Git Flow – Uma forma legal de organizar repositórios git

Uma dúvida muito comum a quem começa a usar o Git de maneira mais ativa é como organizar as branches, afinal, são muitos os problemas que um projeto pode enfrentar: De bugs urgentes que devem ser corrigidos, a criação de inúmeras features em conjunto com *releases* agrupando os deploys relativos a essas features. Mas..como organizar tudo? Pensando nisso é que foi criado o Git Flow, um modelo de organização de branches criado por Vincent Driessen que mais tarde se tornou uma excelente extensão ao Git, permitindo seu uso de forma fácil com qualquer repositório git, e é sobre isso que vou falar hoje.

Antes de começar a falar mais detalhadamente sobre o **Git Flow**, quero chamar atenção para o fato de que o **Git Flow** não é o único modelo de organização de branches: Existem outros por aí, e inclusive existem empresas que adotam modelos de organização próprios, mas, disso tudo, um fato é que o **Git Flow** parece ser o mais conhecido e usado, se encaixando muito bem numa plenitude de situações, e sendo muito bom para trabalhos em equipe (por experiência própria). Se você conhecer alguma alternativa que funciona bem, mande nos comentários!

Agora, continuando as explicações sobre o **Git Flow**, como explicado no primeiro parágrafo, trata-se de um modelo de organização de branches desenvolvido especialmente para o git. Por ser primariamente um modelo de organização de branches, isso significa que o Git Flow estabelece algumas regras de **nomenclaturas** para tipos de branches enquanto, ao mesmo tempo, define o que **cada tipo de branch faz.** Para referência, segue uma lista dos tipos de branches definidos pelo Git Flow e suas respectivas descrições:

- **Branch master** – É a branch que contém código em nível de **produção**, ou seja, o código mais maduro existente na sua aplicação. Todo o código novo produzido eventualmente é juntado com a branch **master**, em algum momento do desenvolvimento;

- **Branch develop** – É a branch que contém código em nível preparatório para o próximo deploy. Ou seja, quando features são terminadas, elas são juntadas com a branch **develop**, testadas (em conjunto, no caso de mais de uma feature), e somente depois as atualizações da branch **develop** passam por mais um processo para então ser juntadas com a branch **master**;
- **Branches feature/*** – São branches no qual são desenvolvidos recursos novos para o projeto em questão. Essas branches tem por convenção nome começando com **feature/** (exemplo: **feature/new-layout**) e são criadas a partir da branch **develop** (pois um recurso pode depender diretamente de outro recurso em algumas situações), e, ao final, são juntadas com a branch **develop**;
- **Branches hotfix/*** – São branches no qual são realizadas correções de bugs críticos encontrados em ambiente de produção, e que por isso são criadas a partir da branch **master**, e são juntadas diretamente com a branch **master** e com a branch **develop** (pois os próximos deploys também devem receber correções de bugs críticos, certo?). Por convenção, essas branches tem o nome começando com **hotfix/** e terminando com o próximo sub-número de versão (exemplo: **hotfix/2.31.1**), normalmente seguindo as regras de algum padrão de versionamento, como o padrão do versionamento semântico, que abordei neste post;
- **Branches release/*** – São branches com um nível de confiança maior do que a branch **develop**, e que se encontram em nível de preparação para ser juntada com a branch **master** e com a branch **develop** (para caso tenha ocorrido alguma correção de bug na branch **release/*** em questão). Note que, nessas branches, bugs encontrados durante os testes das *features* que vão para produção podem ser corrigidos mais tranquilamente, **antes** de irem efetivamente para produção. Por convenção, essas branches tem o nome começando com **release/** e terminando com o número da próxima versão do software (seguindo o exemplo do *hotfix*, dado acima, seria algo como **release/2.32.0**), normalmente seguindo as regras do versionamento semântico, como falado acima;

Um aspecto interessante do Git Flow é que, quando você mistura uma branch **release/** *ou* ***hotfix/*** com a branch **master**, ele automaticamente cria **git tags** correspondentes aos merge commits da mistura, facilitando o trabalho de, por exemplo, mudar para uma versão mais antiga, e organizando todo o trabalho.

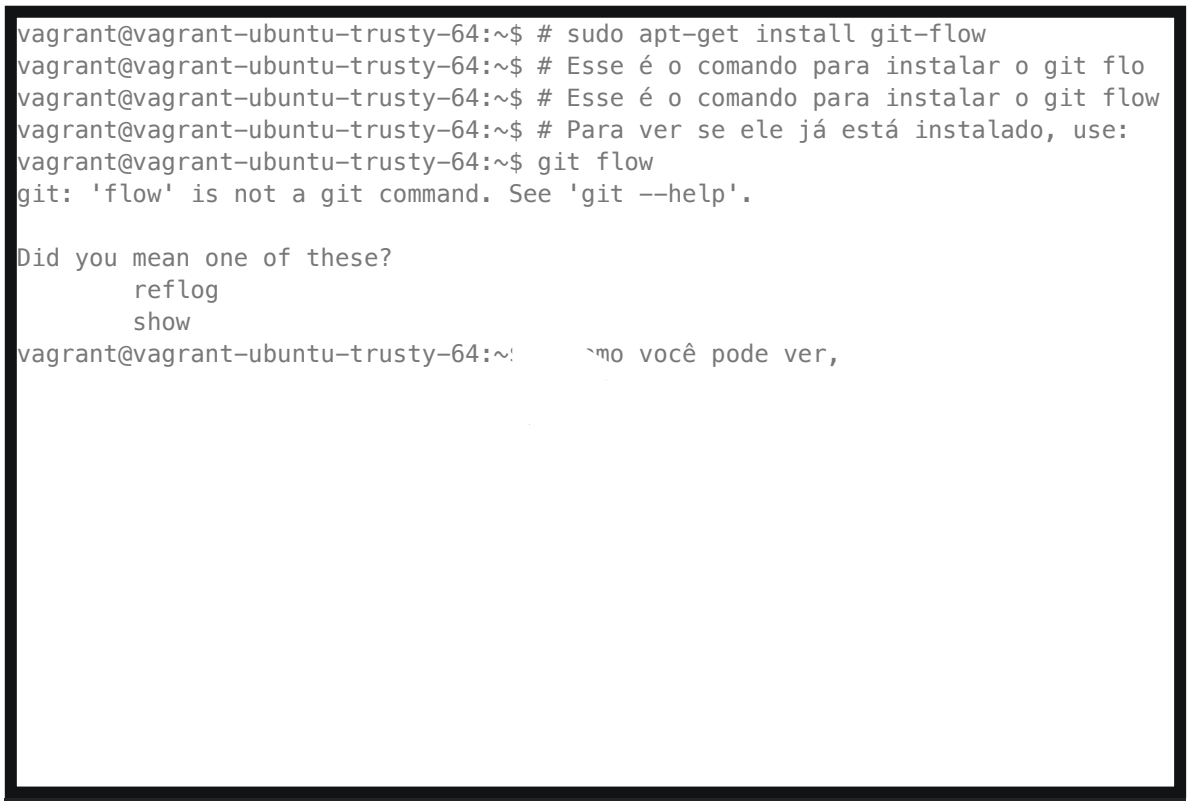
Mas…o que são **git tags**? Simples, se lembra do tutorial básico de git, no qual cada commit assume um *hash* único, ao estilo *bdbb064f6db74509148a8aae67177931413925a5*? Então, **git tags** são como "atalhos" que possuem um nome mais amigável definido por você para um commit específico no repositório, e que podem ser acessadas facilmente usando **git checkout** . Em outro tutorial ensinarei melhor como usá-las, pois não vem ao ponto aqui (visto que as extensões do **git flow** as criam automaticamente).

Enfim, continuando, agora que você já sabe um pouco dos conceitos do Git Flow e como eles se aplicam (se você ficou com alguma dúvida, inclusive, poste sua dúvida nos comentários que eu farei questão de a responder!), vamos aplicar o que você aprendeu na prática? Partiu para o tutorial, então. 😉

Tutorial básico de git flow

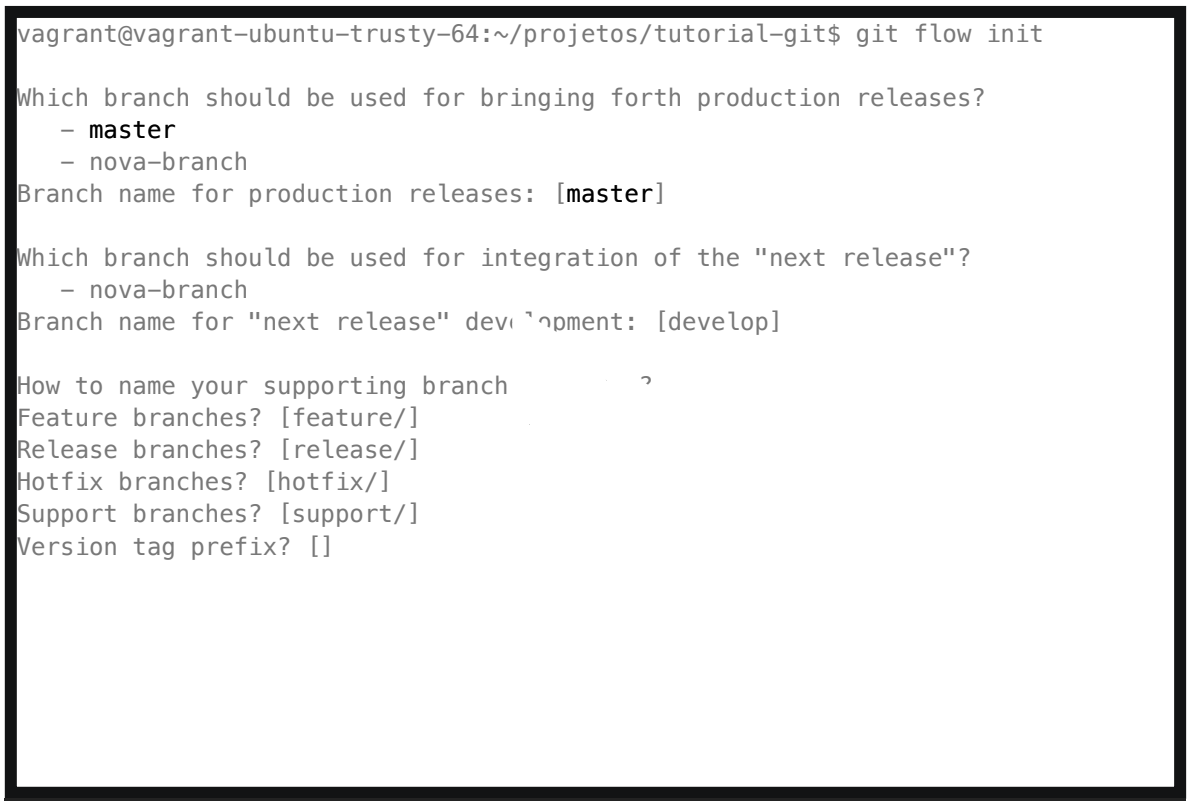
Para esse tutorial básico de git flow, vou assumir que você está com o repositório criado no tutorial básico de git, que vai servir inclusive para mostrar que você, a qualquer momento, pode adaptar **qualquer repositório git** para começar a usar o Git Flow. Segue o *passo-a-passo*:

1) Primeiro, precisamos ter o Git Flow instalado na máquina. Como se trata de uma **extensão** ao Git, ele não vem instalado com o Git por padrão e para isso precisamos instalá-lo manualmente em sua máquina (mesmo se for a máquina do Vagrant usado no primeiro tutorial, pois eu me esqueci de instalar o Git Flow..). Segue o link para a documentação sobre como instalá-lo. E segue a gravação da instalação do Git Flow sendo feita na máquina virtual Vagrant fornecida no post do tutorial básico de git:



2) Agora, na pasta do repositório criado no tutorial básico de git, execute o comando "**git flow init**". Esse comando vai fazer algumas perguntas, como o prefixo desejado para as branches **feature/**, ***release/*** e **hotfix/***, e também o nome das branches

master e **develop**, que são coisas que o Git Flow permite configurar mas que recomendo fortemente que deixe com os valores padrão. Ou seja, nesse caso, basta apertar **Enter** para todas as perguntas, deixando que o Git Flow seja inicializado com os valores padrão e recomendados. Veja:



Note que o Git Flow pergunta também o prefixo para branches **support/**. *Bom, embora disponíveis, branches* ***support/*** ainda são muito experimentais e portanto não serão abordadas nesse tutorial.

Note que, durante a execução do comando, o Git Flow *automagicamente* criará a branch **develop** e fará **git checkout** automático para esta branch.

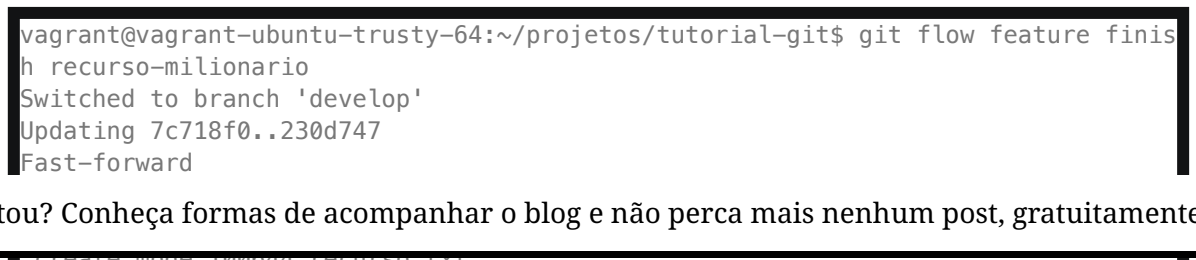
3) Agora, vamos criar uma nova **feature branch**? Para fazer isso, execute o comando "**git flow feature start recurso-milionario**" para criar uma nova **feature branch** chamada **recurso-milionario**, com o nome **feature/recurso-milionario**. Veja o comando em execução:

4) Nessa nova branch, crie um arquivo **recurso.txt** com o conteúdo **"Este é o melhor recurso criado desde sempre!"**. Como feito no primeiro tutorial, execute os comandos "**git add recurso.txt**" e "**git commit -m 'Finished feature'**" para adicionar e commitar o arquivo em questão na feature branch **recurso-milionario** (se você tiver alguma dúvida relacionada a esses comandos, veja o tutorial básico de git). Veja:

00:00

Powered by asciinema

5) Com o commit feito, podemos finalmente juntá-lo a branch **develop**. Para fazer isso, execute o comando "**git flow feature finish recurso-milionario**". Como visto abaixo:

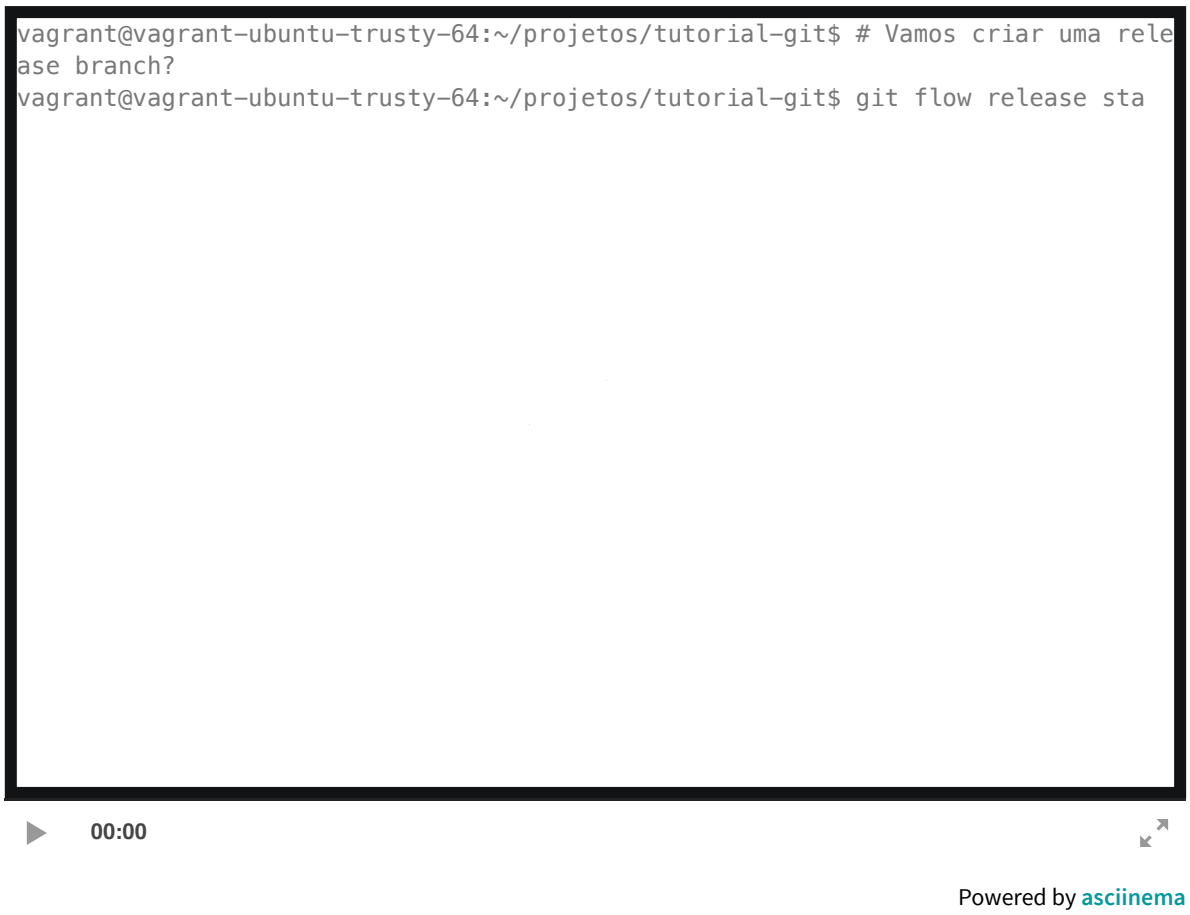


00:00

Powered by asciinema

Como você pode ver, a branch **feature/recurso-milionario** foi correspondentemente integrada à branch **develop** e o git flow fez checkout automático para a branch **develop**, te mostrando todos os passos feitos.

6) Agora que temos um recurso milionário na branch develop (risos), vamos criar uma *release branch* para poder enfim publicar a atualização na **branch master**? Para fazer isso, execute o comando "**git flow release start 0.1.0**". Veja:



no arquivo **recurso.txt**, modificando a frase de "**Este é o melhor recurso criado desde sempre!**" para "**Este talvez seja o melhor recurso criado desde sempre!**", adicionando a mudança à area de estágio do git usando "**git add recurso.txt**" e depois comittar a mudança na release branch usando "**git commit -m 'Little bug-fix in feature'**". Veja:

Note que só estamos fazendo este commit para lhe mostrar que sim, mudanças podem ser feitas antes da branch ser juntada com a branch **master**. Mas quero lhe lembrar que isso **não é necessário**, e que em muitos casos a release branch existe apenas por…existir, sendo imediatamente juntada com a branch **master,** sem mudanças adicionais.

8) Com a mudança registrada, vamos enfim juntar a **release branch** 0.1.0 com a branch **master** e a branch **develop**, para isso, vamos usar o comando "**git flow**

as branches **master** e **develop**. Veja:

Como você pode ver acima, o Git Flow abre o editor de texto três vezes:

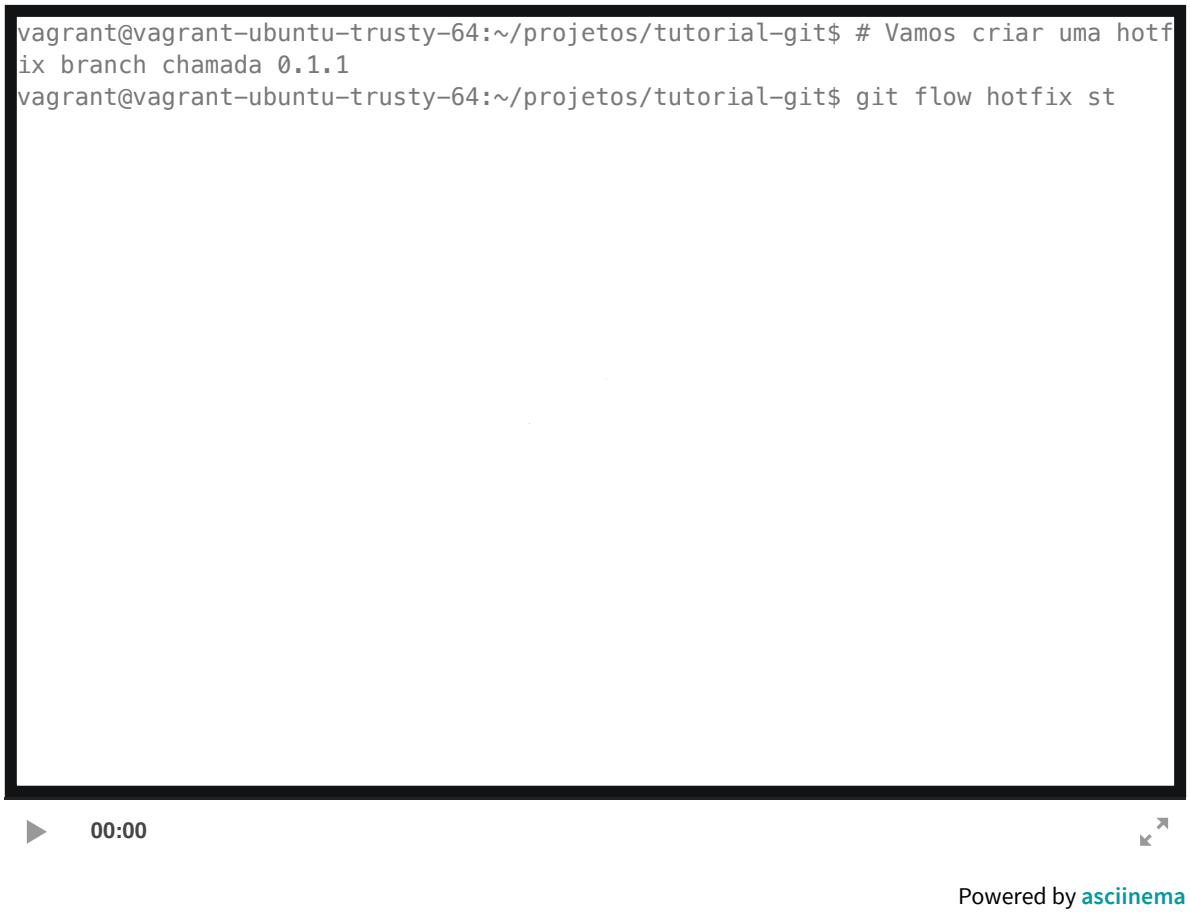
- Uma para você editar o texto do **merge commit** relacionado ao *merge* entre a release branch **0.1.0** e a branch **master;**
- Um para a descrição da tag **0.1.0**, que será criada pelo Git Flow para facilitar mudanças de versão no software;
- Uma para você editar o texto do **merge commit** relacionado ao *merge* entre a branch **master** e a branch **develop**;

Na primeira e na terceira vez, sugiro para que você deixe o texto padrão deixado pelo Git. Apenas na segunda vez, quando ele pede para você colocar uma descrição para a tag, é recomendado (e se não me engano, obrigatório) que você descreva exatamente o que foi adicionado e/ou modificado naquela versão (acredite, serve para referência futura).

9) Agora que a branch **master** foi atualizada com o novo recurso, suponhamos que foi encontrado um bug ultra-mega-hiper-critico na aplicação (coincidentemente, nesse exemplo, suponhamos que foi no mesmo recurso recém adicionado) e que ele é tão grave que está afetando o uso por todos os usuários da aplicação e por isso precisa ser corrigido com urgência máxima.

Para corrigir esse bug critico, vamos criar um **hotfix** usando o comando "**git flow hotfix start 0.1.1**", que criará uma **hotfix branch** chamada **0.1.1** que resolve um

problema encontrado no release **0.1.0** (entendeu porque faz sentido usar versionamento semântico nos nomes de branches usando nas releases branches e hotfixs branches?). Veja:



10) Agora que temos a nossa **hotfix branch 0.1.1** criada, vamos editar o arquivo **recurso.txt** com a correção que queremos aplicar, substituindo a frase "**Este talvez seja o melhor recurso criado desde sempre!"** para "**Este talvez não seja o melhor recurso criado desde sempre! Mas é um dos mais legais!"** no arquivo

"**git commit -m 'Little hotfix in a feature'**". Veja:

11) Com o “bug” corrigido e comittado, podemos agora **finalizar** nossa **hotfix branch** e com isso juntá-la à branch **master** e à branch **develop**. Para fazer isso, basta usar o comando “**git flow hotfix finish 0.1.1**“, que fará todas essas tarefas por nós e ainda criará uma tag para marcar a correção. Veja:

00:00

Powered by asciinema

Novamente, como no caso do **git flow release finish**, estudado acima, o git flow abre o editor três vezes: Uma para editar o **merge commit** para o merge com a branch **master**, outra para editar a descrição da tag que será criada pelo Git Flow, e outra para o **merge commit** para o merge da branch **master** com a branch **develop**. Novamente, como recomendado acima, sugiro que você não edite o texto existente quando o Git Flow abrir o editor pela primeira e pela terceira vez, mas que acrescente uma boa descrição do que foi modificado quando o editor for aberto na segunda vez (ou seja, quando for para configurar a descrição da tag que o Git Flow

Com isso, acho que já dá para ter uma bela introdução do que o Git Flow é capaz de criar. Na minha opinião, acho que esse modelo de organização de branches é ideal para trabalhar com projetos em equipe, pois permite que cada membro da equipe trabalhe em cada *feature* branch com maestria e ainda resolva bugs importantes quando eles forem encontrados.

Vale lembrar também que, neste tutorial, foram abordados apenas os comandos mais importantes que o Git Flow fornece, e que, assim como qualquer outra branch criada pelo Git, eles podem ser sincronizados livremente com sua equipe usando os sistemas de hospedagem de repositórios que abordamos no post anterior, sem precisar de mais nenhum suporte especial por parte desses sistemas. Maneiro, né?

Ah, caso tenha ficado alguma dúvida a respeito do Git Flow, sugiro a leitura do post que descreveu o modelo pela primeira vez, e também o resumo de dicas sobre o que cada comando faz criado por Daniel Kummer, que pode ser bem interessante para ajudar à fixar o que cada comando faz, por exemplo.

No mais, não se esqueça de deixar o seu comentário com sua opinião, critica, dúvida ou sugestão em relação ao post , e também de acompanhar o blog para não perder mais nenhuma dica sobre desenvolvimento e tópicos relacionados, pois apenas com sua ajuda e apoio que poderei trabalhar ainda mais em posts que abordam assuntos importantes para o desenvolvedor moderno, como esse, sobre o Git Flow.

Abaixo estão os links sobre a ferramenta:

- **Repositório do Código Fonte no Github:** http://github.com/nvie/gitflow

RELACIONADO

Jimple - Container de Injeção de Dependências desenvolvido em ES6
Em aplicações simples, é normal você não precisar lidar

Gostou? Conheça formas de acompanhar o blog e não perca mais nenhum post, gratuitamente!

versões distribuído
Em "Ferramentas"

exemplo, para fazer um "Hello
Em "Javascript"

Ou como versionar software
Em "Conceitos"

22/01/2016 Fernando Ferramentas desenvolvimento, ferramentas, git, utilidades