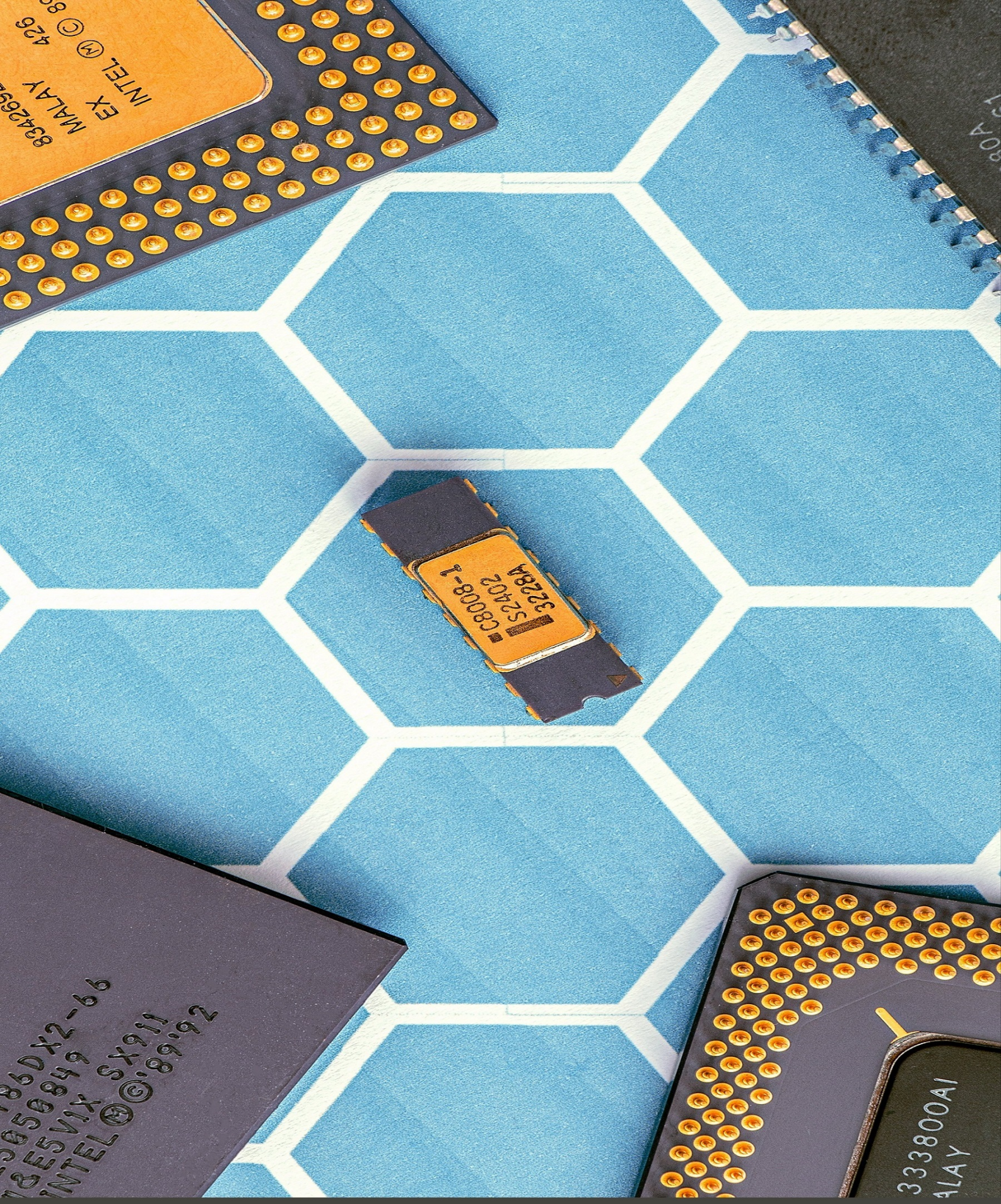



# Manual de Versionamento de Código

# Tabela de conteúdos

# Apresentação

Esse manual é uma ferramenta de aprendizado institucional. Ele condensa o conhecimento acumulado a respeito de versionamento de código e o disponibiliza para todos os membros da organização.

Para que o conteúdo esteja sempre atualizado, é importante que cada funcionário dedique parte do seu tempo para inserir seus aprendizados — relacionados ao tema versionamento de código — neste manual. Dessa forma, é possível tornar toda descoberta local em aprendizado global.

# O que é um *VCS*?

Um *Version Control System* (*VCS*) é um *software* de gerenciamento das alterações feitas em arquivos. Ele possibilita que as alterações feitas possam ser comparadas, restauradas e mescladas.

# Benefícios de um *VCS*

Um *VCS* é uma ferramenta fundamental para o desenvolvimento de *softwares*. Seus principais benefícios são:

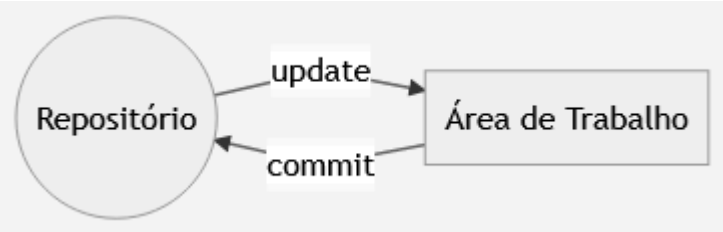
- Registro histórico: Toda a evolução do projeto, cada alteração sobre cada arquivo é guardada. Por conta disso, é possível identificar o autor, o dia e qual alteração foi feita.
- Colaboração concorrente: Possibilita que vários desenvolvedores trabalhem em paralelo sobre os mesmos arquivos sem que um sobrescreva o código de outro.
- Variações no Projeto: Mantém linhas diferentes de evolução do mesmo projeto. Por exemplo, mantendo uma versão 1.0 enquanto a equipe prepara uma versão 2.0.

# Funcionamento de um *VCS*

Um *VCS* é composto de duas partes, são elas:

- Repositório: Armazena todo o histórico de evolução do projeto, registrando toda e qualquer alteração feita em cada arquivo .
- Área de trabalho: Armazena uma cópia dos arquivos presentes no repositório. Essas cópias podem ser modificadas pelo programador de acordo com sua vontade — qualquer alteração feita nos aquisitivos é monitorada, dessa forma é possível identificar todas mudanças feitas.

Na prática, acontece o seguinte:



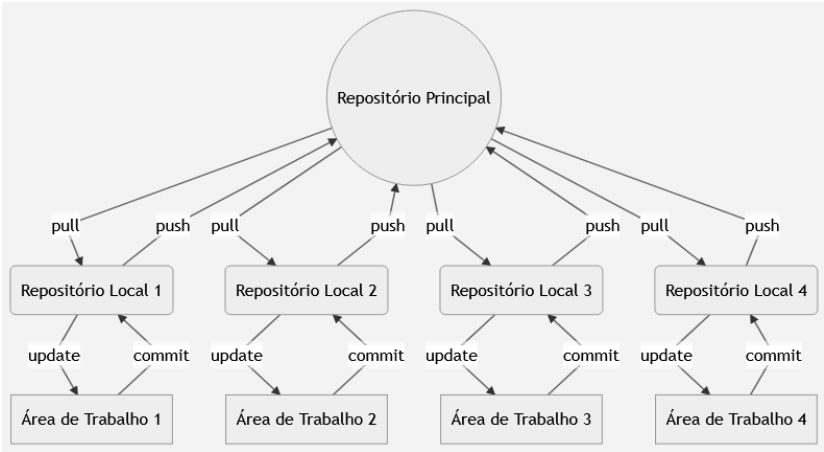

As alterações feitas na área de trabalho podem ser enviadas para o repositório sempre que o desenvolvedor quiser — essa ação é conhecida como *commit*. Em contra partida, caso o desenvolvedor queira ter acesso aos arquivos contidos no repositório, basta atualizar sua área de trabalho — ação conhecida como *update*.

## Tipos de *VCS*

Atualmente, os sistemas de controle de versão são classificados em dois tipos, são eles:

- *Centralized Version Control System*(*CVCS*): Trabalha com um servidor, que funciona como repositório central único, e áreas de trabalho que são utilizadas nas máquinas dos desenvolvedores. As áreas de trabalho se comunicam apenas através do repositório central, por meio de *commit* e *update*. Abaixo segue uma representação de um *CVCS*:


- *Distributed Version Control System*(*DVCS*): Cada área de trabalho tem seu repositórios individuais, ou seja, as operações de *commit* e *update* são feitas nas máquinas dos desenvolvedores. Porém, existe um servidor remoto que funciona como repositório oficial e permite a comunicação entre os repositórios individuais — o envio de arquivos do repositório local para o oficial é chamado de *push*, já o caminho inverso é chamado de *pull*. Abaixo segue uma representação de um *DVCS*:




Os *CVCS's* atende a maioria das equipes de desenvolvimento. Contudo, caso a quantidade de interações feitas entre as áreas de trabalho e o repositório seja maior que a suportada pelo servidor o mesmo pode ser sobrecarregado e o trabalho da equipe ser prejudicado.

Por outro lado, os *DVCS's* não sofrem desse problema — o uso de repositórios individuais acaba poupando o repositório oficial. Entretanto, é necessário um maior conhecimento da ferramenta por parte do desenvolvedor.

# O que é Git?

O Git é um *DVSC Open Source* criado em 2005 por Linus Torvalds. Hoje o Git é o *VSC* mais utilizado no mundo.

# Instalação do Git

O Git está disponível para os três principais sistemas operacionais utilizados em *desktops* e *notebooks* — GNU/Linux, Windows e MacOS. O tutorial de instalação para todos os sistemas operacionais compatíveis pode ser encontrado na página https://git-scm.com/downloads.

# Configurações iniciais

Após instalar o Git, a primeira coisa a ser feita é configurar o nome de usuário — para isso, basta utilizar o comando `git config --global user.name "Fulano de Tal"`, lembre-se de editar o comando com o nome de usuário desejado — e o endereço de e-mail — para isso, basta utilizar o comando `git config --global user.email fulanodetal@exemplo.br`, lembre-se de editar o comando com o email do usuário desejado. Essas informações são importantes porque ficarão carimbadas de forma imutável nos *commits* criados.

Para conferir se o nome e o e-mail foram configurados corretamente, basta utilizar, respectivamente, os comandos: `console git config user.name` e `console git config user.email`.

# Obtendo um repositório

Há duas formas de começar a trabalhar em um projeto com o Git. A primeira, é fazendo um *clone* — ação de copiar um projeto de algum repositório principal para o computador do desenvolvedor. O comando `git clone`, acompanhado do endereço do repositório principal, é o responsável por efetuar um *clone*.

Já a segunda, é transformar um diretório local que não está sob controle de versão em um repositório Git. Para isso, é preciso:

1. Entrar no diretório que se deseja versionar;
2. Executar o comando `git init` — ele criará um subdiretório chamado `.git` que contém todos os arquivos necessários para a criação do repositório local;
3. Indicar para o Git que ele pode cuidar de todos os arquivos presentes no diretório com o comando `git add .` — é necessário que o diretório contenha, no mínimo, um arquivo;
4. Enviar o arquivo para o repositório local por meio do comando `git commit -m "primeiro commit"` — dentro das aspas é colocada uma mensagem que permite identificar facilmente o *commit*.

5. Criar a *branch* principal do projeto. Uma *branch* é uma linha cronológica que organiza os *commits* — todo repositório precisa de no mínimo uma *branch* que é a principal. Por padrão a *branch* principal é chamada de *main*. A criação dela é feita pela execução do comando `git branch -M main`.

Agora é preciso enviar os arquivos guardados no repositório local para um repositório remoto, ou seja, para o repositório principal. Para isso é preciso:

1. Criar o repositório principal no servidor remoto;
2. Sincronizar o repositório local com o remoto com o comando `git remote add origin`, seguido do endereço do repositório principal;
3. Enviar a *branch* principal, com o *commit* feito anteriormente, para o repositório principal. O comando que faz essa tarefa é o `git push -u origin main`.

Devido a complexidade de começar a trabalhar em um projeto Git pela segunda forma é comum, neste momento, não entender todos os conceitos. Não precisa se preocupar, eles serão revisitados e melhor explicados nas próximas seções. No momento, o mais importante é perceber que é mais simples o desenvolvedor criar o repositório principal no servidor remoto e em seguida cloná-lo para sua máquina.

# Gravações e remoções de alterações nos repositórios

Como explicado na seção "Tipos de *VCS*" um *DVCS* é constituído por três partes: a área de trabalho, o repositório local e o repositório principal. No Git, os arquivos presentes na área de trabalho podem está em um dos seguintes estados:

- Não rastreados: Arquivos que não possuem nenhum *snapshots*, ou seja, nunca tiveram algum estado registrados pelo Git ou salvo em um dos repositórios;
- Rastreados: Arquivos que o Git conhece, ou seja, que tiveram seu último *snapshots* registrado pelo Git ou salvo em algum dos repositórios.

  É possível visualizar o estado dos arquivos que estão na área de trabalho utilizando o comando `git status`.

Para começar a rastrear um novo arquivo, deve-se usar o comando `git add` seguido pelo nome do arquivo que se deseja rastrear. Na seção anterior foi mostrado comando `git add .`, ele faz com que todos os arquivos da área de trabalho sejam rastreados — o comando `git add *` também tem a mesma finalidade.

Caso tenha se arrependido de ter adicionado algum arquivo, basta utilizar o comando `git reset` seguido pelo nome do arquivo. Quando o nome não é informado, todos os arquivos adicionados voltarão a ser não rastreados — lembre-se que esse comando não descarta os arquivos ou as informações contidas nele.

O `git add` também serve para selecionar os arquivos que entrarão na área de *stage*, é nela que ficam todos os arquivos que irão constituir um *commit*. Após selecionar os arquivos, chega o momento de *commitar*, ou seja, enviar os arquivos selecionados para o repositório local. Para isso, basta executar o comando `git commit -m "Nome do commit"` — dentro das aspas é colocada uma mensagem que permite identificar facilmente o *commit*.

Há momentos em que um *commit* precisa ser desfeito, existem diferentes formas de atingir esse objetivo, veja:

- Para resetar o repositório local para o último *commit* sem altera os arquivos, apenas o *commit*, o comando utilizado é o `git reset` — para fazer a mesma ação descartando os arquivos o comando é o `git reset --hard`.
- Para resetar o repositório local para o *commit* anterior sem altera os arquivos, apenas o *commit*, o comando utilizado é o `git reset HARD~1` — para fazer a mesma ação descartando os arquivos o comando é o `git reset --hard HEAD~1`.
- Para resetar o repositório local para um *commit* específico sem descartar as alterações feitas, é preciso utilizar o comando utilizado é o `git reset` com o código identificador do *commit* — para ver o código dos *commits* basta utilizar o comando `git log --oneline`. Para fazer a mesma ação descartando os arquivos o comando é o `git reset --hard` com o código identificador do *commit*.

Outro cenário comum é quando se deseja fazer o *update* da área de trabalho, ou seja, quando já existe algum *commit* do arquivo e o mesmo sofreu alterações que estão somente na área de trabalho, mas há necessidade de descartá-las para ter na área de trabalho o arquivo fidedigno ao último *commit*. Nesses casos, utiliza-se o comando `git checkout` seguido pelo nome do arquivo — caso queira descartar as alterações feitas em todo os arquivos basta utilizar `git checkout *`.

Se não houver problemas com o *commit* é possível enviá-lo para o repositório principal com o comando `git push` — o Git permite acumular *commits* no repositório local e utilizar o `git push` para submeter todos de uma só vez.

O processo para desfazer um *commit* que está no repositório principal é semelhante ao utilizado para restaurar um *commit* local descartando os arquivos. O primeiro passo é executar o comando `git reset --hard` com o código identificador do *commit*. Em seguida, utiliza-se o comando `git push --force` para forçar o *push* que descartará os *commits* posteriores ao indicado no comando.

Para atualizar o repositório local com os arquivos do repositório remoto basta executar o comando `git pull`. Para que ele funcione é preciso que tanto a área de trabalho quanto o repositório local não tenha nenhuma alteração que precise ser enviada.

Mover ou excluir um arquivo rastreado do diretório também não é uma tarefa complexa. Para mover o comando utilizado é o `git mv` sucedido pelo nome caminho para o arquivo de origem e o caminho para o diretório de destino. Já para excluir o comando utilizado é o `git rm` sucedido pelo nome do arquivo que será descartado.

# Utilizando branchs

Todo repositório tem que ter no mínimo uma *branch*, a principal — por padrão chamada de *main*. *Branch* é uma linha cronológica que organiza os *commits*.

Não inserir *commits* diretamente *main* é uma boa prática adotada mundialmente. Normalmente, os desenvolvedores codificam de forma isolada. Para isso, eles ramificam a *main*, ou seja, cada um bifurca a *branch* principal para que o código desenvolvido seja feito em uma *branch* particular, que não interfere no trabalho feito pelos outros desenvolvedores.

O comando `git checkout -b` sucedido por um nome, cria uma *branch* com o nome informado no repositório local. Para criá-la no repositório principal, basta executar `git push origin` seguido pelo nome da *branch* criada com o comando anterior.

O Git também permite navegar facilmente entre as *branchs* já criadas. O comando comando `git checkout` acompanhado pelo nome de uma *branch* altera o conteúdo da área de trabalho do desenvolvedor para o conteúdo guardado pela *branch* selecionada.

Seguindo a boa prática, a *main* só recebe *commits* por meio de *merge*. O *merge* ocorre quando o desenvolvedor está seguro que o código feito por ele pode ser mescladas com o código presentes na *main*.

Para efetuar o *merge*, o desenvolvedor deve entrar na *main* e executar o comando `git merge`, seguido pelo nome da *branch* que tem o código que o desenvolvedor deseja trazes para a *main*. Essa lógica pode ser aplicada para fazer o *merge* entre qualquer *branch*.

O comando `git merge` atua apenas no repositório local. Para enviar o *merge* para um repositório principal é preciso rodar o comando `git push`.

# Criando tags

O Git tem a habilidade de marcar pontos específicos na história como sendo importantes.Existem dois tipos de tag, são elas:

- Tag leve: Apenas aponta para um *commit* em especifico. É utilizada para destacar *commits*;
- Tag anotada: Armazenam detalhes sobre o estado do repositório naquele momento. É utilizada para identificar *releases* — quando um recurso ou conjunto deles se torna disponível para todos ou alguns clientes.

Para criar uma tag leve basta utilizar o comando `git tag -a v1.0` — lembre-se de editar o comando com o número da versão. Já para criar uma tag anotada, utiliza-se o comando `git tag -a v1.0 -m "primeira release"` — lembre-se de editar o comando com o número da versão e uma mensagem, respectivamente. Os dois comandos mostrados, criam tags apenas no repositório local. O comando utilizado para enviá-las ao repositório principal é o `git push origin` seguido pelo nome da tag que deseja enviar.

Para listar todas as tags do repositório basta utilizar o comando `git tag`. Porém, se o desejo é visualizar informações de um tag, basta utilizar o comando `git tag show` seguido pelo nome da tag que deseja ver as informações.

## Visualizar os *logs*

O Git possui um registro de *logs*. Para visualizar basta utilizar o comando `git log` — se preferir uma visualização mais resumida, utilize o comando `git log --oneline`.

# O que é Git-Flow

O Git-Flow é um *workflow* criado em 2010 por Vincent Driessen. Ele auxilia tanto no *Continuous Integration*(*CI*) quanto no *Continuous Delivery*(*CD*) ou no *Continuous Deployment*(*CD*) do *software* por meio de um modelo que atribui funções bem específicas para diferentes *branchs* e define quando elas devem interagir.

# Como funciona

O Git-Flow é apenas uma ideia abstrata do fluxo de trabalho Git, ou seja, ele dita que tipos de *branch* configurar e como fazer o *merge*. A seguir serão listadas as *branchs* básicas — *main* e *develop* — as *branchs* de suporte — *feature*, *release*, *hotfix*, *bugfix* e *support* — e seus respectivos objetivos.

## *main*

Contém todo código já testado e versionado que será colocado em ambientes de homologação ou produção. É conveniente marcar todos os *commits* na ramificação *main* com uma *tag* que indique o número de versão do projeto.

## *develop*

É utilizada para integração dos recursos desenvolvidos. Ela é uma ramificação da *main* que é feita logo no início do projeto.

## *feature*

*Branchs* ramificadas da *develop* e utilizadas para desenvolver — o Git-Flow exige que cada nova funcionalidade esteja contido em uma *branch* e seja desenvolvido nela. Por conta disso, é comum um repositório possuir várias *branchs* do tipo *feature*.

Para identificar cada uma das *features*, atribui-se o nome da funcionalidade que será desenvolvida no nome da *branch*. Por exemplo, a *branch* `feature/login`, foi criada para desenvolver a funcionalidade de *login*. Assim, apesar de existir várias *branchs* de *feature*, só existe uma utilizada para desenvolver a funcionalidade *login*, indicada no nome.

Quando a funcionalidade está pronta, o desenvolvedor pode fazer o *merge* da sua *feature* na *develop*. As *features* não devem nunca interagir direto com a ramificação *main*.

## *release*

Uma vez que a *develop* adquiriu recursos suficientes para um lançamento, é feita uma ramificação dela para criar a *branch* do tipo *release*. Após sua criação, nenhuma nova funcionalidade deve ser desenvolvida, só é permitido fazer atualizações de segurança, geração de documentação e outras tarefas relacionadas ao lançamento.

Quando estiver pronta para ser lançada, é feito dois *merges* da *release*. O primeiro, é feito para a *develop*. Dessa forma, as futuras *features* criadas poderão dispor do que foi feito na *release*. Já o segundo, é feito para a *main*. Essa ação significa que o código está apto para ser colocado em ambientes de homologação ou até mesmo de produção.

Ao trabalhar com a *branch release* é comum adicionar ao seu nome o numero da versão do projeto — por exemplo `release/1.0`. Outra boa prática é, após fazer o *merge* da *release* na *main*, criar uma *tag* que indique o número de versão do projeto.

## *hotfix*

Trata-se de uma ramificação da *main* utilizada para corrigir com rapidez *bugs* do ambiente de produção. Essa é a única *branch* que deve ser bifurcada diretamente da *main* a qualquer momento.

Assim que a correção é feita, deve-se fazer dois *merges* da *hotfix*. O primeiro, é feito para a *develop*. Dessa forma, a correção dos *bugs* estarão disponíveis para as próximas *features* criadas. Já a segunda, é feito para a *main*. Esse *merge* tem o objetivo de lançar a correção do *bug* no ambiente de homologação ou até mesmo de produção.

Ao utilizar uma *branch hotfix*é recomendável adicionar ao seu nome alguma referencia que indique o *bug* solucionado — por exemplo `hotfix/cadastro_datas_futuras`. Outra boa prática é, após fazer o *merge* da *hotfix* para a *main*, criar uma *tag* que indique o número de versão do projeto na *main*.

## *bugfix*

As vezes, *bugs* são identificados na *develop* ou na *release*. Nesse caso se cria uma *branch* do tipo *bugfix*. Após finalizar a correção é feito o *merge* da *bugfix* para a *branch* que a originou — seja ela a *develop* ou a *release*.

## *support*

*Branchs* do tipo *support* não são abordadas pelo Git-Flow, mas são essenciais quando existe a necessidade de manter várias versões do mesmo projeto.

É muito importante adicionar ao nome da *suport* o numero da versão do projeto — por exemplo `support/1.0`.

# Palavras-chaves

Esta seção apresenta um glossário com as palavras-chaves presentes neste manual. As explicações contidas aqui tem o objetivo de elucidar, rapidamente, a compreensão do leitor perante as informações trazidas.

| Termos e siglas | Definição e significado |
|---|---|
| *Software* | Programas usados no computador. |
| Servidor | Computador que prestar serviços a outros. |
| *Open Source* | É o desenvolvimento baseado no compartilhamento e na melhoria colaborativa do código-fonte. |
| Sistema Operacional | É o conjunto de programas que fazem a interface do usuário e seus programas com o computador. |
| *Desktop* | Computador que fica sobre a mesa, montados para serem usados de forma fixa. |
| *Notebook* | Computador portátil que pode ser transportado facilmente. |
| *Snapshot* | Registro histórico feito pelo Git de algum arquivo. |
| *Logs* | Registro de todas as operações relevantes de um *software*. |
| *Workflow* | É um fluxo de trabalho adotado por uma organização a fim de aumentar a eficiência do trabalho. |
| *Continuous Integration*(*CI*) | Refere-se à integração contínua, é quando novas mudanças no código de uma aplicação são desenvolvidas, testadas e consolidadas regularmente em um repositório compartilhado. |
| *Continuous Delivery*(*CD*) | Refere-se à entrega contínua, é quando as mudanças feitas pelo desenvolvedor em uma aplicação, são automaticamente testadas contra bugs e podem ser carregadas no ambiente de produção a partir de um comando. |
| *Continuous Deployment*(*CD*) | Refere-se à implantação contínua, é quando o processo de *Continuous Delivery*(*CD*) não precisa de um comando para entrar no ambiente de produção. |
| *Release* | No contexto de desenvolvimento, é quando um recurso ou conjunto de recursos se torna disponível para todos os clientes ou para um segmento deles. No contexto do Git-Flow, é uma das *branch* secundária. |
| Ambiente de Homologação | Servidor utilizado pela equipe de desenvolvimento para validar uma *release*. |
| Ambiente de Produção | Servidor utilizado pelos usuários finais para acessar a aplicação. |

# Referências bibliográficas

Esta seção apresenta o conjunto das fontes efetivamente utilizadas na construção deste manual. Cada referência dessa bibliografia está organizada pelo tópico que a mesma ajudou a construir — para facilitar uma busca mais aprofundada do conteúdo.

## O que é um *VCS*?


- Version Control Systems. **Geeks for Geeks**. Disponível em: https://www.geeksforgeeks.org/version-control-systems/. Acesso em: 14 de ago. 2021.
- What is VCS (Version Control System)? 7 ways to choose perfect VCS for your project!. **LinkedIn**. Disponível em: https://www.linkedin.com/pulse/what-vcs-version-control-system-7-ways-choose-perfect-nilesh-kanawade/. Acesso em: 14 de ago. 2021.
- Sistemas de Controle de Versão. **DevMedia**. Disponível em: https://www.devmedia.com.br/sistemas-de-controle-de-versao/24574. Acesso em: 14 de ago. de 2021.
- Hardware e Software. **GuiaFoca**. Disponível em: https://www.guiafoca.org/guiaonline/iniciante/ch02.html#basico-hardsoft. Acesso em: 14 de ago. de 2021.


### Benefícios de um *VCS*


- What is VCS (Version Control System)? 7 ways to choose perfect VCS for your project!. **LinkedIn**. Disponível em: https://www.linkedin.com/pulse/what-vcs-version-control-system-7-ways-choose-perfect-nilesh-kanawade/. Acesso em: 14 de ago. 2021.
- Conceitos Básicos de Controle de Versão de Software — Centralizado e Distribuído. **Blog Promus**. Disponível em: https://blog.pronus.io/posts/controle-de-versao/conceitos-basicos-de-controle-de-versao-de-software-centralizado-e-distribuido/. Acesso em: 14 de ago. de 2021.


### Funcionamento de um *VCS*


- Conceitos Básicos de Controle de Versão de Software — Centralizado e Distribuído. **Blog Promus**. Disponível em: https://blog.pronus.io/posts/controle-de-versao/conceitos-basicos-de-controle-de-versao-de-software-centralizado-e-distribuido/. Acesso em: 14 de ago. de 2021.
- Sistemas de Controle de Versão. **DevMedia**. Disponível em: https://www.devmedia.com.br/sistemas-de-controle-de-versao/24574. Acesso em: 14 de ago. de 2021.

### Tipos de *VCS*


- Conceitos Básicos de Controle de Versão de Software — Centralizado e Distribuído. **Blog Promus**. Disponível em: https://blog.pronus.io/posts/controle-de-versao/conceitos-basicos-de-controle-de-versao-de-software-centralizado-e-distribuido/. Acesso em: 14 de ago. de 2021.
- Sistemas de Controle de Versão. **DevMedia**. Disponível em: https://www.devmedia.com.br/sistemas-de-controle-de-versao/24574. Acesso em: 14 de ago. de 2021.
- Version Control Systems: Distributed vs. Centralized. **Oshyn**. Disponível em: https://www.oshyn.com/blog/version-control-systems-distributed-vs-centralized. Acesso em: 14 de ago. de 2021.
- Curso de Controle de Versão com Git. **Pronus**. Disponível em: https://www.youtube.com/watch?v=4HyrORBRS8o&t=3982s. Acesso em: 14 de ago. de 2021.


## O que é Git?


- O que é Git. **Atlassian Bitbucket**. Disponível em: https://www.atlassian.com/br/git/tutorials/what-is-git. Acesso em: 14 de ago. de 2021.
- History of the OSI. **Open Source Initiative**. Disponível em: https://opensource.org/history/. Acesso em: 14 de ago. de 2021.


### Instalação do Git


- Sistema Operacional. **GuiaFoca**. Disponível em: https://www.guiafoca.org/guiaonline/iniciante/ch01s03.html. Acesso em: 14 de ago. de 2021.
- Download. **Git**. Disponível em: https://git-scm.com/downloads. Acesso em: 14 de ago. de 2021.
- O que é Desktop?. **Dicas de Informática Básica**. Disponível em: https://www.cursosdeinformaticabasica.com.br/o-que-e-desktop/. Acesso em: 14 de ago. de 2021.


### Configurações iniciais


- Configuração Inicial do Git. **Git**. Disponível em: https://git-scm.com/book/pt-br/v2/Começando-Configuração-Inicial-do-Git. Acesso em: 16 de ago. de 2021.


### Obtendo um repositório


- Obtendo um Repositório Git. **Git**. Disponível em: https://git-scm.com/book/pt-br/v2/Fundamentos-de-Git-Obtendo-um-Repositório-Git. Acesso em: 16 de ago. de 2021.

- Adicionar um projeto existente ao GitHub usando a linha de comando. **GitHub**.Disponível em: https://docs.github.com/pt/github/importing-your-projects-to-github/importing-source-code-to-github/adding-an-existing-project-to-github-using-the-command-line. Acesso em: 16 de ago. de 2021.
- O que é um servidor em computação?. **ControleNet**. Disponível em: https://www.controle.net/faq/o-que-sao-servidores. Acesso em: 16 de ago. de 2021.


## Gravações e remoções de alterações nos repositórios


- Gravando Alterações em Seu Repositório. **Git**. Disponível em: https://git-scm.com/book/pt-br/v2/Fundamentos-de-Git-Gravando-Alterações-em-Seu-Repositório. Acesso em: 21 de ago. de 2021.
- Git, desfazendo commits. **Bruno Orlandi**. https://stackoverflow.com/questions/3293531/how-to-permanently-remove-few-commits-from-remote-branch. Acesso em 21 de ago. de 2021.
- How to permanently remove few commits from remote branch. **StackOverflow**. Disponível em: https://stackoverflow.com/questions/3293531/how-to-permanently-remove-few-commits-from-remote-branch. Acesso em: 21 de ago. de 2021.
- REPRISE - PRIMEIRA AULA PARA A CERTIFICAÇÃO OFICIAL DO GITLAB. **LinuxTips**. Disponível em: https://www.youtube.com/watch?v=SMzaAP09BD4. Acesso em: 21 de ago. de 2021.


## Utilizando branchs


- Branches em poucas palavras. **Git**. Disponível em: https://git-scm.com/book/pt-br/v2/Branches-no-Git-Branches-em-poucas-palavras. Acesso em: 22 de ago. de 2021.


## Criando tags


- Criando Tags. **Git**. Disponível em: https://git-scm.com/book/pt-br/v2/Fundamentos-de-Git-Criando-Tags. Acesso em: 22 de ago. de 2021.
- Git: criando tags. **Medium**. Disponível em: https://medium.com/rafaeltardivo/git-criando-tags-7c34ee6786be. Acesso em: 22 de ago. de 2021.


## Visualizar os *logs*


- REPRISE - PRIMEIRA AULA PARA A CERTIFICAÇÃO OFICIAL DO GITLAB. **LinuxTips**. Disponível em: https://www.youtube.com/watch?v=SMzaAP09BD4. Acesso em: 28 de ago. de 2021.
- Significado de Log. **Dicio** Disponível em: https://www.dicio.com.br/log-2/. Acesso em: 28 de ago. de 2021.


# O que é Git-Flow

- Fluxo de trabalho de Gitflow. **Atlassian**. Disponível em: https://www.atlassian.com/br/git/tutorials/comparing-workflows/gitflow-workflow. Acesso em: 24 de ago. 2021.
- A successful Git branching model. **nvie.com**. Disponível em: https://nvie.com/posts/a-successful-git-branching-model/. Acesso em: 24 de ago. 2021.
- Git Flow // Dicionário do Programador. **Código Fonte TV**. Disponível em: https://www.youtube.com/watch?v=oweffeS8TRc. Acesso em: 24 de ago. 2021.
- O que é workflow e como aplicar no seu negócio. **rockcontent**. Disponível em: https://rockcontent.com/br/blog/workflow/. Acesso em: 24 de ago. 2021.
- Integração e entrega contínuas: pipeline CI/CD. **Red Hat**. Disponível em: https://www.redhat.com/pt-br/topics/devops/what-is-ci-cd. Acesso em: 24 de ago. 2021.

## Como funciona

- Fluxo de trabalho de Gitflow. **Atlassian**. Disponível em: https://www.atlassian.com/br/git/tutorials/comparing-workflows/gitflow-workflow. Acesso em: 24 de ago. 2021.
- Utilizando o fluxo Git Flow. **Medium**. Disponível em: https://medium.com/trainingcenter/utilizando-o-fluxo-git-flow-e63d5e0d5e04. Acesso em: 28 de ago. 2021.
- Gitflow: entenda quando e como você deve utilizar. **Lumis**. Disponível em: https://www.lumis.com.br/a-lumis/blog/gitflow-entenda-quando-e-como-voce-deve-utilizar.htm. Acesso em: 28 de ago. 2021.

### *main*

- Fluxo de trabalho de Gitflow. **Atlassian**. Disponível em: https://www.atlassian.com/br/git/tutorials/comparing-workflows/gitflow-workflow. Acesso em: 24 de ago. 2021.

### *develop*

- Fluxo de trabalho de Gitflow. **Atlassian**. Disponível em: https://www.atlassian.com/br/git/tutorials/comparing-workflows/gitflow-workflow. Acesso em: 24 de ago. 2021.

### *feature*

- Fluxo de trabalho de Gitflow. **Atlassian**. Disponível em: https://www.atlassian.com/br/git/tutorials/comparing-workflows/gitflow-workflow. Acesso em: 24 de ago. 2021.

### *release*

- Fluxo de trabalho de Gitflow. **Atlassian**. Disponível em: https://www.atlassian.com/br/git/tutorials/comparing-workflows/gitflow-

workflow. Acesso em: 24 de ago. 2021.

### *hotfix*


- Fluxo de trabalho de Gitflow. **Atlassian**. Disponível em: https://www.atlassian.com/br/git/tutorials/comparing-workflows/gitflow-workflow. Acesso em: 24 de ago. 2021.


### *bugfix*


- Gitflow: qual a diferença entre “hotfix” e “bugfix”. **Medium**. Disponível em: https://medium.com/@echegorri.rodrigo/gitflow-qual-a-diferença-entre-hotfix-e-bugfix-5f5d6ac4ff18. Acesso em: 28 de ago. de 2021.


### *support*


- GitFlow Examples. **GitVersion**. Disponível em: https://gitversion.net/docs/learn/branching-strategies/gitflow/examples. Acesso em: 28 de ago. 2021