no branch master, enviar e commitar automaticamente.

(master)$ vim teste1.txt

1 Modificando completamente teste1.txt

2 incluindo uma nova linha no teste1.txt

<esc>:wq!

(master)$ git status

No ramo master

Changes not staged for commit:

(utilize "git add <arquivo>..." para atualizar o que será submetido)

(utilize "git checkout -- <arquivo>..." para descartar mudanças no diretório de trabalho)

modified: teste1.txt

nenhuma modificação adicionada à submissão (utilize "git add" e/ou "git commit -a")

(master)$ git commit -a -m "editando o branch master"

[master cfa3e85] editando o branch master

1 file changed, 1 insertion(+)

Ao mudar de branch, veremos que não temos esta alteração no branch1 pois ela está somente no branch master assim como a alteração anterior que está somente no branch “branch1”. Isso nos dá a possibilidade de ir desenvolvendo um projeto em subdivisões, e ir entregando apenas as subdivisões que já estão validadas.

Vamos agora aprender a mesclar nossos branches.

Para mesclar os branches em um só, vamos usar o merge. Imagine que temos o branch1 como branch de desenvolvimento atual; no master, vamos inserir todas as modificações feitas em outros branches para assim, termos um projeto principal ou seja a junção dos nossos módulos subdivididos. Vamos para o branch master e a partir de lá vamos mesclar o conteudo dos demais branches.

(branch1)$ git checkout master

Switched to branch 'master'

Nesse ponto, vamos usar nosso primeiro comando de mesclagem de branches o “merge”.

O merge, como seu próprio nome já sugere, mescla as alterações presentes em cada branch para o branch atual (nesse caso o “master”). No entanto, para que isso seja possível ele gera um commit para descrever esse branch. Vamos a prática.

(master)$ git merge branch1

Seu editor padrão será aberto (no meu caso o vim) e então você deverá descrever o commit a ser usado para criar esse branch.

1 Merge branch 'branch1'

2 Criando nosso primeiro merge

<esc>:wq!

Merge made by the 'recursive' strategy.

branch1.txt | 1 +

1 file changed, 1 insertion(+)

create mode 100644 branch1.txt

(master)$ ls

branch1.txt teste1.txt teste2.txt

(master)$ git log –pretty=oneline -7

1351596581100be591de2e1503a70a2f5a41dda0 Merge branch 'branch1' Criando nosso primeiro merge

cfa3e853ece3736014ee67b8fcbb2ea0358cdca3 editando o branch master

9feb57a19aa760858c3508da71fa1b7cbb29fa25 Incluindo branch1.txt

86ed89e733f3ad5f40c4eafa387b1d4640212564 comitando o diretorio ignorar

d03b619467cc2ef9413604a70a16a0ff467d8cd6 editando inserindo a terceira quarta e quinta linha

9af0ea95bbc07e2a7d26b3d4ec484fd734655e58 inserindo a terceira quarta e quinta linhax

6b6df7b773fa8c2dd2a6b2ed22411d45663375b9 inserindo a terceira linha

Observe que ao listar nosso branch master e puxar os últimos 7 logs, todas as edições e inclusões que foram feitas no branch1 também estão válidas nele; e estamos com todos os commits presentes em ordem cronológica. Porém, se não quisermos gerar um commit ao mesclar os nossos branches, vamos usar o parametro “rebase”. Vamos resetar a HEAD do nosso master para -1 e apagar as modificações feitas pelo “merge” para então testarmos o “rebase”.

(master)$ git reset HEAD~1 --hard

HEAD is now at cfa3e85 editando o branch master

(master)$ git log --pretty=oneline -5

cfa3e853ece3736014ee67b8fcbb2ea0358cdca3 editando o branch master

86ed89e733f3ad5f40c4eafa387b1d4640212564 comitando o diretorio ignorar

d03b619467cc2ef9413604a70a16a0ff467d8cd6 editando inserindo a terceira quarta e quinta linha

9af0ea95bbc07e2a7d26b3d4ec484fd734655e58 inserindo a terceira quarta e quinta linhax

6b6df7b773fa8c2dd2a6b2ed22411d45663375b9 inserindo a terceira linha

O rebase reescreve a HEAD do nosso branch atual e organiza os commits que estão sendo incluídos nele de forma cronológica também. No entanto, ele não faz o uso de um commit para isso. Assim, o branch atual fica configurado como se todos os commits presentes desse ponto em diante fosse gerados a partir dele.

(master)$ git rebase branch1

First, rewinding head to replay your work on top of it...

Applying: editando o branch master

(master)$ git log --pretty=format:"%h - %cd : %s" -5

d3213b5 - Wed Dec 23 10:45:20 2015 -0200 : editando o branch master

9feb57a - Tue Dec 22 11:18:45 2015 -0200 : Incluindo branch1.txt

86ed89e - Thu Dec 17 23:10:25 2015 -0200 : comitando o diretorio ignorar

d03b619 - Thu Dec 17 01:25:51 2015 -0200 : editando inserindo a terceira quarta e quinta linha

9af0ea9 - Thu Dec 17 01:25:09 2015 -0200 : inserindo a terceira quarta e quinta linhax

Observe agora que ao usar o rebase, não necessitamos de um commit adicional e que a HEAD do nosso branch atual foi “rebobinada” e que nossos commits foram inseridos cronologicamente nela.

Agora, como nosso branch master já contem todas as alterações do branch1, podemos excluir o branch1 se desejarmos.

(master)$ git branch -D branch1

Deleted branch branch1 (was 9feb57a).

(master)$ git branch

\* master

Agora que já sabemos trabahar com o git, vamos iniciar nossas atividades com o github :).

O github é um serviço opensource que nos fornece repositórios git para na web para que assim possamos trabalhar com projetos de forma colaborativa.

Antes de seguir com a criação da nossa conta no github, vamos criar as nossas chaves ssh publica e particular. Assim, não vamos necessitar fazer login a cada transação entre nosso repositório local e nossos repositórios github.

$ ssh-keygen -C “seuemail@dominio.com”

Generating public/private rsa key pair.

Enter file in which to save the key (~/.ssh/id\_rsa):

Nesse ponto ao rodar o ssh-keygen o bash vai te perguntar onde deseja salvar as chaves geradas. Para manter o default como sugerido tecle <enter>.

Enter passphrase (empty for no passphrase):

Então o bash vai te pedir para digitar uma senha para a chave ssh. Deixe em branco e tecle <enter>.

Enter same passphrase again:

Na confirmação também tecle <enter>. E então o bash nos retornará a chave gerada.

Your identification has been saved in /home/alnetosilva/.ssh/id\_rsa.

Your public key has been saved in /home/alnetosilva/.ssh/id\_rsa.pub.

The key fingerprint is:

SHA256:Emw8jvZyjh4/djA/S3DJYMYfM66VOdFYJxjGwww0j94 alnetosilva@note-nt

The key's randomart image is:

+---[RSA 2048]----+

| .+=ooo . |

| + =\*+ o |

| % B.. |

| B O O |

| o = E |

| . .o\* . |

| o ++. |

| Bo.+ |

| .o.oo.o |

+----[SHA256]-----+

$ ls ~/.ssh/

id\_rsa id\_rsa.pub known\_hosts

Vamos nos certificar que nosso ssh-agent está habilitado. E adicionar nossa nova chave ssh ao ssh-agent.

$ eval "$(ssh-agent -s)"

Agent pid 4106

$ssh-add ~/.ssh/id\_rsa

Identity added: ~/.ssh/id\_rsa (~/.ssh/id\_rsa)

Ao listar nosso diretório de chaves ssh, veremos que foram criados dois novos arquivos. O “id\_rsa” e “id\_rsa.pub”. O “id\_rsa” é nossa chave privada, não compartilhe com ninguém. Para usar o github vamos uppar o “id\_rsa.pub” que é nossa chave publica.

No portal [https://github.com](https://github.com/) vamos criar uma conta github de preferência, fornecendo os mesmos os dados usados na criação do nosso repositório local.

Após criar sua conta vá até SSH-Keys em personal settings. De um nome para sua nova chave ssh copie o conteúdo da sua id\_rsa.pub. Você pode imprimir na tela o conteúdo com o comando.

$ cat ~/.ssh/id\_rsa.pub

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC3tdcGukjtA+9/mX13p5Xh7/DJkbKdOQVpadQduKYaS4jnokblcrHufyu6xnVtqw7RN9mbSbgbhkJVg0/B6Z0LEnQbTnvt+mBc6tPKiVA4H9qjetpKg0a/CbUulXNWMKTWfvPtnz17ds2+F

Teste se está tudo correto entre o serviço do github e sua chave ssh.

Copie o conteúdo e cole no campo “key” depois clique no botão Add key.

Pronto temos nossa chave publica no nosso github.

Agora vamos criar nosso primeiro repositório git.

Clique na aba repositories e depois em “create one”. De um nome ao seu repositório de preferencia o mesmo do repositório local, deixe selecionada a opção public e clique em create repository.

Não cheque a opção initialize this repository with a README pois vamos upar o conteúdo do nosso repositório local. A opção private só está disponível para contas pagas. Por isso, vamos criar um repositório publico mesmo. :)

depois de clicar em create repository vamos receber uma confirmação de que nosso repositório foi criado e o link para ele. No meu caso:

<https://github.com/alnetosilva/aulagit.git>

Agora vamos ao nosso repositório local para enviar nosso primeiro projeto para o git. Para isso, primeiro vamos ter que incluir nosso repositório github ao escopo do nosso repositório local. Para ver seu repositório ssh selecione ssh no campo select onde está https.

(master)$ git remote add origin git@github.com:alnetosilva/aulagit.git

Veja que foram adicionados parâmetros de configuração que incluem nosso repositório do github ao config do nosso repositório local.

(master)$ cat .git/config

[core]

repositoryformatversion = 0

filemode = true

bare = false

logallrefupdates = true

[remote "origin"]

url = git@github.com:alnetosilva/aulagit.git

fetch = +refs/heads/\*:refs/remotes/origin/\*

Uma convenção comum para incluirmos um novo repositório remoto e dar o nome “origin” a ele.

Então vamos enviar nosso projeto (repositório) aulagit para nosso repositório aula git no github. De dentro do branch master vamos mandar o comando:

(master)$ git push origin master

Counting objects: 41, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (33/33), done.

Writing objects: 100% (41/41), 3.72 KiB | 0 bytes/s, done.

Total 41 (delta 5), reused 0 (delta 0)

To [git@github.com](mailto:git@github.com):alnetosilva/aulagit.git

\* [new branch] master -> master

E então você vai conseguir fazer o upload do seu repositório local para o seu repositório remoto.

Caso queira fazer via https, vamos reconfigurar nosso origin.

$ git remote remove origin

$ git remote add origin <https://github.com/alnetosilva/aulagit.git>

Observe que enviamos o conteúdo do branch atual (no nosso caso o master) para o conteúdo do nosso no origin em nosso branch master online (no git hub). Você deve entrar no repositório e no branch correto para fazer o envio.

E então dar um reload no nosso navegador na url do repositório que acabamos de criar online e veja que seu projeto (branch) foi exportado para o github.

Agora no github, vamos criar um novo branch; então clique no campo select “branch:master” e digite o nome do novo branch como é nosso segundo exemplo de branch vamos nomear como branch2.

Ou vamos criar nosso branch2 localmente e enviar via push para nosso github.

(master)$ git checkout -b branch2

Switched to a new branch 'branch2'

(branch2)$ vim branch2.txt

1 criando um segundo branch para teste do github

<esc>:wq!

(branch2)$ git add .

(branch2)$ git commit -m "criando um segundo branch para teste do github"

[branch2 5e246d6] criando um segundo branch para teste do github

2 files changed, 2 insertions(+)

create mode 100644 branch2.txt

delete mode 100644 ignorar/ignorar.txt

E então vamos enviar nosso segundo branch para o github.

(branch2)$ git push origin branch2

Counting objects: 3, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 394 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To git@github.com:alnetosilva/aulagit.git

\* [new branch] branch2 -> branch2

Vá até o repositório do github e navegue a vontade para verificar os detalhes do nosso novo branch e observe nossos commits e demais peculiaridades.

Inserindo um README.md no nosso diretório github.

Agora que já entendemos como funciona o push, vamos colocar um README.md no nosso repositório github e fazer com que ele seja exibido toda vez que alguém acessar nosso repositório. Além disso, vamos utilizar o recurso markdown (.md ou .markdown) que vai nos ajudar a dar formato (estilo) ao nosso README.md.

Primeiramente, vamos criar o arquivo README.md.

(master)$ vim README.md

1 Primeiro README.md

(master)$ git add README.md

(master)$ git commit -m "incluindo README.md"

[master d1ce993] incluindo README.md

1 file changed, 1 insertion(+)

create mode 100644 README.md

(master)$ git push origin master

Agora que já enviamos nosso README.md observe que nosso repositório já o está exibindo na inicialização do nosso repositório.

Vamos estilizar um pouco nosso README para dar enfase em partes do texto que queiramos destacar.

(master)$ vim README.md

1 #Este é um texto em tag h1

2

3 ##Este é um texto em tag h2

4

5 ###### Este é um texto em tag h6

6

7 \*Este é um texto em itálico\*

8

9 \_Este também é um texto em itálico\_

10

11 \*\*Este é um texto em negrito\*\*

12

13 \_\_Este também é um texto em negrito\_\_

14

15 \*\_\_Este é um texto em negrito e itálico\_\_\*

(master)$ git add README.md

(master)$ git commit -m “Editando README.md”

(master)$ git push origin master

Observe que não devem haver espaços entre o texto estilizado e o carácter de estilo e que para fazermos parágrafos, devemos deixar uma linha em branco. Fique a vontade para brincar nesse ponto! Para mais detalhes:

<https://guides.github.com/features/mastering-markdown/>

Observe agora as modificações feitas no seu repositório github.

Enviando o segundo branch:

Para enviarmos o segundo branch é muito simples. Vamos mudar de branch no nosso projeto e então usar o push.

(master)$ git checkout branch2

Switched to branch 'branch2'

Agora que já mudamos de branch localmente, vamos enviar este branch para o github. Mudar para o branch local é uma boa prática antes de enviar, mas não é obrigatório. Se enviarmos o branch a partir de outro branch também funciona.

(branch2)$ git push origin branch2

Total 0 (delta 0), reused 0 (delta 0)

To git@github.com:alnetosilva/aulagit.git

b0792b9..9c6a385 branch2 -> branch2

Vá té seu repositório no github e veja as modificações enviadas.

Clonando um repositório:

Digamos que vamos iniciar a colaboração com um projeto existente, para isso, vamos trazer os arquivos do projeto para um repositório local e editar os códigos a partir dele. Para isso, vamos usar o comando clone.

Vamos apagar nosso repositório a partir de um diretório a cima aulagit local e depois disso vamos clonar o repositório github para o novo local aulagit.

Para isso, nogit hub copie a url https do seu projeto online, via ssh também pode ser feito. No entanto, em alguns firewalls a porta ssh pode estar bloqueada.

No caso do nosso exemplo, a url do projeto é:

<https://github.com/alnetosilva/aulagit.git> então vamos a clonagem.

~$ git clone https://github.com/alnetosilva/aulagit.git aulagit

Observe que passei como sub parâmetro o nome do novo repositório inicializado a partir do clone. Se deixar-lo em branco, nosso novo repositório irá receber o nome do repositório online sem o “.git”.

~$ cd aulagit/

(master)$ git branch

\* master

Observe que ao tentar listar os branchs locais, verificamos que temos apenas o master. No entanto, nosso origin já vem configurado do modo como foi enviado para o github.

(master)$ vim .git/config

1 [core]

2 repositoryformatversion = 0

3 filemode = true

4 bare = false

5 logallrefupdates = true

6 [remote "origin"]

7 url = git@github.com:alnetosilva/aulagit.git

8 fetch = +refs/heads/\*:refs/remotes/origin/\*

9 [branch "master"]

10 remote = origin

11 merge = refs/heads/master

Para listar os demais brachs (online) vamos usar o subparametro -a.

(master)$ git branch -a

\* master

remotes/origin/HEAD -> origin/master

remotes/origin/branch2

remotes/origin/master

Observe que temos um branch origin/branch2 no nosso github que não está presente no nosso repositorio local. Para que possamos trazer esse branch para nosso repositorio local, podemos usar o comando git checkout -b. No entanto, vamos passar o origin/branch2 como ponto de inicialização para ele.

(master)$ git checkout -b branch2 origin/branch2

Branch branch2 set up to track remote branch branch2 from origin.

Switched to a new branch 'branch2'

(branch2)$ ls

branch1.txt branch2.txt README.md teste1.txt teste2.txt teste3.txt

Então temos todos nossos brachs online no nosso repositório local.

Para checar se temos alguma diferença de commit entre nosso repositório online e local, vamos usar o comando git pull.

(branch2)$ git pull

Already up-to-date.

Como nossas modificações online estão iguais as locais o git irá nos informar que já está tudo na ultima forma.

Push e Pull com novo branch.

Imagine que estamos em vários colaboradores em nosso projeto, e desejamos a cada etapa do desenvolvimento sincronizar todos os diretórios locais de todos os programadores para que todos estejam no mesmo nível de desenvolvimento. Ou seja, todos estarão sempre com as ultimas atualizações feitas pela equipe.

Assim, vamos criar um novo diretório git para simular que um outro programador entrou no circuito de desenvolvimento de um novo projeto.

Primeiro crie um diretório para armazenar nossos diretórios git.

~$ mkdir equipe/

~$ cd equipe/

Agora vamos criar o tal diretório de desenvolvimento.

~/equipe$ mkdir projeto/

Agora iniciaremos controle de versão do git nesse diretório.

$ git init

Initialized empty Git repository in /home/alnetosilva/equipe/projeto/.git/

Crie um arquivo no diretório do projeto.

~/equipe/projeto$ vim arquivo1.txt

1 Primeira linha!

<esc>:wq!

Insira o arquivo no controle de versão do git

~/equipe/projeto$ git add arquivo1.txt

alnetosilva@alnetosilva-dev:~/equipe/projeto$ git commit -m "iniciando novo projeto git"

[master (root-commit) e838ad0] iniciando novo projeto git

1 file changed, 1 insertion(+)

create mode 100644 arquivo1.txt

Crie o diretório do projeto no github e depois adicione o diretório remoto ao diretório local.

(master)$ git remote add origin [git@github.com](mailto:git@github.com):alnetosilva/projeto.git

Envie o projeto iniciado no diretório local para o diretório github.

(master)$ git push origin master

Warning: Permanently added the RSA host key for IP address '192.30.252.131' to the list of known hosts.

Counting objects: 3, done.

Writing objects: 100% (3/3), 243 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To git@github.com:alnetosilva/projeto.git

\* [new branch] master -> master

Volte para o diretório equipe:

~/equipe/projeto (master)$ cd ../

Clone o repositório online para o repositório do “novo desenvolvedor” no nosso caso darei o nome de projeto\_

~/equipe$ git clone https://github.com/alnetosilva/projeto.git projeto\_

Cloning into 'projeto\_'...

remote: Counting objects: 3, done.

remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0

Unpacking objects: 100% (3/3), done.

Checking connectivity... done.

Entre no diretório do segundo desenvolvedor e observe que o config do git vei com padrão https: para o origin. Vamos mudar para ssh.

~/equipe$ cd projeto\_/

~/equipe/projeto\_ (master)$ cat .git/config

[core]

repositoryformatversion = 0

filemode = true

bare = false

logallrefupdates = true

[remote "origin"]

url = https://github.com/alnetosilva/projeto.git

fetch = +refs/heads/\*:refs/remotes/origin/\*

[branch "master"]

remote = origin

merge = refs/heads/master

~/equipe/projeto\_ (master)$ git remote remove origin

~/equipe/projeto\_ (master)$ git remote add origin git@github.com:alnetosilva/projeto.git

~/equipe/projeto\_ (master)$ cat .git/config

[core]

repositoryformatversion = 0

filemode = true

bare = false

logallrefupdates = true

[branch "master"]

[remote "origin"]

url = git@github.com:alnetosilva/projeto.git

fetch = +refs/heads/\*:refs/remotes/origin/\*

Agora no projeto do segundo desenvolvedor, edite o arquivo1.txt

~/equipe/projeto\_ (master)$ vim arquivo1.txt

1 primeira linha!

2 segunda linha

<esc>:wq!

~/equipe/projeto\_ (master)$ git commit -a -m "editando arquivo1.txt"

E envie para o github.

~/equipe/projeto\_ (master)$ git push origin master

Counting objects: 3, done.

Writing objects: 100% (3/3), 278 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To git@github.com:alnetosilva/projeto.git

e838ad0..aa202c4 master -> master

Volte para o repositório do primeiro desenvolvedor.

~/equipe/projeto\_ (master)$ cd ../projeto

E então sincronize os arquivos do github com o repositorio do primeiro desenvolvedor.

~/equipe/projeto (master)$ git pull origin master

remote: Counting objects: 3, done.

remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0

Unpacking objects: 100% (3/3), done.

From github.com:alnetosilva/projeto

\* branch master -> FETCH\_HEAD

e838ad0..aa202c4 master -> origin/master

Updating e838ad0..aa202c4

Fast-forward

arquivo1.txt | 1 +

1 file changed, 1 insertion(+)

Observe que o comando pull é passado igual fazemos com o push. No entanto, ele faz o inverso do comando push. Além disso, ele mostra quais os arquivos modificados após a sincronização.

Para avançarmos ainda mais nos nossos testes, vamos criar um novo branch no diretório do segundo desenvolvedor.

~/equipe/projeto (master)$ cd ../projeto\_/

~/equipe/projeto\_ (master)$ git checkout -b desenvolvedor2

Switched to a new branch 'desenvolvedor2'

~/equipe/projeto\_ (desenvolvedor2)$ vim arquivo2.txt

1 – arquivo2.txt

<esc>:wq!

~/equipe/projeto\_ (desenvolvedor2)$ git add arquivo2.txt

~/equipe/projeto\_ (desenvolvedor2)$ git commit -a -m "commitando arquivo 2"

[desenvolvedor2 1c225af] commitando arquivo 2

1 file changed, 1 insertion(+)

create mode 100644 arquivo2.txt

Vamos enviar o novo branch para o github:

~/equipe/projeto\_ (desenvolvedor2)$ git push origin desenvolvedor2

Counting objects: 3, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 290 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To git@github.com:alnetosilva/projeto.git

\* [new branch] desenvolvedor2 -> desenvolvedor2

Observe que ao tentar listar os branchs remotos a partir do diretório do projeto que pertence ao desenvolvedor1, o novo branch que acabamos de enviar não é listado.

~/equipe/projeto\_ (desenvolvedor2)$ cd ../projeto

~/equipe/projeto (master)$ git branch -a

\* master

remotes/origin/master

Isso acontece, pois não sincronizamos a HEAD do repositório local do desenvolvedor1 com a origin. Façamos a sincronização:)

~/equipe/projeto (master)$ git pull

remote: Counting objects: 3, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0

Unpacking objects: 100% (3/3), done.

From github.com:alnetosilva/projeto

\* [new branch] desenvolvedor2 -> origin/desenvolvedor2

There is no tracking information for the current branch.

Please specify which branch you want to merge with.

See git-pull(1) for details

git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so with:

git branch --set-upstream-to=origin/<branch> master

Pronto … :) Observe que a partir deste momento vamos conseguir listar os outros branchs

(master)$ git branch -a

\* master

remotes/origin/desenvolvedor2

remotes/origin/master

Agora vamos trazer o novo branch para o diretório do desenvolvedor1.

~/equipe/projeto (master)$ git checkout -b desenvolvedor2 origin/desenvolvedor2

Branch desenvolvedor2 set up to track remote branch desenvolvedor2 from origin.

Switched to a new branch 'desenvolvedor2'

Agora use o git log para verificar a sincronização :)

~/equipe/projeto (desenvolvedor2)$ git log

commit 1c225af05872bc91a23137b2830234b87fcfdf30

Author: Neto Silva <alnetosilva@gmail.com>

Date: Thu Jan 7 16:04:08 2016 -0200

commitando arquivo 2

commit aa202c4c6d2b8a5f63e4eff0d7ac740f0d6c19c8

Author: Neto Silva <alnetosilva@gmail.com>

Date: Thu Jan 7 15:43:17 2016 -0200

editando arquivo1.txt

commit e838ad0efa5108285039c90246dfca385352c12f

Author: Neto Silva <alnetosilva@gmail.com>

Date: Thu Jan 7 14:41:59 2016 -0200

iniciando novo projeto git

Pronto os diretórios de projeto de nossos dois programadores estão sincronizados. Vamos agora mesclar os branchs com um rebase e enviar para o github a partir do branch master.

~/equipe/projeto (desenvolvedor2)$ git checkout master

Switched to branch 'master'

~/equipe/projeto (master)$ git rebase desenvolvedor2

First, rewinding head to replay your work on top of it...

Fast-forwarded master to desenvolvedor2.

~/equipe/projeto (master)$ ls

arquivo1.txt arquivo2.txt

~/equipe/projeto (master)$ git branch -D desenvolvedor2

Deleted branch desenvolvedor2 (was 1c225af).

~/equipe/projeto (master)$ git push origin desenvolvedor2

error: src refspec desenvolvedor2 does not match any.

error: failed to push some refs to 'git@github.com:alnetosilva/projeto.git'

~/equipe/projeto (master)$ git push origin :desenvolvedor2

To git@github.com:alnetosilva/projeto.git

Obs: veja que ao tentar passar o nome do branch a ser deletado sem os : deu erro.

Então envie as ultimas alterações que o branch master sofreu após o rebase para o repositório github:

~/equipe/projeto (master)$ git push origin master

Counting objects: 3, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 290 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To git@github.com:alnetosilva/projeto.git

aa202c4..1c225af master -> master

Versionamento semântico:

Versionamento semântico, é uma forma organizada de gerar versões do software, desse modo, temos a seguinte estrutura X.X.X.

A primeira posição (X).x.x: é o major version. Ou seja, a versão principal do sistema onde a cada grade mudança no sistema onde podemos mudar o sistema totalmente.

Na segunda posição x.(X).x temos o minor version. Ou seja uma versão dentro da versão ou uma nova funcionalidade, só que ao trocar a minor version, não podemos grandes alterações na compatibilidade no software.

A terceira posição x.x.(X): é para correções de bugs e melhorias, sem que a compatibilidade do software seja quebrada. Não acrescenta funcionalidades inexistentes anteriormente.

Assim, no git podemos criar tags com nosso versionamento semântico para o software que irá indicar a versão (release) de desenvolvimento no software. As tags são apontadas para um determinado commit assim, podemos indicar que até aquele commit temos a versão 0.1.0. Veja no git log seu ultimo commit.

~/equipe/projeto (master)$ git log --pretty=oneline

1c225af05872bc91a23137b2830234b87fcfdf30 commitando arquivo 2

aa202c4c6d2b8a5f63e4eff0d7ac740f0d6c19c8 editando arquivo1.txt

e838ad0efa5108285039c90246dfca385352c12f iniciando novo projeto git

Veja que nosso ultimo commit foi “commitando arquivo 2”. Assim, ao inserirmos uma tag sabemos que essa versão trará para nós todas as alterações feitas até esse ponto. Assim, nesse ponto criaremos a tag (release) 0.1.0 a partir do nosso branch master. O comando para criar é o git tag e para listar basta passar o -l

~/equipe/projeto (master)$ git tag 0.1.0

~/equipe/projeto (master)$ git tag -l

0.1.0

Vamos mandar para o github apenas nossa tag:

~/equipe/projeto (master)$ git push origin master --tags

Total 0 (delta 0), reused 0 (delta 0)

To git@github.com:alnetosilva/projeto.git

\* [new tag] 0.1.0 -> 0.1.0

Agora no github observe que temos uma nova release. E caso queira pode baixar a versão do github pela aba releases.

Para deletar tags localmente, usamos o seguinte comando:

~/equipe/projeto (master)$ git tag -d 0.1.0

Deleted tag '0.1.0' (was 1c225af)

Para deletar tags remotas, vamos fazer da seguinte forma:

~/equipe/projeto (master)$ git push origin :refs/tags/0.1.0

To git@github.com:alnetosilva/projeto.git

- [deleted] 0.1.0

Assim deletamos a nossa tag no github.

Resolvendo conflitos:

Algumas vezes, vamos ter conflitos no git como: Mesclar alterações no mesmo arquivo e na mesma linha e etc. No entanto, não é um bicho de sete cabeças para solucionar-mos, vamos ver alguns exemplos.

Primeiramente vamos criar um branch a partir das ultimas alterações feitas no master.

~/equipe/projeto (master)$ git checkout -b branch1

Switched to a new branch 'branch1'

agora façamos alterações no arquivo1.txt

~/equipe/projeto (branch1)$ vim arquivo1.txt

~/equipe/projeto (branch1)$ cat arquivo1.txt

1 primeira linha!

2 segunda linha

3 terceira linha!

<esc>:wq!

~/equipe/projeto (branch1)$ git commit -a -m "acrescentando linha3 ao arquivo arquivo1.txt"

[branch1 04d3f80] acrescentando linha3 ao arquivo arquivo1.txt

1 file changed, 1 insertion(+)

Agora no branch master vamos editar o mesmo arquivo no branch master.

~/equipe/projeto (branch1)$ git checkout master

Switched to branch 'master'

~/equipe/projeto (master)$ vim arquivo1.txt

1 primeira linha!

2 segunda linha

3 terceira linha!

4 quarta linha

<esc>:wq!

Agora vamos mesclar o branch1 para o master:

~/equipe/projeto (master)$ git rebase branch1

First, rewinding head to replay your work on top of it...

Applying: incluindo linha3 e 4 ao arquivo1.txt

Using index info to reconstruct a base tree...

M arquivo1.txt

Falling back to patching base and 3-way merge...

Mesclagem automática de arquivo1.txt

CONFLITO (conteúdo): conflito de mesclagem em arquivo1.txt

Failed to merge in the changes.

Patch failed at 0001 incluindo linha3 e 4 ao arquivo1.txt

The copy of the patch that failed is found in:

/home/alnetosilva/equipe/projeto/.git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".

If you prefer to skip this patch, run "git rebase --skip" instead.

To check out the original branch and stop rebasing, run "git rebase --abort".

~/equipe/projeto ((no branch, rebasing master))$

Observe que ao usar o rebase vamos ter um conflito pois fizemos alterações no mesmo arquivo em estágios diferentes e o git não sabe sozinho o que fazer pois ele não sabe à quais alterações vai dar preferencias mesmo aparentemente tendo somente a linha 4 de diferença entre os dois commits. No entanto, o rebase nos levou para um outro “branch” “(no branch, rebasing master)”.

~/equipe/projeto ((no branch, rebasing master))$ vim arquivo1.txt

1 primeira linha!

2 segunda linha

3 terceira linha!

4 <<<<<<< 04d3f80e19842b6dcaa29790be3de7bf99ddb218

5 =======

6 quarta linha

7 >>>>>>> incluindo linha3 e 4 ao arquivo1.txt

Observe que ao abrir para edição o git indica que temos uma inserção na “terceira linha” e outro na “quarta linha” apague os dados da linha 4 5 e 7 (caso seja essa a versão final que deseje para o arquivo) ou edite o arquivo da forma que achar melhor. Após editado, observe que temos que usar o git add para adicionar o arquivo novamente ao controle de versão.

~/equipe/projeto ((no branch, rebasing master))$ git rebase --continue

arquivo1.txt: needs merge

You must edit all merge conflicts and then

mark them as resolved using git add

Vamos fazer da forma certa agora. Como sugerido pelo git:

~/equipe/projeto ((no branch, rebasing master))$ git add arquivo1.txt

~/equipe/projeto ((no branch, rebasing master))$ git rebase --continue

Applying: incluindo linha3 e 4 ao arquivo1.txt

~/equipe/projeto (master)$

E então nosso rebase estará completo.

Vamos fazer agora um merge. Para isso volte a head do projeto ao estagio anterior.

~/equipe/projeto (master)$ git reset HEAD~1 --hard

HEAD is now at 04d3f80 acrescentando linha3 ao arquivo arquivo1.txt

Para resolver conflitos com o merge vamos trabalhar de forma parecida.

(master)$ git merge branch1

Mesclagem automática de arquivo1.txt

CONFLITO (conteúdo): conflito de mesclagem em arquivo1.txt

Automatic merge failed; fix conflicts and then commit the result.

~/equipe/projeto (master)$ vim arquivo1.txt

1 primeira linha!

2 segunda linha

3 terceira linha!

4 <<<<<<< HEAD

5 quarta linha

6 =======

7 quarta linha!

8 >>>>>>> branch1

9 quinta linha

<esc>:wq!

Assim, edite o arquivo com conflito e de um git add e depois git commit.

~/equipe/projeto (master)$ git add arquivo1.txt

~/equipe/projeto (master)$ git commit -m "concluindo o merge"

[master 4954964] concluindo o merge

Repositório central (bare):

Imagine a seguinte situação, você está em uma equipe de desenvolvedores em uma empresa, onde para cada região (filial) a empresa tem desenvolvedores para funcionalidades especificas para aquele local. No entanto, as validações locais precisam entrar no código fonte do projeto master. Por se tratar de um projeto fechado, vamos precisar de um local reservado para nosso repositório. Onde porém, todos os desenvolvedores do projeto tenham acesso. Assim, você criaria um repositório bare no dominio de sua empresa.

Vamos criar nosso primeiro repositório bare.

No diretório que criamos para simular o ambiente de desenvolvimento em equipe, crie o diretório que vai servir como repositório bare.

~/equipe/projeto (master)$ cd ../

~/equipe$ mkdir projetobare.git

Entre no repositório e inicialize o controle de versões só que com o parâmetro –bare. Isso irá passar para o git a informação de que esse repositório é apenas um diretório servidor.

~/equipe/projetobare.git$ git init --bare

Initialized empty Git repository in /home/alnetosilva/equipe/projetobare.git/

~/equipe/projetobare.git$ ls

branches config description HEAD hooks info objects refs

Observe que ao criar um repositório bare o git não cria o diretório .git e monta a estrutura de controle no próprio diretório.

Agora para usarmos nosso repositório bare vamos adicionar ele ao repositório do nosso projeto igual fizemos com nosso repositório do github.

~/equipe/projetobare.git$ cd ../projeto

(master)$ git remote add empresa ssh://127.0.0.1/home/alnetosilva/equipe/projetobare.git

E então vamos mandar nossas modificações do projeto para o repositório git particular da empresa. O git irá pedir confirmação de certificado ssh e a senha do usuário para iniciar a conexão ssh.

~/equipe/projeto (master)$ git push empresa master

The authenticity of host '127.0.0.1 (127.0.0.1)' can't be established.

ECDSA key fingerprint is SHA256:nXYbYXVH3AIcpV64YxQ25SpT2X0lIKiyhq3zhshfdb8.

Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added '127.0.0.1' (ECDSA) to the list of known hosts.

alnetosilva@127.0.0.1's password:

Counting objects: 25, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (19/19), done.

Writing objects: 100% (25/25), 2.35 KiB | 0 bytes/s, done.

Total 25 (delta 4), reused 0 (delta 0)

To ssh://127.0.0.1/home/alnetosilva/equipe/projetobare.git

\* [new branch] master -> master

Assim, as demais transações podem ser feitas igualmente fazemos no github. E para fazer clonagem do projeto basta inserir o link do repositório corretamente para o clone.

~/equipe$ git clone ssh://127.0.0.1/home/alnetosilva/equipe/projetobare.git projetoclone

Cloning into 'projetoclone'...

alnetosilva@127.0.0.1's password:

remote: Counting objects: 25, done.

remote: Compressing objects: 100% (19/19), done.

remote: Total 25 (delta 4), reused 0 (delta 0)

Receiving objects: 100% (25/25), done.

Resolving deltas: 100% (4/4), done.

Checking connectivity... done.

alnetosilva@alnetosilva-dev:~/equipe$ ls

projeto projeto\_ projetobare.git projetoclone

Lembre-se de alterar o 127.0.0.1 pelo ip e o caminho pelo caminho correto do seu repositório bare.

~/equipe$ cd projetoclone/

~/equipe/projetoclone (master)$ ls

arquivo1.txt arquivo2.txt

~/equipe/projetoclone (master)$ git log --pretty=oneline -5

4954964e5e0f925ed09a563faff4a3d70ca64e4c concluindo o merge

2847f21b4f538a3ed1b6c1fad456d7652a41f233 inserindo quinta linha

1804ee2605771c4743ddf27575b638cc212d6b6d inserindo quinta linha

a56e7b66fa141c80c42765da582ceb0cebe774ef Merge branch 'branch1'

5acf8b1bf3f296f7d331f2593c4281418e15039c inserindo quarta linha

Fazendo deplay atomatizado com o git.

Com o github, todas as vezes que dermos um push para sincronizar nosso diretorio local do projeto com o nosso diretório central (no nosso caso o projetobare.git que configuramos anteriormente) o git irá atualizar automaticamente um diretório de produção do nosso projeto como o caso de um site por exemplo.

Entre no repositorio bare.

~$ cd equipe/projetobare.git/

Nele, teremos um diretório chamado hooks. Que é onde o git cria links ou gachos que são executados quando acontecem determinados eventos.

~/equipe/projetobare.git (master)$ ls hooks/

applypatch-msg.sample commit-msg.sample post-update.sample pre-applypatch.sample pre-commit.sample prepare-commit-msg.sample pre-push.sample pre-rebase.sample update.sample

Observe que já temos vários hooks pré-configurados pelo git, vamos criar um novo gancho chamado post-receive que será executado como o nome sugere após o recebimento de alterações no projeto central e nele vamos inserir um codigo que será iniciado via shell “sh” no nosso servidor. Com o comando que vai fazer com que todas as vezes que recebermos uma alteração em nosso repositório central via push, essa alteração será enviada também para um diretorio do nosso site no nosso caso o “~/equipe/www$” que será criado na sequência. Em um projeto em produção esse diretório poderia ser a unidade do seu var/www/html.

Vamos criar nosso diretorio www.

~/equipe/projetobare.git (master)$ mkdir ~/equipe/www

Agora que já criamos, vamos configurar nosso post-receive.

~/equipe/projetobare.git (master)$ vim hooks/post-receive

1 #!/bin/sh

2 GIT\_WORK\_TREE=/home/alnetosilva/equipe/www git checkout -f

<esc>wq!

De permissão de execução para o aquivo criado para que ele possa ser um arquivo executavel.

~/equipe/projetobare.git (master)$ chmod +x hooks/post-receive

Nesse arquivo, o que estamos fazendo é criar uma variavel que irá armazenar o caminho do nosso diretório de produção e enviando para ela os arquivos alterados sem mandar o repositorio do git o “.git” através da opção “-f”.

Agora toda vez que enviarmos um novo push para o nosso repositorio projetobare.git, nosso script será executado e enviará as mudanças também para o nosso diretório www.

~/equipe/projetobare.git (master)$ cd ../projeto

~/equipe/projeto (master)$ vim arquivo1.txt

1 primeira linha!

2 segunda linha

3 terceira linha!

4 quarta linha!

5 quinta linha!

~/equipe/projeto (master)$ git commit -a -m "editando arquivo1.txt"

[master 7d2b135] editando arquivo1.txt

1 file changed, 1 insertion(+), 1 deletion(-)

~/equipe/projeto (master)$ git push empresa master

alnetosilva@127.0.0.1's password:

Counting objects: 3, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 323 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To ssh://127.0.0.1/home/alnetosilva/equipe/projetobare.git

7d2b135..4e44813 master -> master

~/equipe/projeto (master)$ ls ~/equipe/www/

arquivo1.txt arquivo2.txt

Pronto, nossas alterações já foram enviadas e nosso site já está atualizado. :)

No github, temos algumas funcionalidades importantes para colaborar em projetos de desconhecidos ou de conhecidos.

Uma dessas funcionalidades, é a fork. Com a fork você traz uma versão do projeto para a sua conta no github e enviar suas próprias alterações para sua fork sem interferir no projeto principal. Assim, ao encontrar um projeto com o qual deseje contribuir, basta clicar no icone fork no repositório github do projeto, observe que nesse ponto, este repositório vai ser copiado para sua conta, a partir deste momento, você poderá inciar as suas contribuições para o projeto.

Depois de fazer suas alterações no projeto fork, você deve enviar um pull request para que assim, o desenvolvedor do projeto principal veja suas alterações e aceite seu pull request, nesse ponto, o github irá fazer um merge das suas alterações no projeto principal.

Caso deseje receber notificações a cada modificações feitas em um projeto de seu interesse, no repositório github do projeto você vai clicar no botão <Watch> ou você pode clicar no botão <unstar> para tornar publico que é um projeto que você gostou.