Git & Github

Pablo Hugen — PabloASHugen@protonmail.com
Daniel Boll — danielboll.academico@gmail.com

September 2022

Conteúdo

Git

Introdução

> "Developers have the attention spans of slightly moronic woodland creatures." – (Linus Torvalds).

Versionamento

Sistema de controle de versão(*a.k.a. VCS*) é um sistema que guarda as alterações em **qualquer tipo** de arquivo a medida que modificações são sendo feitas neles. Ok, mas e o *Google Drive*? *minha pasta remota* favorita? Meu WinRar com pago zippando meus arquivos?

Um *VCS* possui inúmeras vantagens em relação a métodos "ingênuos" de versionar arquivos. Podemos citar dentre elas:

- **Tamanho dos arquivos**: Um **VCS** guarda somente diferenças entre versões de um arquivo. Já outros metodos duplicam as partes iguais de um arquivo modificado.
- **Comparação de diferencas**: Comparar modificações em arquivos copiados pode ser um trabalho tedioso, ainda mais se o arquivo for binário. Em **VCSs** esse trabalho pode ser um pouco menos tedioso.
- **Prevenção de desastres**: Na maioria esmagadora dos casos, recuperar erros do usuario é mais facil em *VCSs*. (**na maioria**).

Sistemas de Controle de Versão

Temos vários tipos de Sistemas de Controle de Versão. Aqueles locais, em que diferenças entre arquivos são guardados em outros arquivos especiais em disco, aqueles sofisticados e centralizados – que guardam as modificações em um servidor central– como o Perforce e os *VCSs* descentralizados, como o **git**, que operam com diferentes árvores de histárias de modificações espalhadas em múltiplos hosts.

Em linhas gerais, *VCSs* descentralizados são melhores na maioria dos casos, visto que assim é elimininado a desvantagem de um único ponto de falha. Além disso, esse tipo combina com o modelo *Open Source* descentralizado que temos atualmente.

História breve do Git

No inicio de seu desenvolvimento, o **kernel linux** era desenvolvido distribuidamente utilizando *patches* de código enviados por emails (*no kernel isso é muito usado ainda*).

Porém, a medida que o escopo do projeto foi aumentando, a necessidade de um versionamento correto foi ficando cada vez mais evidente. Assim, o kernel começou a usar um *VCS* proprietario chamado *BitKeeper*.

Mas, o desenvolvimento do *BitKeeper* cessou, e a necessidade de um novo sistema se tornou prioridade. Assim, **Linus Torvalds desenvolveu o Git para preencher essa lacuna**.

O que é o Git?

> "Backups are for wimps. Real men upload their data to an FTP site and have everyone else mirror it." – (Linus Torvalds).

Em seu núcleo, o git armazena os seus dados como uma lista de *snapshots* de todos os arquivos em função do tempo. Pense em um *snapshot* como uma "fotografia" do estado de todos os arquivos em um dado momento.

Sendo assim, *in a nutshell*, **o git é um sistema de arquivos baseado em snapshots com um conjunto de ferramentas construidos em cima dele**.

Dado sua natureza distribuida, **a maioria das operações no git são feitas localmente**, ou seja, informações transferidas por rede são necessárias em apenas alguns tipos de operações. Isso torna o git muito rápido, até em operações mais custosas.

Cada snapshot disponível em um repositório é univocamente identificado por um cédigo hash **sha-1**(ou **sha-256**). Esse hash garante a identificação e integridade de cada snapshot, e é central na filosofia do git. Um hash no git possui a seguinte forma: `24b9da6552252987aa493b52f8696cd6d3b00373`.

Conceitos do Git

> "Intelligence is the ability to avoid doing work, yet getting the work done." – (Linus Torvalds).

O conceito mais importante do git é seu **Workflow em três estados**, ou seja, cada arquivo pode estar:

- ➔ **Commited**: Os dados estão guardados de maneira segura no banco de dados local.
- ➔ **Modified**: Os dados estão modificados mas as alterações não foram gravadas no banco de dados local.
- ➔ **Staged**: Os dados modificados foram marcados para serem adicionados no próximo commit.

Assim, temos as **três áreas do git** que um arquivo pode estar:

- ➔ **Diretório**: Seus arquivos em disco.
- ➔ **Stagging Area**: Área especial do git para arquivos modificados e adicionados.
- ➔ **Repositório**: Área do git em que todos os *snapshots*(*commits*) são salvos como *refs*.

Você **modifica** um arquivo no diretório do projeto, **adiciona** ele com o 'git add arquivo.txt', e após isso **commita** as modificações para criar um snapshot com o `git commit -m "Descricao"`.

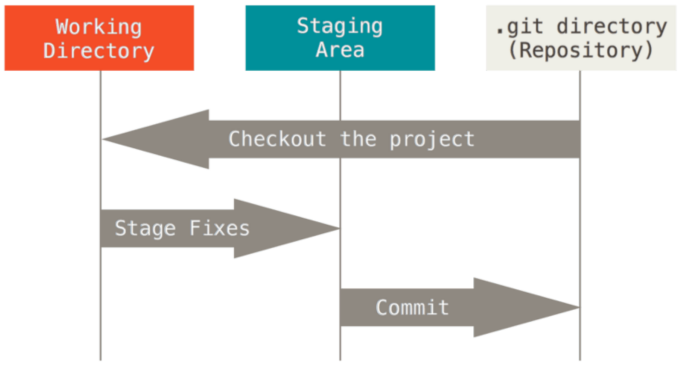

Figura 1: Flow básico do Git

Comandos Básicos

Tendo em vista os conceitos apresentados anteriormente, agora serão apresentados os comandos mais importantes do Git, para que o leitor tenha uma ideia das operações mais importantes e necessárias.

Projetos e Repositorios

Podemos criar um repositório Git de duas maneiras: **criar um localmente** ou **clonar um repositório remoto**.

Para iniciar um repositorio local usamos o comando `git init <dir>`. Por exemplo, para criar um repositório na pasta `repo` usamos o comando:

```
mkdir repo
git init repo
```

Também, podemos **clonar um repositório remoto**, ou seja, copiar para nossa máquina um repositório que esteja guardado em outro computador. Para isso, usamos o comando `git clone <repo>`, onde `repo` é um endereço remoto de rede. Exemplificando, para clonar o repositório Git do próprio código fonte do Git guardado no *github*, usamos o comando:

```
git clone https://github.com/git/git
```

Configuração e Ajuda

O comando `git help <command>` é usado para imprimir a ajuda e a utilizaçãoo dos comandos do Git. Sempre utilize ele caso esteja em dúvida sobre alguma operação. Por exemplo, para visualizar a ajuda do comando de configuração:

```
git help config
```

Por sua vez, o comando utilizado para configurar diversos aspectos do git é o `git config <option>`. Existem uma infinidade de opções para se configurar no Git, por exemplo para mudar as suas credenciais em um repositório, utilize o comando:

```
git config --global user.name "Nome"
git config --global user.email email@example.com
```

Commits e Workflow

Como apresentado anteriormente, o flow de trabalho básico no git é: $modificar \rightarrow adicionar \rightarrow commitar$ os arquivos.

Sendo assim, o comando `git add <arquivos>` adiciona os arquivos especificados na *stagging area*.

É possével utilizar os comandos `git status` e `git diff` para obter, respectivamente, informações sobre os estados dos arquivos em relação as três áreas e as diferenças introduzidas desde o último commit. Modifique um arquivo e teste esses comandos:

```
echo "a" >> arquivo.txt
git status
git diff
git add arquivo.txt
git status
git diff
```

Seguindo, após modificar arquivos, o próximo passo é adicioná-los para o repositório de fato, utilizando o comando `git commit`. Ao rodar esse comando, seu editor de texto padrão será aberto para você escrever uma **mensagem de commit**. Cada commit possui uma mensagem associada, que produz algum contexto para o usuário de quais modificações foram inseridas nele.

```
git commit
```

Porém, podemos escrever a mensagem *inline* no comando de commit, usando:

```
git commit -m "Mensagem"
```

Por fim, quando realizarmos um commit faltando arquivos, ou esquecemos de adicionar uma modificação em um arquivo num commit específico, podemos utilizar a flag `--amend --no-edit` para "recriar" o ultimo commit com o estado atual dos arquivos, também não é necessário editar a mensagem:

```
git add <arquivos>
git commit --amend --no-edit
```

Seguindo, o comando `git reset` é utilizado principalmente para, sem surpresa alguma, *resetar* alterações. Ele move o ponteiro `HEAD` (ponteiro do grafo de commits) entre os commits, e possivelmente muda a stagging area. Assim, para remover um ou mais arquivos apos adicioná-los na área de commit(stagging) usamos o comando:

```
echo "modificacao" >> file.txt
git add file.txt
# ops, vamos remover file.txt da area de commits
git reset HEAD -- file.txt
```

Uma tarefa rotineira em repositórios git e deletar e mover arquivos, tanto do diretório atual como da *stagging area*. Podemos lidar com eles utilizando os comandos do sistema operacional `rm` e `mv`, e após isso utilizar o comando `git add` para atualizar a *stagging area*. Porém, o git nos oferece os *wrappers* `git rm` e `git mv` que fazem essas duas ações de uma vez só:

```
# Apagar o arquivo
git rm file1.txt
# Mover o arquivo
git mv file2.txt file3.txt
```

Um comando muito útil também é o `git clean`, que apaga todos os arquivos desconhecidos para o git do repositório atual. Isso é muito util para, por exemplo, apagar artefatos de build do projeto.

Branches e Merges

Uma dos conceitos centrais do git é o de **Branches**. Uma *branch*, de maneira superficial, é uma bifurcação no grafo de commits, indicando que a partir daquele commit os outros commits seguirão um caminho diferente. No exemplo a seguir, ao criar uma branch `testing`, a história de commits fica da seguinte forma:

```
git branch testing
```

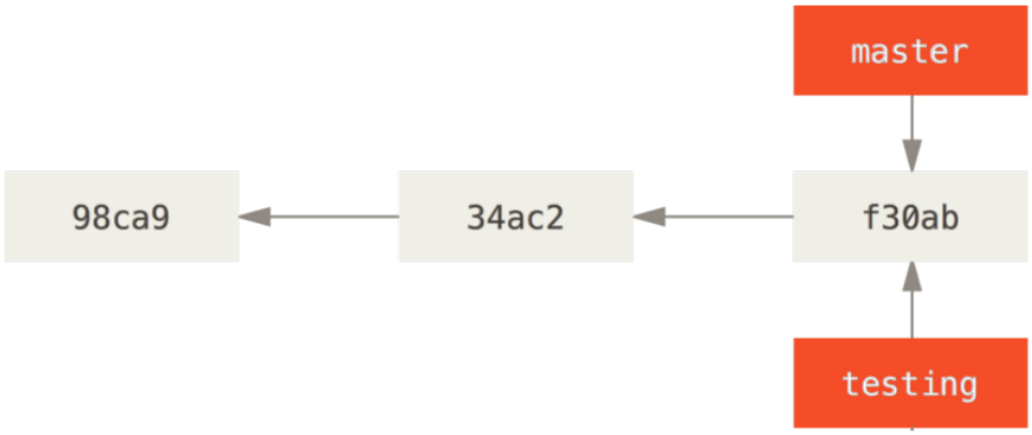

Figura 2: Criação da branch testing

Porém, após a historia ser modificada, e ser adicionado commits em ambas as branches `master` e `testing` a história fica da forma:

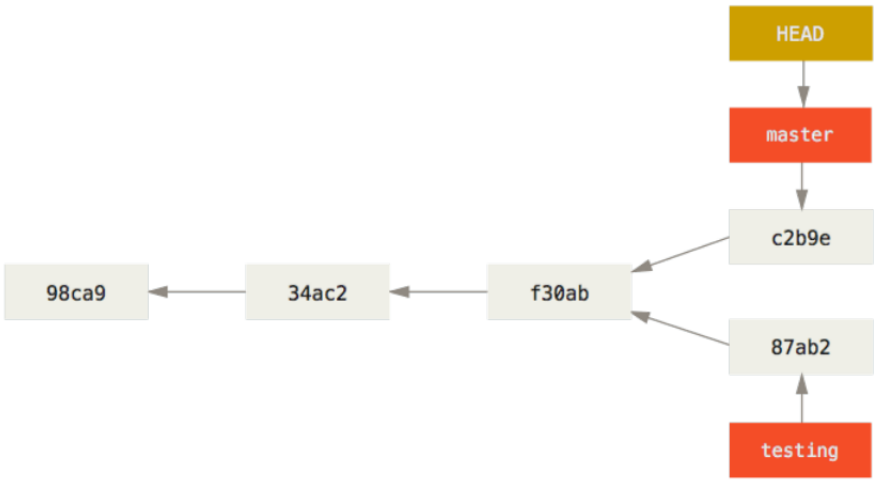

Figura 3: História apos commits em ambas as branches

Tambem, podemos criar uma branch com o comando `git checkout -b <name>`. Note o uso da flag `-b`.

O comando `git checkout` é usado principalmente para navegar entre branches.

```
# Cria uma branch e ja muda pra ela
git checkout -b testing

# Cria uma branch
git banch testing
# Muda para essa branch
git checkout branch
```

Assim, ao terminar o trabalho em uma *branch* e precisarmos integrar o trabalho em outra, utilizamos o comando `git merge <other-branch>`:

```
git checkout main
git merge testing
```

Após um *merge* bem sucedido, a história do repositório ficará parecida com a seguinte:

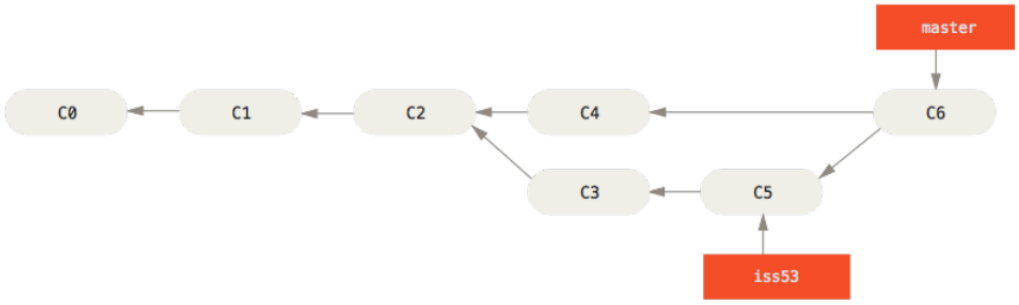

Figura 4: Historia apos um merge

Com o objetivo de mesclar as alterações de duas branches, podemos utilizar o `git rebase <dest>`, que "recria" os commits da branch atual e os reaplica no topo da branch alvo.
As seguintes figuras comparam os resultados de um `merge` e de um `rebase`:

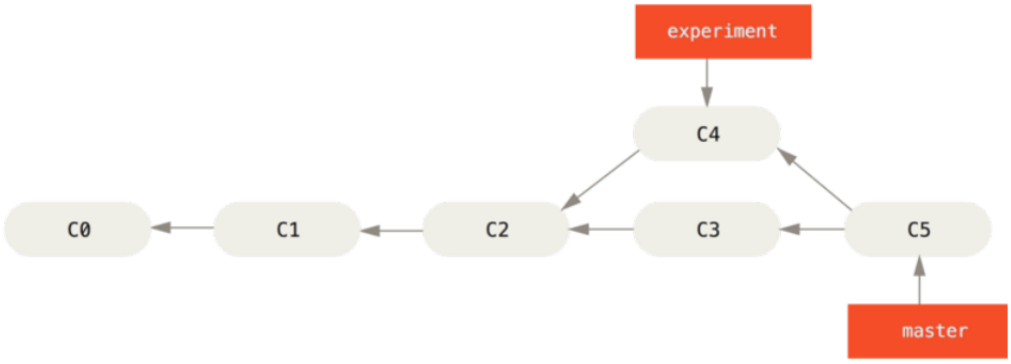

Figura 5: História após um merge

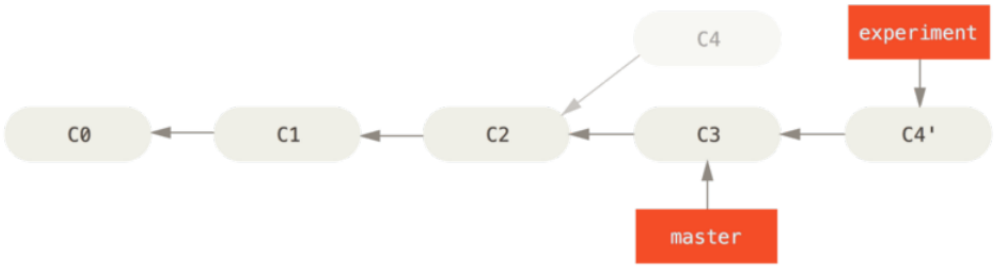

Figura 6: História após um rebase

Em alguns merges mais complicados, geralmente onde você modificou a mesma parte de um arquivo em branches diferentes, haverá os chamados *merge conflicts*. Por exemplo, se modificarmos a mesma linha de um arquivo de texto como a seguir:

```
diff --cc arquivo.txt
index a056732,94e7826..0000000
--- a/arquivo.txt
+++ b/arquivo.txt
@@@ -1,2 -1,2 +1,8 @@@
  primeira linha: yes
++<<<<<<< HEAD
 +segunda linha: perharps
++||||||| cdf8dc3
++segunda linha: no
++=======
+ segunda linha: yes
++>>>>>>> feature
```

Figura 7: Modificações na mesma linha de um arquivo

Seguindo com o merge, ele dará *conflict*, pois temos 3 versões daquela mesma linha modificada: A versão no topo da branch *main*, a versão no topo de *feature* e a versão do commit ancestral comum entre as duas *branches*. Assim, o git irá inserir *marcadores de conflito* no arquivo a fim de sinalizar essas diferenças. Cabe ao usuário modificar o arquivo, produzir uma versão correta e seguir com o merge.

Podemos fazer isso manualmentem, ou usando as chamadas *mergetools*. Após um conflito ser sinalizados usamos o seguinte comando para invocar uma *mergetool*:

```
git mergetool
```

Está fora do contexto desse material entrar em detalhes sobre essas ferramentas, porém podemos usar os editores de texto mais conhecidos como **mergetool**, como por exemplo o *vim, emacs, vscode, etc.

Após resolver o conflito, o proximo passo e dar o comando `git status` para verificar o status do merge e `git commit` para prosseguir com o merge.

Um dos comandos mais úteis – e desconhecidos – do git é o `git stash`. A *stash* no git e, como o nome indica, uma pilha utilizada para guardar modificações temporárias. Por exemplo, imagine que

voce está trabalhando em uma branch A, e já realizou varias modificações nela. Porém, agora você precisa realizar modificações em uma branch B, mas você não consegue mudar para ela pois ainda existe trabalho não commitado em A. Justamente nesse caso a *stash* se torna indispensável, pois assim você irá rodar o comando `git stash` na *branch A*, mudar para a branch B e realizar o trabalho lá, e quando você voltar pra a branch B, poderá rodar o comando `git stash pop` para reaplicar o trabalho temporário que você salvou anteriormente.

Logs e Visualização

O commando `git log` é usado para mostrar a história de commits da branch atual. Esse comando possui uma infinidade de opções, um exemplo de modificacao nele é o seguinte:

```
    git config --global alias.lg2 "log --graph --abbrev-commit --decorate\
                          --date=format:'%Y-%m-%d %H:%M:%S'\
                          --format=format:'%C(bold blue)%h%C(reset)\
                              - %C(bold cyan)%ad%C(reset) %C(bold green)\
                              (%ar)%C(reset)%C(bold yellow)%d%C(reset)%n''\
                              %C(white)%s%C(reset) %C(dim white)- %an%C(reset)'"
```

Essa customização gerará saídas de log da seguinte maneira:

```
* 1fc3c0ad40 - 2022-10-21 11:37:36 (5 days ago) (HEAD -> master, origin/master, origin/main, origin/HEAD)
|           The fifth batch - Junio C Hamano
*   c2058ea237 - 2022-10-21 11:37:29 (5 days ago)
|\            Merge branch 'rj/branch-edit-description-with-nth-checkout' - Junio C Hamano
| * 0dc4e5c574 - 2022-10-11 01:24:58 (2 weeks ago)
| |           branch: support for shortcuts like @{-1}, completed - Rubén Justo
* |   1f20aa22d7 - 2022-10-21 11:37:28 (5 days ago)
|\ \            Merge branch 'ds/cmd-main-reorder' - Junio C Hamano
| * | 413bc6d20a - 2022-10-08 16:21:37 (3 weeks ago)
| | |           git.c: improve code readability in cmd_main() - Daniel Sonbolian
* | |   91d3d7e6e2 - 2022-10-21 11:37:28 (5 days ago)
|\ \ \            Merge branch 'ab/grep-simplify-extended-expression' - Junio C Hamano
| * | | db84376f98 - 2022-10-11 11:48:45 (2 weeks ago)
| | | |           grep.c: remove "extended" in favor of "pattern_expression", fix segfault - Ævar Arnfjörð Bjarmason
* | | |   4a48c7d25f - 2022-10-21 11:37:28 (5 days ago)
|\ \ \ \            Merge branch 'jc/symbolic-ref-no-recurse' - Junio C Hamano
| * | | | b77e3bdd97 - 2022-10-07 15:00:39 (3 weeks ago)
| |/ / /            symbolic-ref: teach "--[no-]recurse" option - Junio C Hamano
* | | |   6269c46ada - 2022-10-21 11:37:27 (5 days ago)
|\ \ \ \            Merge branch 'jk/use-o0-in-leak-sanitizer' - Junio C Hamano
| * | | | d3775de074 - 2022-10-18 16:15:33 (8 days ago)
| | | | |           Makefile: force -O0 when compiling with SANITIZE=leak - Jeff King
* | | | |   cc7574322f - 2022-10-21 11:37:27 (5 days ago)
|\ \ \ \ \            Merge branch 'ab/macos-build-fix-with-sha1dc' - Junio C Hamano
| * | | | | 32205655dc - 2022-10-19 03:03:19 (7 days ago)
| | | | | |           fsmonitor OSX: compile with DC_SHA1=YesPlease - Ævar Arnfjörð Bjarmason
* | | | | | 45c9f05c44 - 2022-10-19 14:25:03 (7 days ago)
| | | | | |           The fourth batch - Junio C Hamano
```

Figura 8: Log customizado

Temos também o comando `git shortlog`, que basicamente sumariza todas as informações contidas em um *log comum*.

Por fim, o comando `git show <commit>` é bastante útil, pois nos mostra um resumo de todas as informações de um commit, como sua mensagem e as modificações inseridas.

Trabalhando com repositórios remotos

Com o objetivo de gerenciar repositórios remotos, o git nos oferece o comando `git remote`. Com ele, podemos adicionar *remotes* usando `git remote add <name> <url>` e listar *git remote -v*

O comando `git fetch` se comunica com um repositório remoto e baixa todas as referências daquele repositório para o seu repositório local.

Já o comando `git pull` é uma combinação dos comandos `git fetch` e `git merge`, pois ele baixa todas as informações do repositório especificado e tenta mesclar elas com o seu repositorio local.

Por fim, para fazer upload de todas as referências do seu repositorio local para um remoto, usamos o comando `git push`.

Github

Github é exatamente o que é o nome indica, um hub de repositórios git compartilhado entre vários usuários. Você pode criar uma conta nessa plataforma e hospedar de graça seus repositórios sem a necessidade de manter um servidor git por si próprio.

Devido a essa natureza aberta e compartilhada é uma das maiores plataformas de código aberto do mundo, fomentando a indústria FOSS (Free and Open Source).

Obviamente por um conceito simples de hospedagem compartilhada há diversos competidores como GitLab, BitBucket (que é mais reservado), Gitea, Gitee, Source Hut, Gerrit e muitos outros.

Perfil

Hoje em dia o Github serve muito como o portfólio do desenvolvedor, pois consegue facilmente promover seus feitos de uma forma clara e muito fácil. Se tornando também uma rede social de certa forma, onde você consegue seguir colegas e desenvolvedores que trabalham com coisas que você gostaria de acompanhar.

Todo usuário tem seu perfil no github que serve de porta de entrada para seus repositórios, sendo acessível no formato https://github.com/nome-usuario.

Uma coisa muito comum que será abordada também na sessão 2.2 são os *README*s. Esse padrão de ter um texto explicando uma aplicação ou algo similar é beeem antigo tendo uma possível origem nos repositórios UNIX.

Hoje em dia os README servem muito mais como a capa do livro que você vai ler, se ela não vender bem a ideia do produto o usuário não vai ver o conteúdo, então tornou-se uma arte fina de atingir a construção de um belo README.

Você pode criar um README não só para servir de porta de entrada para seu repositório como para seu perfil ou para sua organização.

Repositórios

Os repositórios no github são a forma de armazenar remotamente seus repositórios criados no git. Serve para se obter ou publicar as informações de um repositório local.

Ainda que hoje o Github esteja tão abstraído que você consegue fazer edições sem precisar (aparentemente) ter uma cópia local para fazer as alterações.

Para além disso, os repositórios tem muitos módulos que tornam sua usabilidade maior propondo uma grande versatilidade, esses módulos tem vários propósitos, como mapear atividades (Issues), realizar atividades e submeter ao repositório (Pull requests), clonar o repositório, mas fazendo uma "versão"para si (Fork), Gerenciar projetos e Milestones (Projects), distribuir versões de software (Releases), integração contínua e entre contínua com o (Actions).

Com todas essas peças os repositórios acabam sendo muito mais do que apenas um local que guarda seu código versionado.

Gists

Os gists são uma maneira mais simples de quando se quer compartilhar apenas algumas informações como um arquivo só ou um trecho só do código com a comunidade ou com colegas.

Github.io

O Github oferece também uma plataforma de hospedagem com domínio *github.io* que você pode por seus sites.