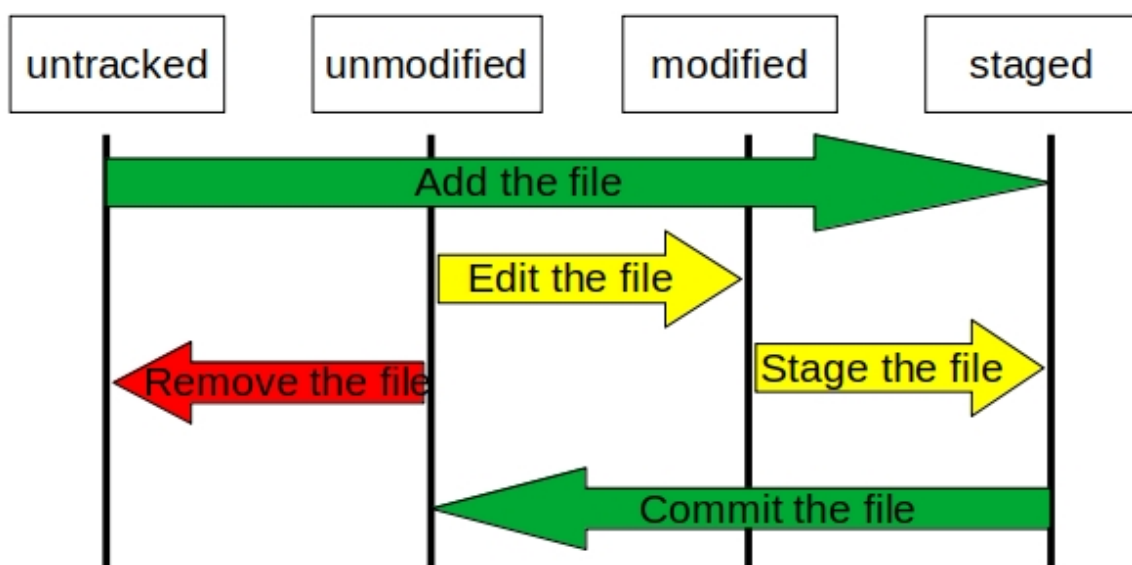
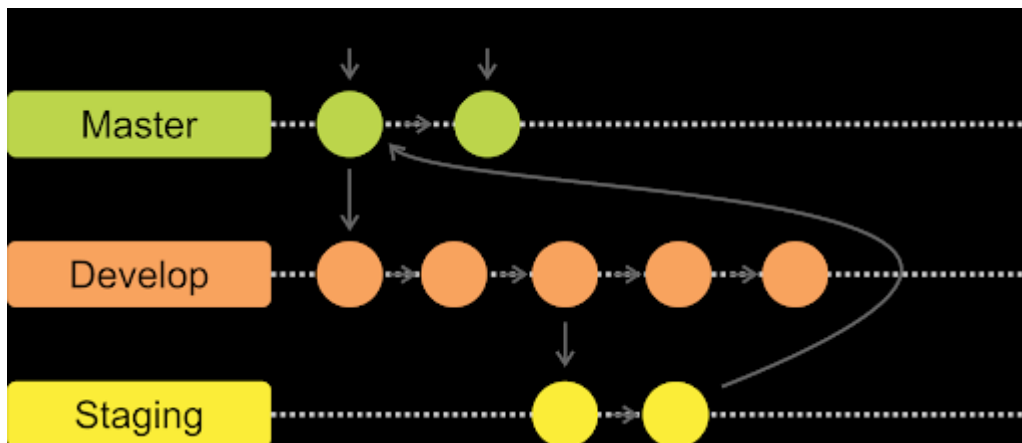


A figura abaixo traz um exemplo de um repositório Git seguindo o fluxo de trabalho Gitflow. Podemos enxergar o repositório de código como uma árvore de versões de código. Inicialmente, quando criamos um repositório, há somente um **ramo** (do inglês, **branch**), com o nome “**master**” que representa o ramo principal (versão v 0.0.0). Posteriormente a criação do repositório, seguindo o modelo de fluxo de trabalho Gitflow, devemos criar um ramo com o nome “**develop**”, a partir do branch master. É este ramo que receberá todas as modificações para correção ou evolução de funcionalidades já existentes, assim como para o desenvolvimento de novas funcionalidades. O ramo “**staging**” também é criado a partir do ramo “develop” e é utilizado para colocar em testes as modificações de código realizadas pelos desenvolvedores. Somente quando as modificações são aprovadas em todas as rotinas de testes existentes, isto é, testes regressivos, a versão do ramo “staging” é mesclada à versão “master”, dando origem a uma versão evoluída do software (v 0.0.1). Este processo de verificação por testes regressivo confere qualidade ao software ao garantir que a nova versão do software passa em todos os testes da versão anterior e também em novos testes que verificam as modificações realizadas no código.



Agora que já entendemos um pouco sobre o git, vamos partir para a prática. A primeira coisa que devemos fazer quando queremos trabalhar com git, é criar um repositório, fazemos isso com o comando **git init**. Crie um diretório local para o código fonte do seu software e, em seguida, execute este comando. Esse comando cria um subdiretório `.git` com todos os metadados necessários. No exemplo abaixo, criamos o diretório "hello_world".

```
(base) higor@Higor:~/Desktop$ mkdir hello_world
(base) higor@Higor:~/Desktop$ cd hello_world/
(base) higor@Higor:~/Desktop/hello_world$ git init
initialized empty Git repository in /home/higor/Desktop/hello_world/.git/
(base) higor@Higor:~/Desktop/hello_world$
```

Vamos criar um arquivo `README.md` em nosso projeto. Este arquivo é apenas um arquivo texto que, em geral, acompanha projetos de software explicando questões iniciais para seu uso ou instalação. Depois, vamos adicionar este arquivo à staging area para que faça parte do próximo commit. Fazemos isso com o comando **git add**. O comando **touch** do Linux é usado para criar arquivos vazios, além de alterar o registro de data e hora (timestamp) de arquivos ou pastas.

```
~/Desktop/hello_world$ touch README.md
~/Desktop/hello_world$ git add README.md
```

O comando **git status** pode ser utilizado para verificar que o arquivo `README.md` foi adicionado a staging area.

```
(base) higor@Higor:~/Desktop/hello_world$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   README.md
```

Agora vamos realizar o **commit** de nossas alterações, para que os arquivos na staging area passem a ser versionados. Só arquivos que sofreram commit terão suas versões controladas pelo Git. Para isso utilizamos o comando **git commit -m "Mensagem do commit"**. A mensagem passada como parâmetro deve ser utilizada com inteligência para que seja útil ao time de desenvolvimento, ela deve justificar e explicar as alterações pelas quais o código fonte passou.

```
(base) higor@Higor:~/Desktop/hello_world$ git commit -m "Add README.md"
[master (root-commit) 98a71ed] Add README.md
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 README.md
```

Utilizando o comando **git log** podemos ver o histórico de commits realizados em nosso repositório.

```
(base) higor@Higor:~/Desktop/hello_world$ git log
commit f861569c65b969f9d49da8384a722fdbf807148b (HEAD -> master)
Author: Higor Duarte
Date: Sat May 29 09:49:10 2021 -0300
```

Ignore os arquivos que não lhe interessa versionar

Muitas vezes, em um projeto de software, haverá arquivos binários que não devem ser adicionados a um repositório de código, evitando que a ferramenta Git tente realizar o controle de versão dos mesmos. Para exemplificar, vamos criar um arquivo **main.cpp** que imprime a mensagem “Hello World!” na tela e, em seguida, compilá-lo.

```
#include<iostream>

using namespace std;

int main(){
    cout << "Hello World!\n";
}
```

Após compilarmos, podemos verificar que foi gerado o arquivo **main** que é nosso executável.

```
(base) higor@Higor:~/Desktop/hello_world$ ls
main  main.cpp  README.md
```

Porém, nós não queremos realizar um commit com esse arquivo. Para resolver isso, vamos criar o arquivo **.gitignore** onde vamos colocar todos os arquivos que queremos ignorar ao realizar um commit. O comando **echo** é utilizado apenas para adicionar a nova linha “main” ao arquivo **.gitignore**.

```
~/Desktop/hello_world$ touch .gitignore
~/Desktop/hello_world$ echo "main" >> .gitignore
```

Agora vamos salvar nossas alterações adicionando os arquivos **main.cpp** e **.gitignore** no staging. Podemos fazer isso através do comando “**git add .**”, que adiciona todos os arquivos modificados no staging.

```
~/Desktop/hello_world$ git add .
```

Se checarmos o estado do nosso repositório, podemos observar que todas as modificações foram adicionadas ao staging.

```
(base) higor@Higor:~/Desktop/hello_world$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   .gitignore
    new file:   main.cpp
```

Se tentarmos agora adicionar o executável main no staging, recebemos um aviso.

```
(base) higor@Higor:~/Desktop/hello_world$ git add main
The following paths are ignored by one of your .gitignore files:
main
Use -f if you really want to add them.
```

Se quisermos adicionar um arquivo ignorado ao staging, devemos utilizar a opção -f no comando git add, para forçar a adição.

Agora podemos realizar o commit das nossas alterações

```
(base) higor@Higor:~/Desktop/hello_world$ git commit -m "Add main.cpp"
[master 28dda67] Add main.cpp
2 files changed, 8 insertions(+)
create mode 100644 .gitignore
create mode 100644 main.cpp
```

Utilizando o comando git log, podemos observar que agora temos dois commits.

```
(base) higor@Higor:~/Desktop/hello_world$ git log
commit 28dda6741092523448b9a51a6c81fdbb3a87e080 (HEAD -> master)
Author: Higor Duarte
Date: Sat May 29 10:28:24 2021 -0300

    Add main.cpp

commit f861569c65b969f9d49da8384a722fdbf807148b
Author: Higor Duarte
Date: Sat May 29 09:49:10 2021 -0300

    Add README.md
```

Restaurando versões anteriores do código

Mas e se nos arrependermos de algum commit e desejarmos desfazê-lo? Bom, para isso temos dois comandos, **git reset** e **git revert**.

No comando **git reset** existem três opções, **soft**, **mixed** e **hard**. A opção **soft**, move o HEAD para o commit indicado, mas mantém o staging e o working directory inalterados. A opção **mixed**, move o HEAD para o commit indicado, altera o staging e mantém o working directory. A opção **hard** faz com que o HEAD aponte para algum commit anterior, mas também altera a staging area e o working directory para o estado do commit indicado, ou seja, todas as alterações realizadas após o commit ao qual retornamos serão perdidas. Isso não é recomendável se as alterações já tiverem sido enviadas para o repositório remoto. Nesse caso devemos utilizar o **git revert**.

Vamos supor que queremos desfazer as alterações do último commit em nosso exemplo, utilizando o comando **git reset** com a opção **hard**. Dessa forma precisamos indicar ao comando **git reset** o código SHA-1 do commit ao qual queremos que o HEAD aponte. Outra forma de utilizar o comando **git reset**, é indicando quantos commits queremos retornar o HEAD, fazemos isso com o comando **git reset HEAD~n**. O parâmetro **HEAD~n**, nos indica que queremos posicionar o HEAD para **n** commits atrás. Por exemplo, para retornar para o commit anterior usamos **HEAD~1**.

```
(base) higor@Higor:~/Desktop/hello_world$ git reset --hard f861569c65
HEAD is now at f861569 Add README.md
(base) higor@Higor:~/Desktop/hello_world$ git log
commit f861569c65b969f9d49da8384a722fdbf807148b (HEAD -> master)
Author: Higor Duarte <[REDACTED]>
Date: Sat May 29 09:49:10 2021 -0300

    Add README.md
(base) higor@Higor:~/Desktop/hello_world$ ls
main README.md
```

Na última linha da figura acima, podemos observar que o arquivo **main.cpp** que havíamos adicionado no segundo commit, não está mais em nosso working directory.

Por outro lado, o comando **git revert** cria um novo commit com as alterações do commit indicado. Utilizando **git revert** em nosso exemplo, nosso repositório ficaria assim

Como podemos verificar ao lado,
um novo commit foi criado
revertendo para as
alterações do commit f861569.

```
(base) higor@Higor:~/Desktop/hello_world$ git revert f861569c65b96
Removing README.md
[master c9c3918] Revert "Add README.md"
1 file changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 README.md
(base) higor@Higor:~/Desktop/hello_world$ git log
commit c9c3918db1b7c3a4cf65f7eb2daa44fba599a5bf (HEAD -> master)
Author: Higor Duarte <[REDACTED]>
Date: Sat May 29 10:43:43 2021 -0300

    Revert "Add README.md"

    This reverts commit f861569c65b969f9d49da8384a722fdbf807148b.

commit 28dda6741092523448b9a51a6c81fdbb3a87e080
Author: Higor Duarte <[REDACTED]>
Date: Sat May 29 10:28:24 2021 -0300

    Add main.cpp

commit f861569c65b969f9d49da8384a722fdbf807148b
Author: Higor Duarte <[REDACTED]>
Date: Sat May 29 09:49:10 2021 -0300

    Add README.md
```

Entendendo o uso de um repositório remoto

Agora que já aprendemos como utilizar o Git no repositório local, vamos entender como utilizamos o Git com um repositório remoto seguindo o fluxo de trabalho Gitflow.

Vamos supor que tenhamos um repositório em um projeto do **Gitlab**. Nosso primeiro passo é utilizar o comando **git clone** para baixar o repositório em nossa máquina.

```
(base) higor@Higor:~/Desktop$ git clone https://gitlab.com/tutorial-git2/hello-world.git
Cloning into 'hello-world'...
Username for 'https://gitlab.com': 
Password for 'https://@gitlab.com': 
warning: You appear to have cloned an empty repository.
```

Nosso repositório está vazio, então agora vamos realizar nosso primeiro commit.

```
(base) higor@Higor:~/Desktop/hello-world$ touch README.md
(base) higor@Higor:~/Desktop/hello-world$ git add README.md
(base) higor@Higor:~/Desktop/hello-world$ git commit -m "add README"
[master (root-commit) c7a8dcbl] add README
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 README.md
```

Vamos subir as alterações para o repositório remoto no Gitlab. Fazemos isso com o comando **git push -u origin master**.

```
(base) higor@Higor:~/Desktop/hello-world$ git push -u origin master
Username for 'https://gitlab.com': 
Password for 'https://@gitlab.com': 
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 219 bytes | 5.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://gitlab.com/tutorial-git2/hello-world.git
* [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

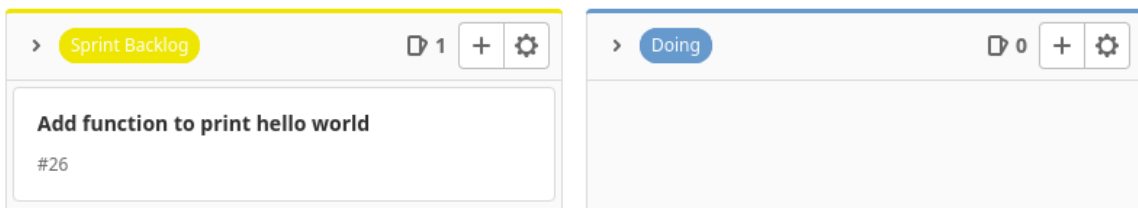
Podemos observar que o ramo master foi criado em nosso repositório no Gitlab. Agora precisamos criar os ramos develop e staging. Para isso podemos utilizar o comando **git checkout -b develop master**, que cria o ramo develop a partir do ramo master. Realizamos o mesmo procedimento para o ramo staging com o comando **git checkout -b staging develop**. Após criarmos os ramos, utilizamos os comandos **git push -u origin develop** e **git push -u origin staging** para subir as alterações no repositório remoto.

```
(base) higor@Higor:~/Desktop/hello-world$ git checkout -b develop master
Switched to a new branch 'develop'
(base) higor@Higor:~/Desktop/hello-world$ git branch
* develop
  master
```

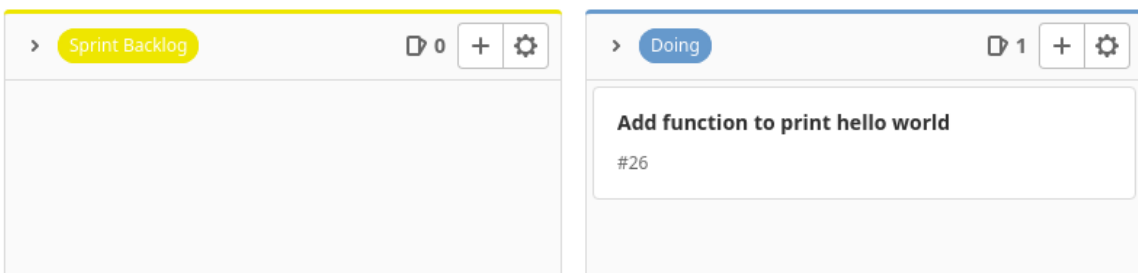
Utilizando o comando **git branch** podemos verificar os ramos em nosso repositório local. Para verificarmos os ramos remotos, utilizamos o comando **git branch -r**.

Resolvendo issues no GitLAB

Agora vamos supor que durante as reuniões realizadas com seu time, seu líder (em geral, um *Product Owner*) definiu e atribuiu a você uma funcionalidade que deverá ser desenvolvida no próximo ciclo de desenvolvimento (*sprint*), que em nosso exemplo é a funcionalidade “Add function to print hello world”. Para isso, no GitLab, ele adicionou essa **issue** ao board “**Sprint backlog**”, conforme ilustra a figura a abaixo.



Antes de começar a implementar esta funcionalidade, precisamos a nova issue para o board “**Doing**”, conforme ilustra a figura abaixo.



Agora, precisamos criar um novo ramo onde para essa *issue* será resolvida, no jargão da área chamamos esse ramo de **feature branch**. Para isso, o criamos a partir do ramo develop e começamos a desenvolver a funcionalidade. Faremos isso clicando com o mouse no nome da issue, então seremos redirecionados para a página com a definição da issue, conforme ilustra a figura abaixo. Depois, vamos clicar no botão “Create merge request” e digitar no campo “Source (branch ou tag)” o nome da branch na qual iremos trabalhar, em nosso exemplo o branch “develop”, e por fim clicar em “Create merge request”.

Add function to print hello world



Drag your designs here or [click to upload](#).

Linked issues 0 0 +



0



0



Oldest first ▾

Show comments only ▾

Create merge request ▾

Write Preview

B *I* ”

Write a comment or drag your files here...

Markdown and [quick actions](#) are supported

Comment ▾

Close issue

✓ Create merge request and branch
Create branch

Branch name

1-add-function-to-print-hello-work

Source (branch or tag)

develop

Source is available

Create merge request

Então, seremos redirecionados para a tela de definição do merge request, apresentada abaixo. Perceba que não é possível realizar o merge, pois ainda não existem alterações na nova branch criada.

Draft: Resolve "Add function to print hello world"

Overview 0 Commits 0 Changes 0

Closes #1

Request to merge 1-add-function-to-p... into develop

Open in Web IDE

Check out branch



This merge request contains no changes.

Use merge requests to propose changes to your project and discuss them with your team. To make changes, push a commit or edit this merge request to use a different branch. With [CI/CD](#), automatically test your changes before merging.

Create file



0



0



Oldest first ▾

Show all activity ▾

Ao entrar na página **Active branches** do repositório remoto, podemos observar que foi criado um ramo cujo nome é definido pelo número e nome da issue que deu origem ao ramo.

Active branches			
Y 1-add-function-to-print-hello-world ↗ c7a8dcba · add README · 1 hour ago	0 0	Merge request	Compare
Y develop ↗ c7a8dcba · add README · 1 hour ago	0 0	Merge request	Compare
Y master default protected ↗ c7a8dcba · add README · 1 hour ago			






Agora, é preciso atualizar nosso repositório local baixando o novo branch criado. Fazemos isso com os comandos **git fetch origin nome_branch_nova** que irá buscar o branch criado do repositório remoto e **git checkout nome_branch_nova**, que cria um branch local que rastreia o branch remoto que acabamos de criar.

```
higor@higor-Aspire-ES1-572:~/Desktop/hello-world$ git fetch origin 1-add-function-to-print-hello-world
Username for 'https://gitlab.com':
Password for 'https://gitlab.com':
From https://gitlab.com/tutorial-git2/hello-world
 * branch 1-add-function-to-print-hello-world -> FETCH HEAD
higor@higor-Aspire-ES1-572:~/Desktop/hello-world$ git checkout 1-add-function-to-print-hello-world
Branch '1-add-function-to-print-hello-world' set up to track remote branch '1-add-function-to-print-hello-world'
from 'origin'.
Switched to a new branch '1-add-function-to-print-hello-world'
higor@higor-Aspire-ES1-572:~/Desktop/hello-world$ git branch
* 1-add-function-to-print-hello-world
  develop
  master
higor@higor-Aspire-ES1-572:~/Desktop/hello-world$
```

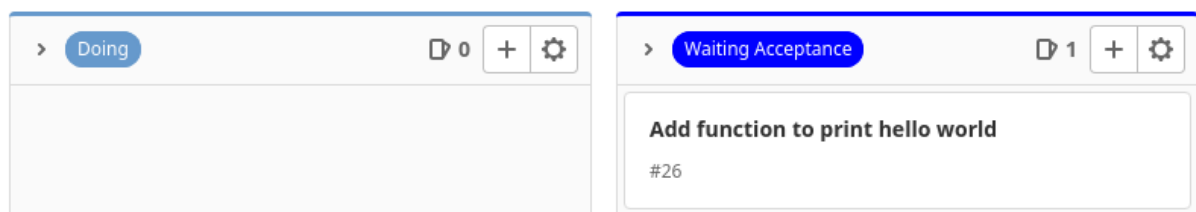
Finalmente, após implementar a funcionalidade, devemos realizar o commit e enviar para o repositório remoto através do comando **git push**.

```
(base) higor@Higor:~/Desktop/hello-world$ git add .
(base) higor@Higor:~/Desktop/hello-world$ git commit -m "feat: Added function to print hello world"
[1-add-function-to-print-hello-world f4b4e4c] feat: Added function to print hello world
2 files changed, 8 insertions(+)
create mode 100644 .gitignore
create mode 100644 main.cpp
(base) higor@Higor:~/Desktop/hello-world$ git push
Username for 'https://gitlab.com': higorduarte98
Password for 'https://higorduarte98@gitlab.com':
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 433 bytes | 433.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0)
remote:
remote: View merge request for 1-add-function-to-print-hello-world:
remote: https://gitlab.com/tutorial-git2/hello-world/-/merge_requests/1
remote:
To https://gitlab.com/tutorial-git2/hello-world.git
c7a8dcba..f4b4e4c 1-add-function-to-print-hello-world -> 1-add-function-to-print-hello-world
```

Como podemos ver na página principal do projeto, apresentada abaixo, o repositório remoto no Gitlab foi atualizado.

 feat: Added function to print hello world Higor Duarte authored 4 minutes ago			f4b4e4c4	
Name	Last commit	Last update		
 .gitignore	feat: Added function to print hello world	4 minutes ago		
 README.md	add README	2 hours ago		
 main.cpp	feat: Added function to print hello world	4 minutes ago		

Como terminamos de implementar a funcionalidade, precisamos mover a issue para o board “**Waiting Acceptance**” e aguardar que alguém aprove as mudanças realizadas.





Após atualizar o repositório no Gitlab, podemos voltar na página da definição do merge request (abaixo) e após uma alguém (uma liderança) garantir que a issue foi resolvida, podemos realizar o merge dessas alterações clicando em “**Mark as ready**” e depois em “**Merge**”. Neste exemplo, apenas para sermos breves evitaremos o passar pelo ramo staging e código será mesclado diretamente com ramo develop.


Resolve "Add function to print hello world"

Overview 0 Commits 1 Changes 2

Closes #1



 Request to merge 1-add-function-to-p... into develop


Open in Web IDE Check out branch 

 No pipeline [Add the .gitlab-ci.yml file](#) to create one.

Are you adding technical debt or code vulnerabilities?
Use [CI pipelines to test your code](#) by simply adding a GitLab CI configuration file to your project. It only takes a minute to make your code more secure and robust.

[Show me how to add a pipeline](#)

 [Approve](#) Approval is optional 

 Merge ☒ Delete source branch

> 1 commit and 1 merge commit will be added to develop. [Modify merge commit](#)

No entanto, o procedimento descrito anteriormente, também é válido para o branch staging. Além disso, em uma situação crítica, supondo que tenha ocorrido uma falha de software no ambiente de produção, é preciso corrigi-la o mais rapidamente possível. Para isso, deve-se seguir o mesmo procedimento descrito anteriormente. Porém, desta vez, deve-se criar um **branch hotfix** a partir do ramo master e fazer o merge de volta no ramo master e também no ramo develop para propagar as correções.

Conventional Commits 1.0.0

Resumo

A especificação do Conventional Commits é uma convenção simples para utilizar nas mensagens de commit. Ela define um conjunto de regras para criar um histórico de commit explícito, o que facilita a criação de ferramentas automatizadas baseadas na especificação. Esta convenção se encaixa com o [SemVer](#), descrevendo os recursos, correções e modificações que quebram a compatibilidade nas mensagens de commit.

A mensagem do commit deve ser estruturada da seguinte forma:

<tipo>[escopo opcional]: <descrição>

[corpo opcional]

[rodapé(s) opcional(is)]

O commit contém os seguintes elementos estruturais, para comunicar a intenção ao utilizador da sua biblioteca:

1. **fix:** um commit do *tipo* fix soluciona um problema na sua base de código (isso se correlaciona com [PATCH](#) do versionamento semântico).
2. **feat:** um commit do *tipo* feat inclui um novo recurso na sua base de código (isso se correlaciona com [MINOR](#) do versionamento semântico).
3. **BREAKING CHANGE:** um commit que contém no rodapé opcional o texto BREAKING CHANGE:, ou contém o símbolo ! depois do tipo/escopo, introduz uma modificação que quebra a compatibilidade da API (isso se correlaciona com [MAJOR](#) do versionamento semântico). Uma BREAKING CHANGE pode fazer parte de commits de qualquer *tipo*.
4. Outros *tipos* adicionais são permitidos além de fix: e feat:, por exemplo [@commitlint/config-conventional](#) (baseado na [Convenção do Angular](#)) recomenda-se build:, chore:, ci:, docs:, style:, refactor:, perf:, test:, entre outros.
5. Outros *rodapés* diferentes de BREAKING CHANGE: <descrição> podem ser providos e seguem a convenção simular a [git trailer format](#).

Observe que esses tipos adicionais não são exigidos pela especificação do Conventional Commits e não têm efeito implícito no versionamento semântico (a menos que incluam uma

BREAKING CHANGE). Um escopo pode ser fornecido ao tipo do commit, para fornecer informações contextuais adicionais e está contido entre parênteses, por exemplo feat(parser): adiciona capacidade de interpretar arrays.

Exemplos

Mensagem de commit com descrição e modificação que quebra a compatibilidade no rodapé

feat: permitir que o objeto de configuração fornecido estenda outras configurações

BREAKING CHANGE: a chave `extends`, no arquivo de configuração, agora é utilizada para estender outro arquivo de configuração

Mensagem de commit com ! para chamar a atenção para quebra a compatibilidade

refactor!: remove suporte para Node 6

Mensagem de commit com escopo e ! para chamar a atenção para quebra a compatibilidade

refactor(execução)!: remove suporte para Node 6

Commit message with both ! and BREAKING CHANGE footer

refactor!: remove suporte para Node 6

BREAKING CHANGE: refatorar para usar recursos do JavaScript não disponíveis no Node 6.

Mensagem de commit sem corpo

docs: ortografia correta de CHANGELOG

Mensagem de commit com escopo

feat(lang): adiciona tradução para português brasileiro

Mensagem de commit de uma correção utilizando número de ticket (opcional)

fix: corrige pequenos erros de digitação no código

veja o ticket para detalhes sobre os erros de digitação corrigidos

Revisado por: Daniel Nass

Refs #133

Especificação

As palavras-chaves “DEVE” (“MUST”), “NÃO DEVE” (“MUST NOT”), “OBRIGATÓRIO” (“REQUIRED”), “DEVERÁ” (“SHALL”), “NÃO DEVERÁ” (“SHALL NOT”), “PODEM” (“SHOULD”), “NÃO PODEM” (“SHOULD NOT”), “RECOMENDADO” (“RECOMMENDED”), “PODE” (“MAY”) e “OPCIONAL” (“OPTIONAL”), nesse documento, devem ser interpretados como descrito na [RFC 2119](#).

1. A mensagem de commit DEVE ser prefixado com um tipo, que consiste em um substantivo, feat, fix, etc., seguido por um escopo OPCIONAL, símbolo OPCIONAL !, e OBRIGATÓRIO terminar com dois-pontos e um espaço.
2. O tipo feat DEVE ser usado quando um commit adiciona um novo recurso ao seu aplicativo ou biblioteca.
3. O tipo fix DEVE ser usado quando um commit representa a correção de um problema em seu aplicativo ou biblioteca.
4. Um escopo PODE ser fornecido após um tipo. Um escopo DEVE consistir em um substantivo que descreve uma seção da base de código entre parênteses, por exemplo, fix(parser): .
5. Uma descrição DEVE existir depois do espaço após o prefixo tipo/escopo. A descrição é um breve resumo das alterações de código, por exemplo, *fix: problema na interpretação do array quando uma string tem vários espaços*.
6. Um corpo de mensagem de commit mais longo PODE ser fornecido após a descrição curta, fornecendo informações contextuais adicionais sobre as alterações no código. O corpo DEVE começar depois de uma linha em branco após a descrição.
7. Um corpo de mensagem de commit é livre e PODE consistir em infinitos parágrafos separados por uma nova linha.
8. PODE(M) ser fornecidos um ou mais rodapés, uma linha em branco após o corpo. Cada rodapé DEVE consistir em um token de palavra, seguido por um separador :<espaço> ou <espaço>#, seguido por um valor de uma string (isso é inspirado pelo [git trailer convention](#)).
9. Um token de rodapé DEVE usar - no lugar de espaços em branco, por exemplo, Acked-by (isso ajuda a diferenciar a seção de rodapé de um corpo de vários parágrafos). Uma exceção é feita para BREAKING CHANGE, que PODE também ser usado como um token.
10. O valor de um rodapé PODE conter espaços e novas linhas, e a análise (parsing) DEVE terminar quando o próximo token/separador de rodapé válido for encontrado.

11. BREAKING CHANGES DEVEM ser indicadas após o tipo/escopo de uma mensagem de commit, ou como uma entrada no rodapé.
12. Se incluída como um rodapé, uma alteração de quebra DEVE consistir no texto em maiúsculas QUEBRAR ALTERAÇÃO, seguido por dois pontos, espaço e descrição, por exemplo, *BREAKING CHANGE: as variáveis de ambiente agora têm precedência sobre os arquivos de configuração.*
13. Se incluído no prefixo de tipo/escopo, as BREAKING CHANGES DEVEM ser indicadas por um ! imediatamente antes de :. Se o símbolo ! for usado, BREAKING CHANGE: PODE ser omitido da seção de rodapé, e a descrição da mensagem de commit DEVE ser usada para descrever a BREAKING CHANGE.
14. Tipos diferentes de feat e fix PODEM ser usados em suas mensagens de commit, por exemplo, *docs: documentos de referência atualizados*
15. As unidades de informação que compõem o Conventional Commits NÃO DEVEM ser tratadas com distinção entre maiúsculas e minúsculas pelos implementadores, com exceção de BREAKING CHANGE que DEVE ser maiúscula.
16. BREAKING-CHANGE DEVE ser sinônimo de BREAKING CHANGE, quando usado como um token em um rodapé.

Porque utilizar Conventional Commits

- Criação automatizada de CHANGELOGs.
- Determinar automaticamente alterações no versionamento semântico (com base nos tipos de commits).
- Comunicar a natureza das mudanças para colegas de equipe, o público e outras partes interessadas.
- Disparar processos de build e deploy.
- Facilitar a contribuição de outras pessoas em seus projetos, permitindo que eles explorem um histórico de commits melhor estruturados.

Perguntas Frequentes

Como devo lidar com mensagens de commit na fase inicial de desenvolvimento?

Recomendamos que você prossiga como se já tivesse lançado o produto. Normalmente *alguém*, mesmo que seja seus colegas desenvolvedores, está usando seu software. Eles vão querer saber o que foi corrigido, novas features, breaking changes etc.

Os tipos no título das mensagens commit são maiúsculos ou minúsculos?

Qualquer opção pode ser usada, mas é melhor ser consistente.

O que eu faço se o commit estiver de acordo com mais de um dos tipos?

Volte e faça vários commits sempre que possível. Parte do benefício do Conventional Commits é a capacidade de nos levar a fazer commits e PRs de forma mais organizada.

Isso não desencoraja o desenvolvimento rápido e a iteração rápida?

Desencoraja a movimentação rápida de forma desorganizada. Ele ajuda você a ser capaz de se mover rapidamente a longo prazo em vários projetos com múltiplos colaboradores.

O Conventional Commits leva os desenvolvedores a limitar o tipo de commits que eles fazem porque estarão pensando nos tipos fornecidos?

O Conventional Commits nos encorajam a fazer mais commits de tipos específicos, por exemplo correções. Além disso, a flexibilidade do Conventional Commits permite que sua equipe crie seus próprios tipos e altere ao longo do tempo.

Qual a relação com o SemVer?

Commits do tipo fix devem ser enviados para releases PATCH. Commits do tipo feat devem ser enviados para releases MINOR. Commits com BREAKING CHANGE nas mensagens, independentemente do tipo, devem ser enviados para releases MAJOR.

Como devo versionar minhas extensões utilizando a especificação do Conventional Commits Specification, e.g. @jameswomack/conventional-commit-spec?

Recomendamos utilizar o [SemVer](#) para liberar suas próprias extensões para esta especificação (e incentivamos você criar essas extensões!)

O que eu faço se acidentalmente usar o tipo errado de commit?

Quando você usou um tipo da especificação, mas não do tipo correto, por exemplo fix em vez de feat

Antes do merge ou release com o erro, recomendamos o uso de git rebase -i para editar o histórico do commit. Após o release, a limpeza será diferente de acordo com as ferramentas e processos que você utiliza.

Quando você usou um tipo que *não* é da especificação, por exemplo feat em vez de feat

Na pior das hipóteses, não é o fim do mundo se um commit não atender à especificação do Conventional Commits. Significa apenas que o commit será ignorado por ferramentas baseadas nessa especificação.

Todos os meus colaboradores precisam usar a especificação do Conventional Commits?

Não! Se você usar um workflow de git baseado em squash, os mantenedores poderão limpar as mensagens de commit à medida que forem fazendo novos merges, não adicionando carga de trabalho aos committers casuais. Um workflow comum para isso é fazer com que o git faça squash dos commits automaticamente de um pull request e apresente um formulário para o mantenedor inserir a mensagem do commit apropriada para o merge.

Como o Conventional Commits trata os commits de reversão?

Reverter código pode ser complicado: você está revertendo vários commits? se você reverter um recurso, a próxima versão deve ser um patch?

O Conventional Commits não o força a definir um comportamento de reversão. Em vez disso, deixamos isso para que os autores de ferramentas usem a flexibilidade de *tipos* e *rodapés* para desenvolver sua própria lógica para lidar com reversões.

Uma recomendação é usar o tipo revert e um rodapé que referencia os SHAs de commit que estão sendo revertidos:

revert: nunca mais falaremos do incidente do miojo

Refs: 676104e, a215868