Quando iniciamos um novo projeto, uma das primeiras perguntas que surge é: onde os arquivos desse projeto ficarão armazenados? Principalmente quando trabalhamos em equipe com outros desenvolvedores, faz-se necessário que o projeto esteja armazenado em um local que seja acessível a todos os membros da equipe, que permita ainda a atribuição de permissões para que os membros dessa equipe possam alterar os arquivos existentes, ou ainda permitir que novos arquivos sejam criados e que arquivos existentes possam ser excluídos.

Pensando nesse problema foram criadas ferramentas de **controle de versão** e esse curso apresentará uma das mais utilizadas hoje em dia, o **Git**. O Git é utilizado em muitos projetos de código aberto no mundo todo.

Uma grande vantagem do Git é a existência de ferramentas comerciais de hospedagem de código na Web. A maior provedora de hospedagem de repositórios Git, abertos ou privados, hoje é o [Github] (<http://www.github.com/>). Ele permite criar gratuitamente repositórios abertos.

Para que seja possível utilizar o Git teremos que instalá-lo. O Git é uma ferramenta baseada em linha de comando, ou seja, realizamos as operações de controle dos arquivos pelo prompt de comando. Se você é adepto de ferramentas gráficas não se preocupe, existem algumas opções para facilitar seu trabalho.

**1 - Download e instalação do Git**

A instalação do Git é diferente para cada sistema operacional.

<https://gitforwindows.org/>

## Windows

Para instalar o Git, inicialmente acessaremos a página acima e baixaremos o instalador do msysgit. Uma vez baixado, basta executar o programa e seguir as instruções (caso haja uma tela de instalação que não esteja aqui, pode aceitar com as opções padrão).

## 2 - Criando conta no Github

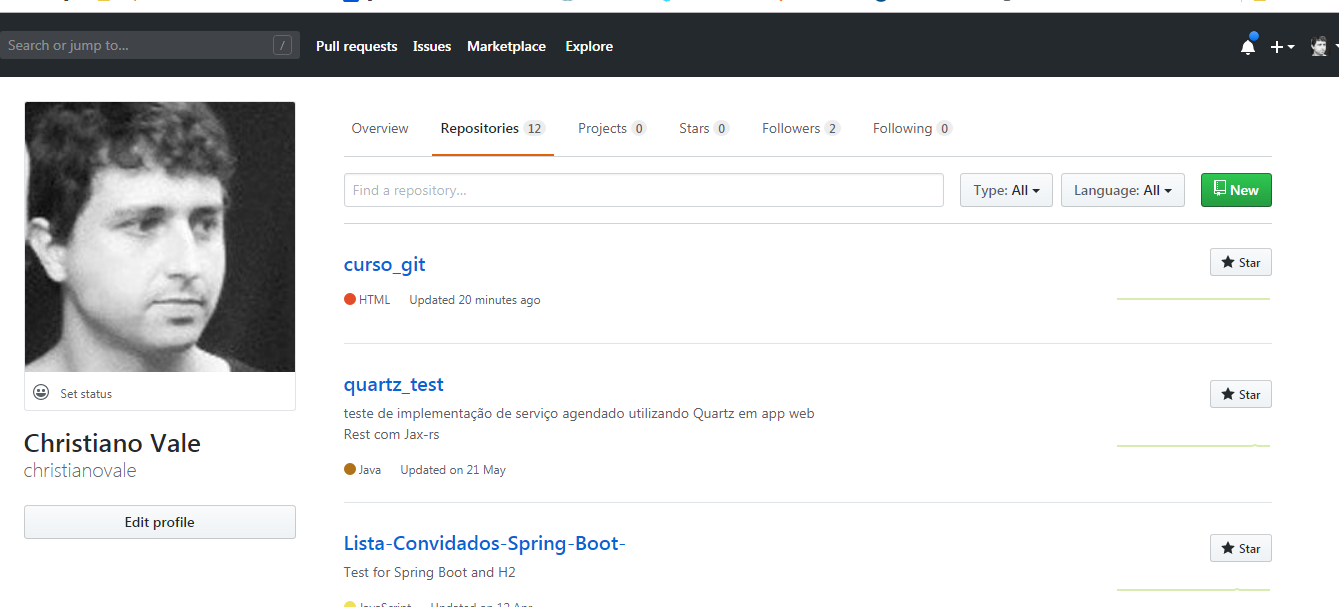
Agora que temos o Git instalado, podemos utilizar os serviços do Github. Primeiro precisamos criar uma conta. Para isso acessamos a página <https://github.com/plans>, escolhemos o tipo de conta que queremos criar e prosseguimos preenchendo os formulários pedidos.

# 2.1 - Criação de repositórios no Github para guardar os arquivos

Sempre que desejarmos armazenar os arquivos de um novo projeto, precisamos criar um novo repositório. O Github nos permite a criação de ilimitados repositórios, podendo ser repositórios públicos ou privados (antigamente só poderíamos ter acesso a repositórios privados a partir de planos pagos).

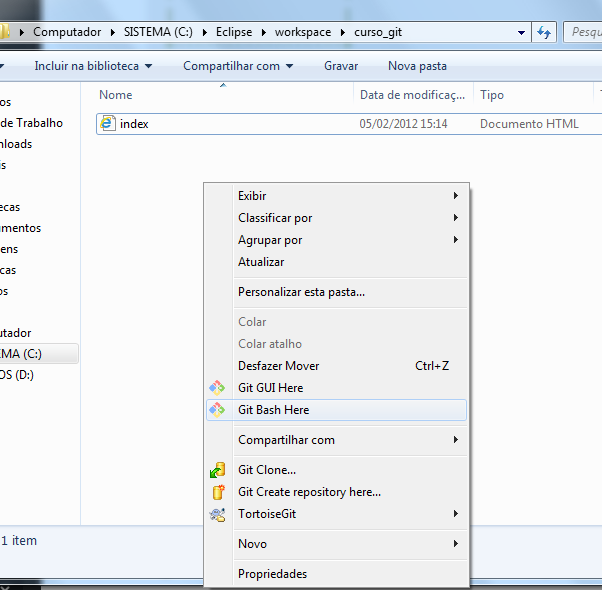
Para conseguirmos criar nosso repositório, é preciso estar logado no Github, com uma conta já previamente criada e configurada.

Uma vez logado no Github, existe na [página inicial](https://github.com/), do lado direito, um botão chamado "New Repository", que permitirá a criação do novo repositório. Em seguida, basta indicarmos o nome do projeto, que no nosso caso é "curso\_git", e, opcionalmente, uma descrição. Nesse instante, você deverá estar em uma tela similar à abaixo.



**2.2 - Criar um repositório local**

Depois de criada nossa conta no GitHub podemos criar um repositório local e conecta-lo a nossa conta remota. Para isso, na pasta onde se deseja criar o repositório clicamos com o botão direito e selecionamos “*git bash here”*



Caso o repositório ainda não tenha sido iniciado, podemos fazer nosso primeiro envio de código usando a sequencia de comandos:

**git init**

*Diz para o git que ele deve controlar essa pasta como repositorio*

**git add index.html**

*Adiciona o arquivo para ser monitorado pelo Git (adiciona arquivos novos e alterados)*

**git status**

*Mostra o status do seu repositório*

**git commit -m "first commit"**

*Realiza o envio para a Staging Area*

**git remote add origin** [**https://github.com/christianovale/curso\_git.git**](https://github.com/christianovale/curso_git.git)

*Cria uma ligação entre sua pasta local e o repositório remoto.*

*A sintaxe completa do comando é:*

*git remote add [alias\_do\_repositorio] [uri\_do\_repositorio]*

*Uma convenção adotada é a utilização do nome do repositório remoto como "origin". No entanto, qualquer outro nome pode ser* utilizado.

**git push -u origin master**

*Faz o envio para o repositório remoto.*

*O repositório remoto será o "origin", que acabamos de configurar, enquanto a branch será a "master", criada por padrão sempre que um repositório é criado.*

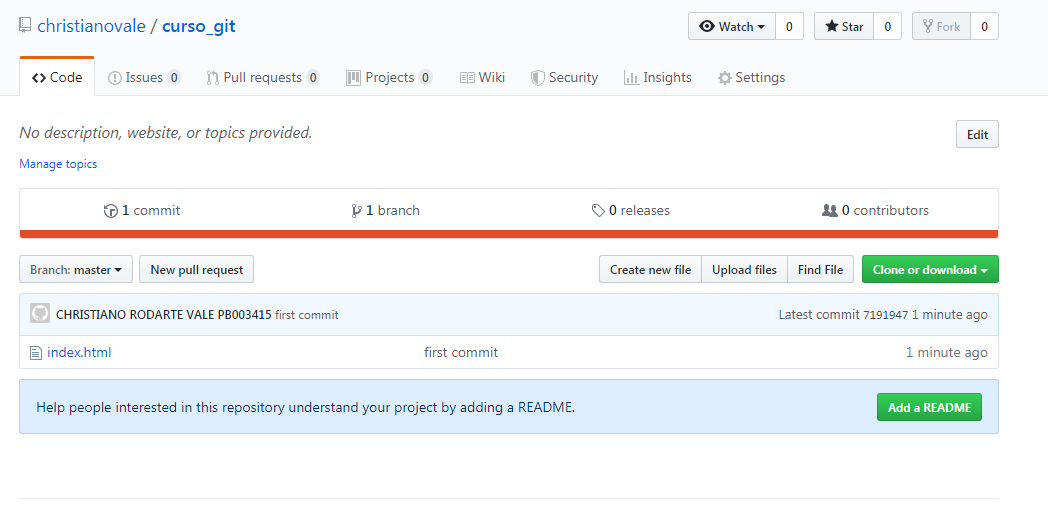
*Tecnicamente, o sinalizador –u adiciona uma referência de rastreamento para o servidor upstream para o qual você está empurrando.*

*O que é importante aqui é que esta permite que você faça um git pull sem fornecer quaisquer mais argumentos. Por exemplo, quando você faz um git push -u origin master, mais tarde você pode chamar git pull e git vai saber que você realmente quis fazer  git pull origin master.*

*Caso contrário, você teria que digitar o comando inteiro.*

**git log**

*mostra a lista de alterações no repositório*



# 3 - Recapitulação do processo

Durante o ciclo básico que demonstramos anteriormente, nós interagimos com 3 estágios diferentes do repositório. O primeiro deles enquanto nós criamos o repositório, mas não indicamos nenhum arquivo para ser rastreado.

Nesse estágio, estamos interagindo com um estado do projeto que chamamos de **"Working Directory"**, ou seja, é o nosso sistema de arquivos atual. Nele estão as alterações que estamos realizando no momento.

O Working Directory pode estar "limpo", quando não há diferença entre os arquivos armazenados no repositório e como estão atualmente. Quando há diferença (por exemplo, alteramos determinado arquivo, mesmo que uma alteração mínima), o Working Directory fica marcado como "dirt" (sujo).

Em nosso caso o arquivo não existia para se ter uma comparação, pois tínhamos um repositório novo. Após modificarmos nossos arquivos a ponto de definirmos que um "passo" foi concluído, criamos uma visão desse passo, um ponto de controle preliminar com o comando "git add".

Esse comando cria um novo estágio do repositório, o que chamamos de **"Index"** ou **"Staging Area"**. Esse estágio é transitório e pode ser alterado ainda antes de se tornar um passo do projeto: podemos adicionar novos arquivos, removê-los ou mesmo alterá-los.

Quando satisfeitos com o conteúdo do "Index", utilizamos o comando "git commit" para persistí-lo, gravar esse passo com todos os arquivos novos e alterações efetuadas. O comando "commit" criou um terceiro estágio do repositório que conhecemos como **"HEAD"**. O HEAD é o último estado que o Git usa como referência, é a visão do último passo do projeto que foi concluído e entregue.



# 3 - Copiando o projeto para sua máquina: o comando clone

Agora que o repositório do projeto está disponível e inclusive já possui commits realizados, outros desenvolvedores podem querer contribuir com o projeto. O primeiro passo para que um outro usuário realize suas alterações é ter em seu próprio computador uma cópia do repositório. Dessa forma, ele terá todos os arquivos disponíveis em seu computador, para alterar como desejar.

Para podermos fazer isso, nós temos que copiá-lo para nossa máquina. O repositório que vamos copiar está hospedado no Github. Para garantir que vamos adquirir todos os arquivos necessários, utilizaremos o comando git clone que, além de copiar os arquivos do projeto, traz todos os arquivos necessários para que o controle de versão dos arquivos continue sendo feito.

Abra o prompt de comando do computador e navegue até uma pasta que será a pasta "pai" do seu repositório local, por exemplo:

cd eclipse/workspace

Lembre-se de que você pode escolher o local mais conveniente para você em seu computador. Agora vamos executar o comando que copiará o repositório para essa pasta:

git clone https://github.com/christianovale/curso\_git.git

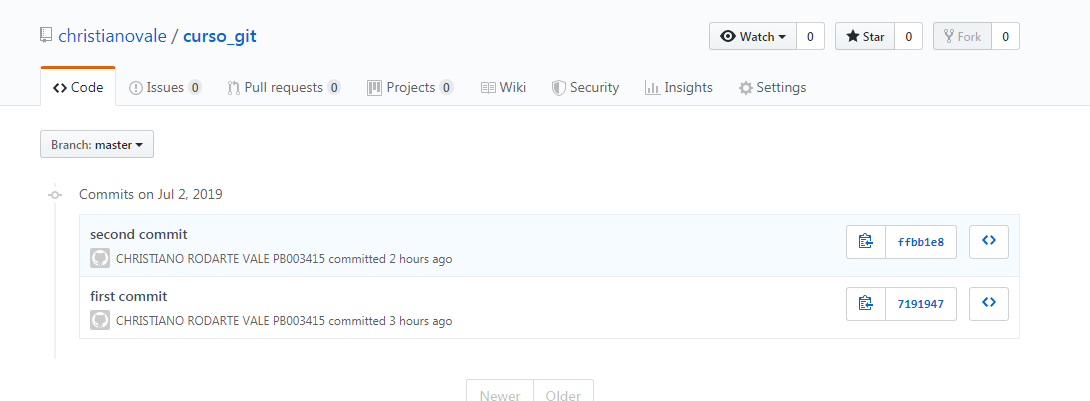
Esse comando **clone** criará uma pasta com o mesmo nome padrão do repositório (que no caso é "curso\_git") e copiará para esse diretório todos os arquivos que estavam disponíveis nele.

Ao clonar um repositório, não precisamos adicionar o repositório remoto através do comando git remote add, pois tudo já é feito pelo comando git clone, dando um alias "origin" para o repositório.

Nosso projeto de exemplo contém páginas HTML. Para visualizar o projeto basta abrir o arquivo **index.html** em qualquer navegador (Chrome, Firefox, Internet Explorer, entre outros).

Uma vez que o colaborador realizou o clone do repositório em seu computador e agora possui todos os arquivos localmente, as alterações desejadas por ele serão feitas localmente, igual aprendido na seção anterior. Ao final das alterações, ele realizará o envio dos commits para o repositório remoto através da execução do comando git push origin máster.

A partir desse momento, acessando a página do repositório no Github, é possível ver a lista com os commits realizados.



# 3.1 - Sincronização com as novas alterações do repositório

A partir do momento em que fizemos atualizações no repositório, os outros desenvolvedores que já o possuem em seus computadores não estarão sincronizados com estas alterações, ou seja, eles ainda não possuem em seus computadores as novas versões dos arquivos.

Para que a sincronização seja realizada e o desenvolvedor tenha em seu computador as novas versões dos arquivos, basta que ele execute o comando git pull origin master. A saída no prompt será similar à abaixo, indicando os arquivos que tiveram alterações e quantas linhas do arquivo foram afetadas.

remote: Counting objects: 5, done.

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

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

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

From github.com:[usuario\_no\_github]/curso-git

c886b6a..b9aa50c master -> origin/master

Updating c886b6a..b9aa50c

Fast-forward

index.html | 1 +

1 files changed, 1 insertions(+), 0 deletions(-)

**4 – Organização de trabalho com Branches**

Já sabemos que a utilização de branches facilita no dia a dia do desenvolvedor. Mas como criar uma nova branch? Para tal, utilizamos o comando git branch, passando como opção o nome da branch que desejamos criar. No nosso caso, criaremos a branch design, onde realizaremos algumas alterações referentes ao design da nossa página html:

git branch design

Ao executarmos o comando, nenhuma saída é mostrada no prompt.

Agora, como verificamos quais são as branches existentes em um projeto? Isso se resolve com o comando git branch. Ele nos fornece todas as branches criadas na máquina. Ele também possibilita visualizar qual a branch que estamos atualmente através de um "\*" que precede o nome da branch atual.

git branch

design

\* master

Observe que o "\*" precede uma branch chamada master. Mas nós não a criamos agora. De onde ela surgiu? A branch master é criada quando executamos o nosso primeiro commit do projeto. Ela é considerada a branch principal do projeto.

Mas, se quisermos alterar o projeto numa outra branch, como é que fazemos para alterar a branch atual? Isto é feito através do comando git checkout, passando o nome da branch para a qual desejamos mudar. No nosso caso, temos:

git checkout design

Switched to branch 'design'

  Também pode-se usar o comando: git checkout -b design.

E pronto. Todas as alterações que realizaremos a partir de agora estarão na branch design.

Vamos adicionar estilo para a nossa página. Para tal, copie o seguinte código num arquivo chamado design.css. Adicione este arquivo na pasta do seu projeto.

body {

background-color: blue;

}

Adicione também a seguinte linha no header do seu arquivo index.html:

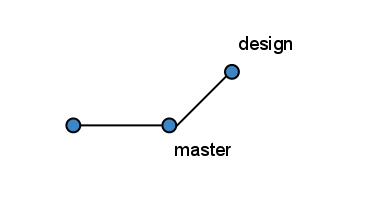
<link rel="stylesheet" type="text/css" href="/design.css"/>

Porém, essas alterações ainda não foram atualizadas no repositório. Para isso, precisamos adicionar os arquivos e commitar as alterações:

git add design.css index.html

git commit -m "Adicionando estilo para a nossa página"

E pronto! As alterações estão salvas na branch design. O esquema do projeto com as branches pode ser vista na figura a seguir:

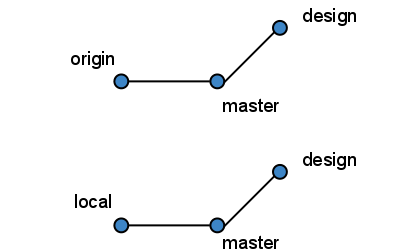


Se voltarmos para a branch master com o comando git checkout master, vemos que as alterações feitas anteriormente não estão mais presentes.

Já se pode enviar a branch criada localmente para o repositório remoto. Isso é feito utilizando o comando git push passando dois argumentos: o primeiro é o nome do repositório e o segundo, o nome da branch que deseja-se enviar. No nosso caso, temos:

git push origin design

Com isso, o repositório remoto conterá uma cópia fiel da branch design local. Isso pode ser visto na figura a seguir:

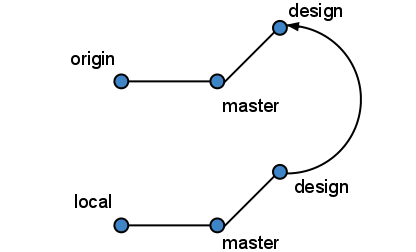


Porém, toda vez que atualizarmos tanto o nosso projeto local quanto o projeto remoto, precisaremos indicar qual o repositório e o nome da branch que a nossa branch local se refere no remoto, isto é, precisaremos digitar git pull origin design e git push origin design para atualizar os repositórios locais e remotos, respectivamente.

Para evitar tal trabalho, podemos indicar o caminho (track) da branch remota para a nossa branch local. Isso pode ser feito no instante em que criamos a branch remota através da opção "-u". No nosso caso, temos:

git push -u origin design

Com isso, a nossa branch local sabe qual a branch remota que ela se referencia, ou seja, *-u* diz pro git que a branch design remota é igual a branch local.



Para juntar o trabalho desenvolvido na Branch para master após a conclusão, usa-se o comando git merge. Este comando puxa as alterações da branch *design* para a branch a qual está ativa no momento.

git checkout master

git merge design

git push

*para atualizar seu repositório local com a mais nova versão, execute*

git pull

### Anexo I – Guia exemplo dos comandos mais utilizados

Entre no diretório que deseja controlar a versão e inicie o Git assim:

1. git init

Feito isso, seus arquivos ainda não estão sendo versionados, mas eles estão esperando para serem adicionados no estágio de controle. Para fazer isso digite o comando

1. git add *nome-do-arquivo-incluindo-extensão*

Se você precisa adicionar todos os arquivos do diretório, basta digitar:

1. git add .

Saber o status do projeto é importante. Com o comando abaixo você consegue ver quais arquivos estão fora do controle, quais foram modificados e estão esperando por uma descrição de modificação etc:

1. git status

Voltando ao estágio anterior do adicionamento:

1. git reset HEAD nome-do-arquivo

Commit – Comitando:

1. git commit -m "Mensagem do commit"

Adicionando e comitando ao mesmo tempo:

1. git commit -a -m "Mensagem do commit"

### Voltando commits a versões anteriores

Voltar um commit:

1. git reset HEAD~1

Voltar dois commits:

1. git reset HEAD~2

Voltando um commit e deixando o arquivo no estagio anterior:

1. git reset HEAD~1 --soft

Voltando um commit e excluindo o arquivo, deixando no estágio anterior:

1. git reset HEAD~1 --hard

Verificando o histórico de commits:

1. git log

Verificando o que foi mudado, diferença entre um arquivo e outro:

1. git log -p

Verificando os 2 últimos commits:

1. git log -p -2

Mostrando as estatísticas de todos os commits:

1. git log --stat

Mostrando todos os commits, cada um em uma linha:

1. git log --pretty=oneline

Mostrando todos os commits dos últimos 2 dias até o momento atual

1. git log --since=2.days

Criando um branch – uma ramificação

1. git checkout -b *nome-do-branch*

Verificando em que branch você está

1. git branch

Voltando para o branch master

1. git checkout master

### Jogando o branch criado no branch master

Entre como branch master:

1. git merge nome-do-branch-que-foi-criado

### Grudando o branch criado no branch master sem o commit

Somente localmente – localhost, entre como branch master:

1. git rebase nome-do-branch-que-foi-criado

Removendo um branch:

1. git branch -D *nome-do-branch*

Vendo branchs remotos:

1. git branch -a

Mostrando o início do hash, quem comitou, quanto tempo atrás, mensagem: descrição do commit:

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

Deletando arquivos:

1. git rm *nome-do-arquivo*

Deletando todos os aquivos removidos ao mesmo tempo:

1. git ls-files --deleted | xargs git rm

### Ignorando arquivos

Existem alguns arquivos que muito provavelmente você não vai precisar versionar, como por exemplo os arquivos de cache do SASS, arquivos de configuração e etc. Nesse caso você precisa fazer com que o controle de versão ignore estes arquivos. Para tanto, crie um arquivo chamado **.gitignore**. Feito isso, dentro deste arquivo, digite o nome ou o endereço das pastas que você quer ignorar. Um exemplo:

1. # See http://help.github.com/ignore-files/ for more about ignoring files.
2. #
3. # If you find yourself ignoring temporary files generated by your text editor
4. # or operating system, you probably want to add a global ignore instead:
5. # git config --global core.excludesfile ~/.gitignore\_global
7. # Ignore bundler config
8. /.bundle
10. # Ignore the build directory
11. /build
13. # Ignore Sass' cache
14. /.sass-cache
16. # Ignore .DS\_store file
17. .DS\_Store
18. .cache
19. .rvmrc
21. vendor/\*
23. .DS\_Store
25. # Vim
26. \*.swp
27. \*.swo
29. Gemfile.lock
30. .vagrant
31. Vagrantfile
33. # rbenv
34. .ruby-version
36. # Ignore deploy related files
37. deploy
39. Gemfile.lock

O arquivo **.gitignore** fica na raiz do projeto.

### Clonando e puxando alterações de projetos

Clonando um projeto remoto:

1. git clone url-do-projeto

Fazendo um clone de outros branchs:

1. git checkout -b *nome-do-branch* origin/ *nome-do-branch*

Trazendo, puxando as alterações feitas por outros usuários:

1. git pull origin master

Sincronizando tudo que está no repositório remoto:

1. git pull

Enviando o(s) projeto(s), arquivo(s) para o repositório:

1. git push origin master

Enviando um branch para o repositório:

1. git push origin *nome-do-branch*

### Tags

As tags servem para marcar uma etapa. Imagine que você vai lançar uma versão, que resolve uma série de problemas. Você pode marcar aquela etapa criando uma tag. Assim fica simples de fazer qualquer rollback do projeto para uma tag específica em vez de voltar para um commit. Você sabe que tudo o que foi feito até aquela tag está funcionando.

Criando tags:

1. git tag versão-da-tag

Listando tags:

1. git tag -l

Enviando a tag para o repositório

1. git push origin –-tags
2. git push origin versão-da-tag

Removendo as tags criadas localmente:

1. git tag -d versão-da-tag

Removendo tag no repositório remoto:

1. git push origin :refs/tags/versão-da-tag