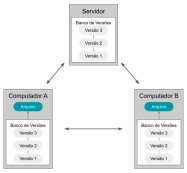
Git e GitHub

É importante ter sempre um código que funcione, mas que seja fácil de fazer mudanças. Essa facilidade permite consertarmos bugs, ou adicionarmos novas features ao nosso projeto. Assim, é comum uma equipe escrever diversas versões do código, cada uma ajustando um ponto da anterior. Isso gera duas dificuldades: como fazer para controlar todas as mudanças e permitir que cada pessoa da equipe ataque diferentes tarefas em paralelo?

O **git** é uma ferramenta criada com o intuito de facilitar essa colaboração e versionamento de códigos. Ele permite o controle e monitoramento de todas as mudanças realizadas, mostrando também quem realizou as mudanças e quando.

Criado por Linus Torvalds, o git tinha o propósito de apoiar no desenvolvimento do motor do Linux [2], que é um sistema operacional com código fonte aberto. Esse código fonte recebe contribuições de milhares de pessoas, e por isso precisava de ferramentas que apoiassem nas dificuldades organizacionais envolvidas.

O **git** ajuda com essas dificuldades da seguinte forma: a base de código fica disponível em um servidor remoto, e cada pessoa que for alterar algum arquivo pode baixar essa base para sua máquina local, junto com todo o controle das mudanças realizadas. Diz-se então que temos um *versionamento distribuído* [1].



(**Fonte:** tradução livre da imagem original, disponível no livro Pro Git [1])

Este servidor remoto, usado para hospedagem do código, pode ser próprio da empresa que esteja fazendo o desenvolvimento, mas o mais comum é utilizar o site **GitHub**, da Microsoft.

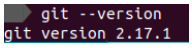
O GitHub nos dá espaço para hospedar os arquivos que usamos em projetos de desenvolvimento, e possui uma interface gráfica simples para visualizar todos os controles realizados pelo git. É importante saber, contudo, que existem também outros sites com a mesma finalidade, como o Bitbucket (da Atlassian), e o GitLab (da GitLab Inc.), e embora cada um tenha algumas características diferentes, todos permitem essa visualização mais intuitiva dos controles do git.

Neste texto, iremos sempre assumir que estamos usando o GitHub, para facilitar as explicações. Porém, os comandos do git serão os mesmos independentes do site no qual o código está hospedado.

1. Começando com o git

O git é instalado como qualquer outro programa, e funciona tanto no Windows, quanto no Linux e no MacOS. Um guia com o passo a passo de como instalar o git em cada um desses sistemas operacionais se encontra no livro Pro Git, disponível online sem custo [1], com versão em português.

Depois de instalado, podemos usar o git via linha de comando em um terminal (no caso do Windows, é possível usar o command prompt, o powershell ou o git bash). Para confirmar que o git está disponível para uso, basta executar o comando `git --version`. Se tudo estiver funcionando bem, aparecerá uma saída no terminal com a versão do git que está instalada.



(Neste caso, o git instalado foi a versão 2.17.1. Esta versão já está bem defasada, visto que a versão mais atual do Git é a 2.37.1.) (**Fonte:** Imagem original Lets Code)

Com o git instalado, uma das primeiras coisas a ser feita é configurá-lo. Isso é importante para que todas as alterações que você realize em um código controlado pelo git sejam identificadas como tendo sido feitas por você. Para isso, é preciso executar 2 comandos:

```
git config --global user.name "Seu Nome"
git config --global user.email seuemail@exemplo.com
```

No primeiro comando, você irá colocar seu nome (em geral, nome de usuário da sua conta do GitHub, se for o caso), e no segundo comando, você irá colocar seu email. Esses comandos criam um arquivo `.gitconfig` na sua pasta principal (pode ser `C:\` no caso do Windows, ou `~` no caso do Linux).

Depois disso, já podemos começar a usar o git.

Se estivermos usando o GitHub, é possível esconder o e-mail utilizado, de forma que ele não fique público. Basta seguir o passo a passo do site para ver como isto é feito.

Ao utilizar o git, precisaremos utilizar um editor de texto para alterar diferentes mensagens e comunicados do controle de versão. Para escolhermos o editor a ser usado como padrão, nós usamos o comando:

```
git config --global core.editor [Caminho do executável do editor desejado]
```

Como exemplo, se o nosso editor for o notepad++, e ele estiver na pasta "C:\users\meu_usuario\notepad++.exe", nós então escrevemos:

```
git config --global core.editor "C:\users\meu_usuario\notepad++.exe"
```

Atualmente, existem comandos específicos para quando quisermos usar o visual studio code (VSCode) ou o notepad. Para o VSCode:

```
git config --global core.editor "code -w"
```

Para o notepad:

```
git config --global core.editor "notepad -w"
```

A opção `-w` é utilizada para avisar ao git que espere até fecharmos a janela do editor.

No caso do Linux e do MacOs, se existir um comando no terminal para abrir o editor desejado, esse comando pode ser usado. Por exemplo, no Linux existe o comando `vim`, que abre o editor de mesmo nome. Neste caso, usamos:

```
git config --global core.editor vim
```

Uma última configuração que se costuma recomendar é executar o comando:

```
git config --global init.defaultBranch main
```

Isso muda o nome padrão do ramo principal de versionamento de "master" para "main", quando vamos criar um novo projeto git. Esse nome é muito mais utilizado hoje em dia, sendo preferível utilizá-lo sempre que possível.

Contudo, é importante lembrar que essa configuração só funciona a partir da versão 2.28 do git [1].

1.1. Inicializando um repositório

Cada projeto tem o seu repositório, que é como é chamada a *pasta* que guarda os arquivos de código. Existem duas formas de inicializar um repositório: *localmente* ou *importando de um servidor*. Um repositório em um servidor é chamado de **repositório remoto**.

Repositório Local

Para iniciarmos um repositório localmente, criamos uma pasta com o nome desejado e usamos o comando abaixo dentro da pasta.

```
git init .
```

Podemos ver se a pasta virou um repositório pela existência de uma nova subpasta chamada `.git`, que é onde o git guarda todos os arquivos necessários para controle das mudanças do código.

Se quisermos colocar esse repositório local em um servidor remoto, precisamos primeiro criar um projeto no servidor e depois sincronizar com o nosso projeto local. Fazemos isso copiando o link do projeto disponibilizado pelo servidor. Essa URL pode ser obtida pelo repositório no GitHub, ao clicar em *Clone or Download*.

Usamos então os comandos abaixo [3]. Eles assumem que fizemos a configuração da `main` anteriormente. Caso contrário, utilize `master`.

```
git remote add origin [link do servidor]
git push -u origin main
```

Por exemplo, se nosso usuário no GitHub for `my_user`, e nós criamos o projeto `projeto_exemplo_01`, teremos a URL `git@github.com:my_user/projeto_exemplo_01.git` (ou `https://github.com/my_user/projeto_exemplo_01.git`, cuja diferença será explicada mais adiante no texto). O comando ficará:

```
git remote add origin git@github.com:my_user/projeto_exemplo_01.git
git push -u origin main
```

Repositório Remoto

Para importar um repositório remoto, nós usamos o comando `git clone` com a URL descrita no fim da seção anterior.

```
git clone git@github.com:my_user/projeto_exemplo_01.git
```

O git irá criar uma pasta `projeto_exemplo_01`, onde ficará o repositório. É possível também definir outro nome para essa pasta, usando um segundo argumento para o comando. Se quisermos que ela se chame `projeto_pessoal`, por exemplo, escrevemos:

```
git clone git@github.com:my_user/projeto_exemplo_01.git projeto_pessoal
```

Seja ao importar um repositório remoto, ou ao sincronizar o nosso local com um projeto remoto, é necessário autenticar para o servidor que somos nós rodando o comando. Essa autenticação pode ser feita usando dois protocolos diferentes: HTTPS ou SSH.

Autenticação HTTPS

O HTTPS (sigla para *Hypertext Transfer Protocol Secure*) é um protocolo web, igual ao que usamos normalmente em browsers. Por exemplo, ao abrir o google, nós colocamos a URL `https://www.google.com`, embora em geral o browser já complete automaticamente a parte "https".

Quando usarmos HTTPS para enviar mudanças para o servidor, ele irá pedir o usuário e a senha toda vez. Além disso, o link do repositório que discutimos antes ficaria `https://github.com/my_user/projeto_exemplo_01.git`

Uma vantagem do HTTPS é que não é necessário configurar nada previamente no sistema. Basta emitirmos no GitHub um token de acesso pessoal (PAT - personal access token), e usarmos ele como senha, como descrito neste guia.

Uma desvantagem, porém, é que será necessário digitar seu usuário e senha (ou seu PAT) toda vez que enviar mudanças locais para o repositório online. Dependendo da senha que você usa na sua conta, também pode ser fácil para um(a) hacker descobrí-la, e assim ter acesso aos seus projetos.

Autenticação SSH

O SSH (sigla para *Secure Shell Protocol*) é um protocolo que funciona pela troca de chaves criptográficas entre a sua máquina e o servidor. Assim, o servidor checa essas chaves para autenticar que é você mesmo enviando as mudanças, e assim aceitar ou não todas as mudanças de código que enviarmos a ele.

Para usar SSH, é necessário gerar uma chave SSH (ou usar uma existente) e adicionar essa chave à sua conta no GitHub. Esse guia explica como fazer isso e testar se a configuração está correta.

A URL do repositório que discutimos antes ficaria
`git@github.com:my_user/projeto_exemplo_01.git`

Usando SSH, não será necessário digitar as informações da sua conta toda vez que enviar mudanças para o servidor. Além disso, a chave utilizada é bem mais complicada que senhas, em geral, sendo menos suscetível a ataques hacker. O problema é que ela ficará no formato de um arquivo em seu computador, então cuidado para não dar acesso a ela para ninguém.

Outra complicação é que é necessário configurar algumas coisas no computador antes de usar SSH, embora em geral essa configuração normalmente só é feita uma vez (a não ser que você precise de novas chaves criptográficas).

2. Trabalhando com o repositório

Agora que você já tem uma cópia de um repositório remoto no seu computador, já podemos começar a fazer mudanças no código que serão controladas pelo git.

Sempre que vamos usar um repositório que já existe há algum tempo, e onde trabalhamos junto com outras pessoas, é comum sempre executarmos o comando

```
git pull
```

antes de começar qualquer trabalho. Esse comando baixa para a nossa máquina local as últimas alterações realizadas no repositório remoto.

Sem isso, o nosso repositório local fica rapidamente desatualizado. Assim, quando formos fazer quaisquer mudanças no código, podemos estar trabalhando em um programa já defasado, gerando conflitos.

Com o repositório local atualizado com as mudanças dos nossos pares, podemos então trabalhar em cima do código que estamos desenvolvendo. Em alguns pontos ao longo do seu desenvolvimento, você pode entender que as mudanças que você fez já poderiam ser versionadas. Em geral é porque você já corrigiu algum bug, ou adicionou alguma feature de interesse ao programa.

2.1. O que são commits

Nestes momentos, queremos então fechar um "pacote" de mudanças que enviaremos para o GitHub. Para isso, vamos ver como é o fluxo do git ao controlar essas mudanças.

No seu repositório local, você tem os arquivos nos quais se encontra o código em desenvolvimento. Dizemos que esse repositório é o seu *diretório de trabalho*. Quando você tiver alterado alguns arquivos, e quiser adicioná-los ao controle de mudanças do git, você os coloca na área de *staging*, através do comando `git add <arquivo>` (`<arquivo>` vai ser substituído pelo nome do arquivo alterado, junto com o caminho relativo de onde você está no terminal até o arquivo).

Se por algum motivo desejarmos adicionar todos os arquivos da nossa pasta os quais fizemos alterações, usamos o comando `git add .`.

Quando estivermos satisfeitos com todas as mudanças que colocamos na *staging*, e quisermos fechar o nosso "pacote" de mudanças, nós usamos o comando `git commit`.

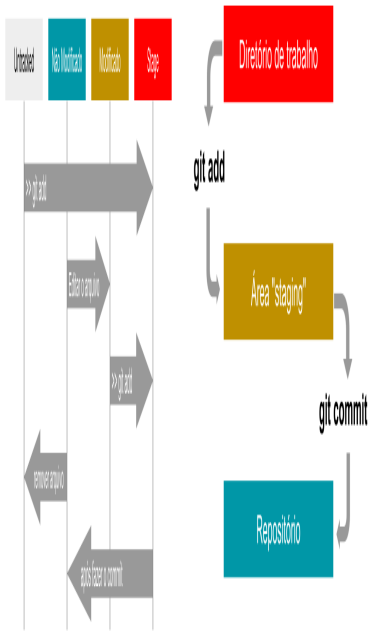
Neste caso, o git irá abrir no seu editor de texto uma janela para que você coloque comentários explicando as mudanças desse "pacote" para os seus pares. Quando você fechar o editor, o git vai salvar esses comentários e deixar pronto o seu *commit*. Um *commit* é um pacote de mudanças, e possui uma identificação própria, além do comentário adicionado pela pessoa que o fez.

(Exemplo de commit. O nome do autor foi apagado por questões de privacidade.)
(**Fonte:** Imagem original Ada)

Seguindo esse caminho, o commit terá um título (a primeira linha da descrição), e o resto (segunda linha pra baixo) será compreendido como um texto resumido. Porém, podemos querer fazer um commit rápido apenas com uma frase explicativa curta. Neste caso, é possível usar o comando de *commit* com a flag `-m` e uma mensagem entre aspas, como por exemplo `git commit -m "Mensagem a ser enviada"`.

Esses commits estão apenas em nossa máquina local por enquanto. Porém, no fim do dia, queremos enviar todos para o repositório remoto, para compartilhar com os outros. Neste caso, usamos o comando `git push` para enviar nossas alterações para o servidor.

Todo o fluxo que discutimos acima são representados nas imagens abaixo. Arquivos novos nascem no estado "untracked". Isso significa que o Git não está acompanhando as mudanças realizadas neles ainda.



(Fluxo dos arquivos, dentro do controle do Git.)
(**Fonte:** tradução livre da imagem original, disponível no livro Pro Git [1])

Resumindo, o fluxo desde as mudanças que realizamos até o envio dos *commits* para o repositório remoto pode ser visto como a sequência de comandos abaixo.

```
git add .
git commit -m "Mensagem"
git push
```

Se quisermos ver o histórico de todos os commits já realizados no passado, usamos o comando `git log`.

Contudo, o mais comum é que nos percamos um pouco nas mudanças que estamos fazendo. Assim, para ver todos os arquivos que o Git acompanha e que nós alterados, usamos o comando `git status`. Este comando mostra um resumo de quais arquivos estão em quais das situações descritas acima.

No caso de arquivos que estejam na situação de "modificados", usamos o comando `git diff <arquivo>` para ver, linha a linha, quais modificações foram feitas.

Ele mostra as linhas com um código de cores, sendo vermelho para a versão "pré-mudança" daquela linha, e verde para a versão nova da linha. Para quem é daltônico e não consegue diferenciar o vermelho do verde, é possível alterar essas cores, como vemos neste link do stack overflow.

Também podemos ver as mudanças realizadas apenas nos arquivos da área staging, usando `git diff --cached` ou `git diff --staged`. Por fim, é possível até ver as mudanças realizadas em um dado arquivo entre dois commits. Se quisermos ver o arquivo `<arquivo>` entre o commit `<id01>` e o commit `<id02>`, usamos o comando `git diff <id01> <id02> <arquivo>`.

A identificação (o ID) dos commits é um código alfanumérico (nesse caso, uma *hash*) que aparece no topo do commit. Como exemplo, no commit de exemplo que vimos duas imagens acima, o ID seria *3c2cc0a602233dbf3f8d1e5461725efc2f00481d*.

Se deletarmos um arquivo que estava sendo rastreado pelo Git, após deletar o arquivo é necessário utilizar o comando `git rm <arquivo>` para avisar ao Git. Neste caso, ele irá adicionar na área de *staging* a informação de que aquele arquivo foi deletado.

Além disso, quando renomeamos ou movemos um arquivo, o Git enxerga como sendo um novo arquivo criado, e o antigo sendo deletado. Para que ele entenda corretamente que apenas renomeamos ou movemos, é importante colocar no mesmo commit a deleção do arquivo antigo, e a criação do novo.

Por fim, também podemos querer voltar atrás no nosso trabalho.

Se errarmos a mensagem do nosso último commit, ou até esquecermos de adicionar algum arquivo, o comando `git commit --ammend` nos permite alterar esse commit. O comando irá adicionar os novos arquivos que estiverem na área de staging ao commit, e vai nos permitir alterar a mensagem do commit.

Para tirar um arquivo da área de staging, usamos o comando `git reset HEAD <arquivo>`. Se o arquivo está sendo rastreado e aparece como "modificado" (sem estar na área de staging), e nós queremos desfazer essas modificações, usamos o comando `git checkout -- <arquivo>`.

Embora seja difícil lembrar esses comandos no início, o Git nos ajuda relembrando de ambos os comandos sempre que usarmos `git status`.

(**Fonte:** imagem original Ada)

Por fim, pode ser que a gente deseje voltar atrás na versão do código inteiro (chamamos isso de *rollback*). Pode ser porque o código está com um bug de muito impacto, ou por outros problemas que podem levar um tempo para consertar.

Neste caso, nós usamos o comando `git reset <id03>` para fazer um rollback para o commit `<id03>`. Contudo, este comando não altera os arquivos do nosso repositório, apenas faz o git passar a considerar o commit `<id03>` como último. O que tiver de diferente entre os nossos arquivos e o commit `<id03>` aparecerá como modificações quando usarmos `git status`. Se trocarmos de ideia, é fácil desfazer esse "reset" sem perder o trabalho atual. Basta usar o comando `git reset HEAD@{1}` [6]. Porém cuidado! Após algum tempo, o Git irá deletar permanentemente os commits que vieram depois do `<id03>`.

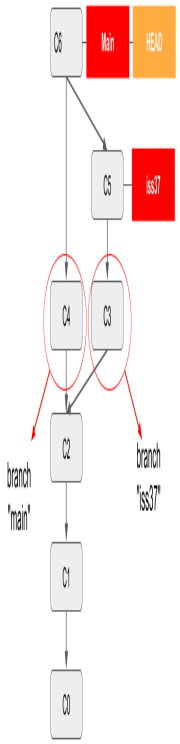
2.2. Usando branches

O que vimos até agora é o básico de Git. É o mínimo necessário para podermos criar um repositório, e começar a versionar o nosso código com a segurança de que todas as mudanças estão sendo acompanhas pelo Git.

Se usarmos apenas esses conhecimentos, nós teríamos uma "linha central" do Git com todas as mudanças que fizemos até hoje. Todas as versões do código estariam ali, iguais para todos.

No entanto, o Git nos dá a capacidade de fazer alterações em diferentes partes do código (e compartilhar com outros usuários) em paralelo, montando "ramos" separados, ou seja, históricos de commits separados. Depois, quando quisermos, podemos juntar estes ramos, obtendo devolta um único histórico de mudanças.

Assim, mudanças feitas em cada ramo não comprometem código que está funcionando na "linha central" (e que possivelmente está rodando em produção). Estes ramos são chamados de **branches**.



(**Fonte:** imagem adaptada da original, disponível no livro Pro Git [1])

Na imagem de exemplo acima, nós tínhamos um histórico de commits (C0 <- C1 <- C2, onde a seta aponta para o commit prévio àquele descrito), e fizemos uma ramificação. Começamos um histórico da branch "iss37", separado do principal na branch "main". Assim, podemos ir fazendo alterações em uma parte do código, na branch "iss37", e apenas depois juntamos novamente com a branch "main".

Vemos também um ponteiro *HEAD*. Ele é um ponteiro especial do git que sempre aponta para onde nosso diretório de trabalho está. Se trocarmos de branch, ele passa a estar em outra branch. Ele é parecido ao ID de um commmit, no sentido que podemos usar ele nos comandos que vimos na sessão anterior. Por exemplo, `git reset HEAD` remove todos os arquivos da área de *staging*. O commit que se encontra antes da HEAD pode ser utilizado como `HEAD^1`. Assim, `git reset HEAD^1` dá um rollback no repositório, voltando 1 commit no passado.

Para criar uma nova branch, usamos o comando `git branch [nome da branch]`. Para ver todas as branches que temos localmente, o comando é `git branch`. Com o comando `git branch -a` podemos ver todas as branches existentes tanto no repositório local quanto no remoto. Quando quisermos trocar de branch, no nosso diretório local, usamos o comando `git checkout [nome da branch]`.

É possível fazer as duas coisas ao mesmo tempo, criar a branch e alterar nosso diretório de trabalho para usar a branch criada. Basta usar o comando `git checkout -b [nome do branch]`.

Para mandar essa branch para o GitHub e criar uma branch remota que dialogue com essa local, usamos o comando `git push origin -u [nome da branch]`. Depois de fazer isso pela primeira vez, podemos usar o comando `git push` direto, desde que estejamos na branch certa.

Tome cuidado! Ao mudar entre branches locais, qualquer mudança que você tiver feito e mas que esteja apenas na área de *staging* ou de mudanças pode ser jogada fora. Para evitar isso, crie um commit (se for o caso) ou, se não estiver certo que irá querer manter as alterações, use o comando `git stash`. A *stash* é um espaço do git que guarda mudanças em um commit temporário, e que pode ser recuperado facilmente. Para recuperar essas mudanças, usamos o comando `git stash pop`. Se tiver mais de um commit salvo na stash, precisaremos usar esse comando diversas vezes. Podemos também olhar o ID do commit temporário específico que queremos com o comando `git stash list`.

Em geral, o commit temporário irá estar com o ID `stash@{N}`, onde N é a posição na lista da *stash*. Assim, ele pode estar nomeado `stash@{0}`, `stash@{1}`, `stash@{2}`, etc.

Se você estiver satisfeito com o trabalho feito em uma branch, e quiser juntá-la à main, basta voltar para a main e usar o comando `git merge [nome da branch]`. Você também pode fazer isso para juntar os trabalhos feitos em quaisquer duas branches, bastando usar o comando a partir de uma delas. É comum chamar esse processo de um *merge* entre as duas *branches*.

Normalmente, esse comando bastaria para que o git juntasse o histórico das duas branches. Contudo, caso a mesma parte de um arquivo tenha sido modificada nas 2 branches, o merge vai falhar, e te avisará quais os arquivos problemáticos.

No caso de um sucesso, teremos um commit relativo ao merge. Assim, se quiser, você já pode deletar a sua branch antiga, pois tudo já estará na main (ou na nova branch).

Para deletar uma branch local, usamos `git branch -d [nome da branch]`. Já para deletar uma branch remota, usamos `git push origin --delete [nome da branch]`. É possível também usar uma versão reduzida do comando, substituindo a flag por dois pontos, dessa forma:

```
git push origin :[nome da branch]
```

2.3. Resolvendo conflitos de merge

Quando você tenta dar um merge em dois branches ou dar pull nas mudanças remotas do seu repositório, o git tenta puxar todas as alterações feitas nos arquivos do repositório (ou na branch que estamos dando merge). Quando ele faz isso, ele vê os arquivos que estão diferentes, e busca juntar as duas versões linha a linha, aplicando as mudanças mais recentes controladas pelo Git.

No entanto, se tiver alterações na mesma parte do arquivo nas duas versões (se existirem commits que alteram o mesmo bloco do arquivo), o merge (ou o pull) vai falhar, e você vai ter que ajustar as diferenças manualmente. Essas diferenças é o que chamamos de *conflitos*, e o Git irá não só te avisar em quais arquivos há conflitos, como também marcar na linha específica do arquivo qual é o conflito.

Em cada arquivo, veremos um texto parecido com o da imagem abaixo.

(**Fonte:** imagem original Lets Code)

Nesse caso, onde temos o marcador `<<<<<<< HEAD` é onde estão as mudanças na main (ou na branch na qual rodamos o comando de merge). Abaixo do marcador `=======` estão as mudanças da branch que usamos como argumento do comando. Algo semelhante ocorre ao fazermos pull, quando o repositório remoto tem uma alteração em um arquivo que se sobrepõe a alguma alteração que fizemos localmente.

Para resolver esse conflito, você escolhe qual versão que quer manter, e edita o arquivo. Lembre-se de remover todos os marcadores de conflito (`<<<<<<<`, `=======` e `>>>>>>>`)!

Depois de terminarmos de fazer isso em todos os conflitos em toda a base de código, adicionamos os arquivos à área de *staging* e criamos o commit final que irá resolver o merge das branches (ou um commit após o pull). O conflito agora está resolvido!

3. Git Workflow

Ao longo deste texto, vimos que o Git tem milhares de funcionalidades. O objetivo final de todas elas é facilitar a logística do trabalho em equipe, na hora do desenvolvimento de códigos complexos.

Existem diversas formas de aproveitarmos essas funcionalidades para organizarmos um fluxo de trabalho para a nossa equipe. Essa organização do fluxo de trabalho é chamada de **git workflow**.

Uma das formas mais comuns de se trabalhar com uma equipe usando o Git é separar trabalhos individuais em branches, que irão depois sofrer um merge com a *main*. Em empresas, é comum inclusive existir uma liderança técnica que deve rever as alterações e aprovar (ou não aprovar) esse merge com a *main* no repositório remoto. Pedir aprovação para um merge com a main no repositório remoto é comumente chamado de *pull request*.

A imagem abaixo mostra um exemplo de como esse fluxo de trabalho funciona.

(**Fonte:** imagem adaptada da original, disponível nos tutoriais da Atlassian [5])

Neste caso, a branch *main* é deixada como a branch principal. A versão do código disponível nela é a que será utilizada diretamente com os nossos usuários. Em geral, isso é feito copiando esse código para o chamado *ambiente de produção*.

É na branch *main* que fazemos o controle de versão do nosso software (sempre que você ver uma versão "2.1", "2.9", "3.2", é a versão da branch main). O evento de lançar esse código para ser utilizado pelo usuário final é chamado de **release**.

Criamos também uma branch com o nome *develop* como uma branch secundária. O objetivo dela é ser uma branch central que vai ficar recebendo todas as alterações do código, antes de enviarmos a versão final para a *main*.

O objetivo aqui é que, dessa forma, podemos realizar diversos testes na branch *develop* antes de mandá-las para a main e fazer o release (colocar as alterações na *main* e subir para o ambiente de produção é oq ue chamamos de *deploy*). Se qualquer teste falhar, pelo menos não

afetamos os nossos usuários, e podemos dar um rollback na *develop* e tentar novamente consertar os bugs que ficaram.

Por fim, cada pessoa que for colaborar com o código irá criar uma nova branch de trabalho, seja para adicionar uma nova feature ao programa (como as branches *Feature1* e *Feature2* na imagem), seja para consertar um bug (branch *Bug1* na imagem). Todo o trabalho da pessoa será feito nessa branch. Apenas quando este trabalho estiver completo, é que fazemos o merge de volta para a branch *develop*.

Cada equipe (e cada empresa) tem um padrão diferente de nomenclatura para essas branches de trabalho.

4. Aliases

No fim deste texto, tratamos de uma última funcionalidade interessante do git, chamada de **aliases**.

Aliases são atalhos para comandos do git. Esses atalhos são úteis, principalmente para simplifcar comandos muito longos ou comandos não intuitivos. Há duas formas de criar aliases:

1. Usando o comando `git config --global alias.<atalho> <comando>`.
2. Editando o arquivo `.gitconfig` no diretório home (`~`) (no caso do Windows, em geral será no seu diretório `C:\`), adicionando linhas dessa forma:

```
[alias]
    <atalho1> = <comando1>
    <atalho2> = <comando2>
    (...)
```

Por exemplo, podemos executar o comando
`git config --global alias.graph log --all --graph --decorate --color --oneline` ou editar o arquivo `.gitconfig`, adicionando o abaixo:

```
[alias]
    graph = log --all --graph --decorate --color --oneline
```

Isso nos permite executar o comando `git log --all --graph --decorate --color --oneline` digitando apenas `git graph`.

Referências:

1. Livro Pro Git, disponível em https://git-scm.com/book/en/v2 ;
2. Wikipedia: Git, disponível em https://en.wikipedia.org/wiki/Git ;
3. Searching for RH Counterexamples - Setting up Pytest, Jeremy Kun, disponível em https://jeremykun.com/2020/09/11/searching-for-rh-counterexamples-setting-up-pytest/ ;
4. Git for Windows, disponível em https://gitforwindows.org/ ;
5. Tutorial da Atlassian, disponível em https://www.atlassian.com/git/tutorials ;

6. Stackoverflow, "How do I undo 'git reset'?", disponível em https://stackoverflow.com/questions/2510276/how-do-i-undo-git-reset ;