

# Controle de Versão e Git

Instrutor: Tarik Ponciano

# Links da Disciplina

1. Discord: <https://discord.gg/wt5CVZZWJs>
2. Drive:  
<https://drive.google.com/drive/folders/1hOl0DaPeAor7gnhKBUIlZ5n8lLRDNvUY?usp=sharing>
3. Github:  
<https://github.com/TarikPonciano/Programador-de-Sistema-SENAC>

# Controle de versão – O que é?

Um sistema de controle de versão (como o próprio nome já diz) tem a finalidade de gerenciar diferentes versões de um documento. Com isso ele te oferece uma maneira muito mais inteligente e eficaz de organizar seu projeto, pois é possível acompanhar um histórico de desenvolvimento, desenvolver paralelamente e ainda te oferecer outras vantagens, como exemplo, customizar uma versão, incluir outros requisitos, finalidades específicas, layout e afins sem mexer no projeto principal ou resgatar o sistema em um ponto que estava estável, isso tudo sem mexer na versão principal.

# Controle de versão – O que é?

Um Iniciante da área de desenvolvimento de software ao se deparar com seu primeiro projeto de maior porte com uma equipe de desenvolvimento, geralmente, não conhece as ferramentas usadas para trabalhar paralelamente de forma eficiente e acaba sempre com aquela terrível dúvida:

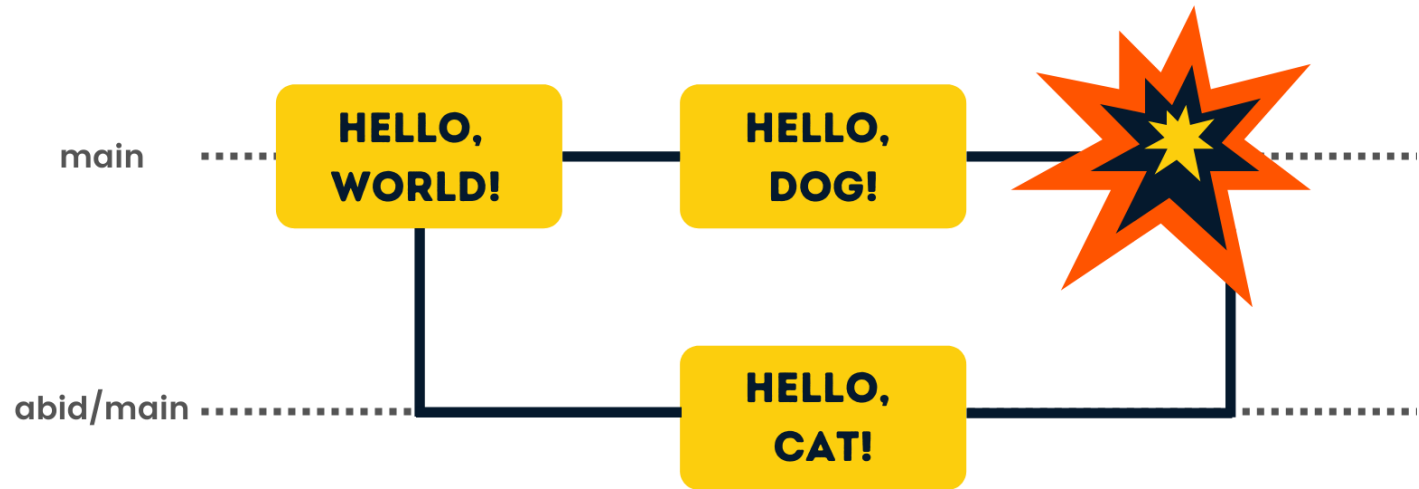
Como iremos desenvolver o projeto em paralelo? Existe a possibilidade de fazer isso sem que haja sobreposição das minhas alterações ou alguém esteja utilizando a versão errada? Como iremos resolver esse problema sem apelar para a nossa linda, muitas vezes útil, gambiarra?

# Controle de versão – Como funciona?

Basicamente, os arquivos do projeto ficam armazenados em um repositório (um servidor em outras palavras) e o histórico de suas versões é salvo nele. Os desenvolvedores podem acessar e resgatar a ultima versão disponível e fazer uma cópia local, na qual poderão trabalhar em cima dela e continuar o processo de desenvolvimento. A cada alteração feita, é possível enviar novamente ao servidor e atualizar a sua versão a partir outras feitas pelos demais desenvolvedores.

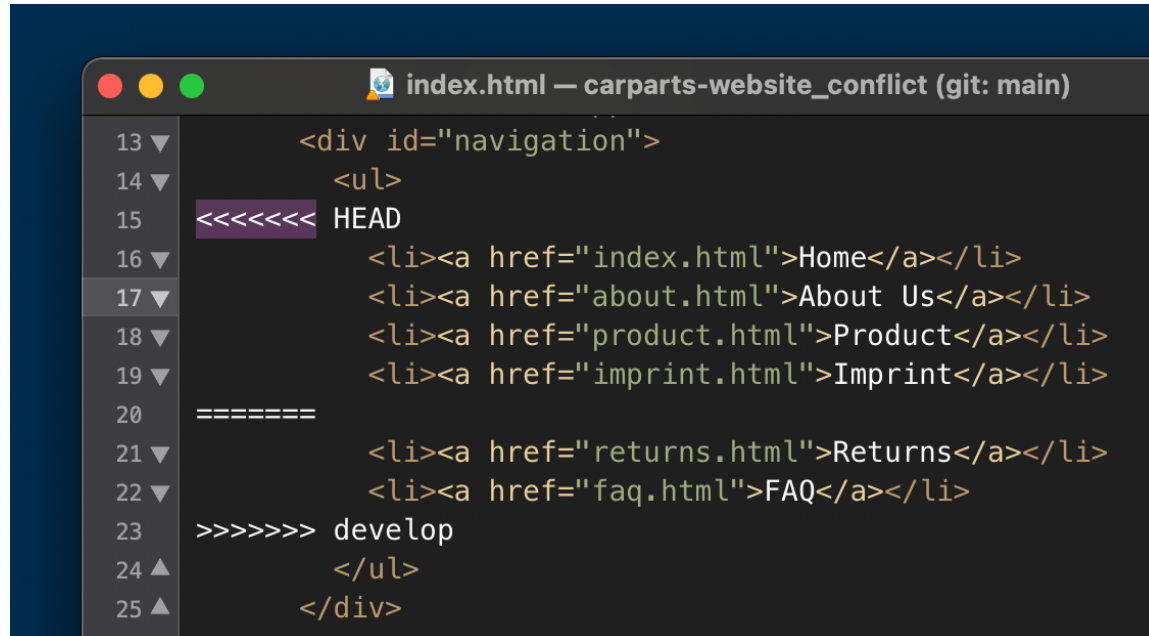


**E se por acaso os desenvolvedores estiverem  
editando o mesmo arquivo? O que irá  
acontecer se enviarem ao mesmo tempo para o  
servidor?**



# Controle de versão – Resolução de Conflitos

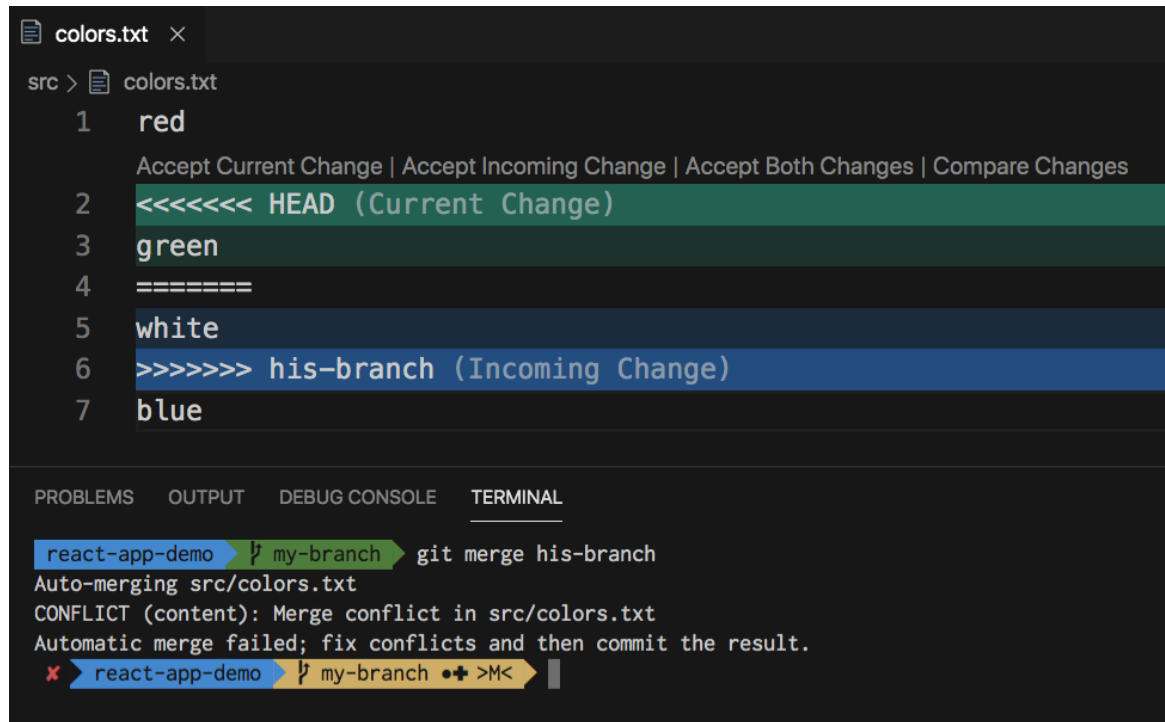
Para evitar problemas como esse, o Sistema de Controle de Versão oferece ferramentas uteis para mesclar o código e evitar conflitos.



```
13 <div id="navigation">
14 <ul>
15 <<<<<<< HEAD
16 <li><a href="index.html">Home</a></li>
17 <li><a href="about.html">About Us</a></li>
18 <li><a href="product.html">Product</a></li>
19 <li><a href="imprint.html">Imprint</a></li>
20 =====
21 <li><a href="returns.html">Returns</a></li>
22 <li><a href="faq.html">FAQ</a></li>
23 >>>>>>> develop
24 </ul>
25 </div>
```



# Controle de versão – Resolução de Conflitos



```
colors.txt x
src > colors.txt
1 red
   Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
2 <<<<<< HEAD (Current Change)
3 green
4 =====
5 white
6 >>>>>> his-branch (Incoming Change)
7 blue

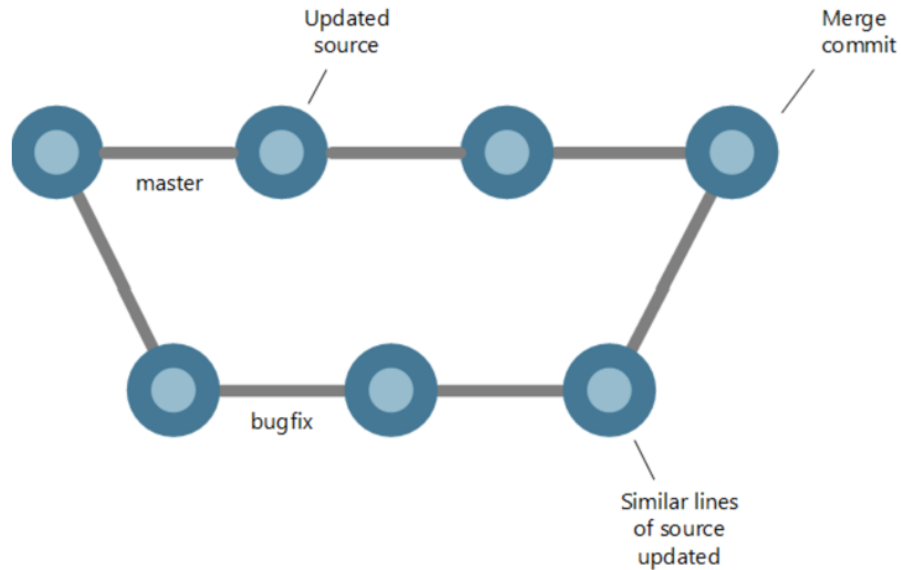
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
react-app-demo my-branch git merge his-branch
Auto-merging src/colors.txt
CONFLICT (content): Merge conflict in src/colors.txt
Automatic merge failed; fix conflicts and then commit the result.
x react-app-demo my-branch >M<
```

# Controle de versão – Resolução de Conflitos

Por exemplo: Você atualizou seu projeto (usando uma função chamada de check-out ou update) e começou a fazer suas alterações. Ao mesmo tempo, outro desenvolvedor fez alterações e atualizou a versão no servidor. Quando for enviar sua versão (usando uma função chamada de check-in ou commit) o Sistema de Controle de Versão irá alertar que o seu arquivo está desatualizado. Ele enviará as novas informações adicionadas e permitirá mesclar as diferentes versões. Não apenas isso, ele também mostrará onde foram feitas atualizações, trechos de código incluídos ou removidos e casos de conflito, onde linhas podem se sobrescrever e oferecerá opções para mesclar manualmente, escolhendo a melhor solução.



# Controle de versão – Resolução de Conflitos



# Controle de versão – Tipos

Atualmente, os sistemas de controle de versão são classificados em dois tipos: Centralizados e Distribuídos.

O **centralizado** trabalha apenas com um servidor central e diversas áreas de trabalho, baseados na arquitetura cliente-servidor. Por ser centralizado, as áreas de trabalho precisam primeiro passar pelo servidor para poderem comunicar-se. Essa versão atende muito bem a maioria das equipes de desenvolvimento que não sejam enormes e trabalhem em uma rede local, além de não ter problemas de velocidade para enviar e receber os dados e ter um bom tempo de resposta do servidor. Um dos principais sistemas com o tipo de controle de versão centralizado é o Subversion.

# Controle de versão – Tipos

O **distribuído** vai mais além. Ele é recomendado para equipes com muitos desenvolvedores e que se encontram em diferentes filiais. Esta versão funciona da seguinte maneira: cada área de trabalho tem seu próprio “servidor”, ou seja, as operações de check-in e check-out são feitas na própria máquina. Porém diferentemente do centralizado, as áreas de trabalho podem comunicar-se entre si, recomenda-se usar um servidor como centro do envio dos arquivos para centralizar o fluxo e evitar ramificações do projeto e a perda do controle sobre o mesmo, geralmente o sistema te da essa opção, oferecendo um servidor remoto para hospedar o projeto. A comunicação entre o servidor principal e as áreas de trabalho funciona com outras duas operações, para atualizar e mesclar o projeto, chamadas de **pull** e **push** (puxar e empurrar).

- pull**: Com esta operação é possível pegar a versão de outra área de trabalho e mesclar com a sua.
- push**: Com esta operação temos o processo inverso do pull, ou seja, enviando para outra área a sua versão do projeto.

# Controle de versão – Tipos

Por ser na própria máquina, o sistema de controle distribuído acaba sendo mais rápido, porém exige maior conhecimento da ferramenta e de início podem atrapalhar o desenvolvedor. Como exemplo, o sistema de mesclagem em edições concorrentes, se torna diferente por trabalhar em um sistema de arquivos binários (sequenciais de bits compostos por zero e um) que em determinadas situações não permite a comparação entre atualizações concorrentes. O sistema centralizado trabalha com arquivos de texto, que permite a comparação em atualizações concorrentes e da opção ao desenvolvedor para escolher a melhor solução.

Portanto, por esse tratamento de mesclagem ser diferente, podem ocorrer situações onde o trabalho de alguém possa ser sobreposto e gerando tormento para os desenvolvedores. Para isso existe uma função chamada lock, que bloqueia o arquivo para que não seja modificado por outros enquanto estiver com você. Os sistemas distribuídos mais conhecidos são o Git e o Mercurial.

# Git – O que é

De longe, o sistema de controle de versão moderno mais usado no mundo hoje é o Git. O Git é um projeto de código aberto maduro e com manutenção ativa desenvolvido em 2005 por Linus Torvalds, o famoso criador do kernel do sistema operacional Linux. Um número impressionante de projetos de software depende do Git para controle de versão, incluindo projetos comerciais e de código-fonte aberto. Os desenvolvedores que trabalharam com o Git estão bem representados no pool de talentos de desenvolvimento de software disponíveis e funcionam bem em uma ampla variedade de sistemas operacionais e IDEs (Ambientes de Desenvolvimento Integrado).

Tendo uma arquitetura distribuída, o Git é um exemplo de DVCS (portanto, Sistema de Controle de Versão Distribuído). Em vez de ter apenas um único local para o histórico completo da versão do software, como é comum em sistemas de controle de versão outrora populares como CVS ou Subversion (também conhecido como SVN), no Git, a cópia de trabalho de todo desenvolvedor do código também é um repositório que pode conter o histórico completo de todas as alterações.

# Git – Vantagens

Hoje, o Git é a melhor escolha para a maioria das equipes de software.

**O Git é bom** - O Git tem a funcionalidade, desempenho, segurança e flexibilidade que a maioria das equipes e desenvolvedores individuais precisa.

**Git é um padrão de fato** - Um grande número de desenvolvedores já tem experiência com o Git e uma proporção significativa de recém-formados pode ter experiência apenas com o Git.

**Git é um projeto de código aberto de qualidade** - O Git é um projeto de código aberto muito bem suportado, com mais de uma década de administração sólida.

## **Crítica:**

Uma crítica comum ao Git é que pode ser difícil de aprender. Algumas das terminologias do Git vão ser novas para os iniciantes e, para usuários de outros sistemas, a terminologia do Git pode ser diferente, por exemplo, revert no Git tem um significado diferente do que no SVN ou CVS. No entanto, o Git é muito capaz e disponibiliza muitos recursos aos usuários. Aprender a usar esses recursos pode levar algum tempo. No entanto, uma vez aprendidos, podem ser usados pela equipe para aumentar a velocidade de desenvolvimento.



# GitHub – O que é

O GitHub é considerado é uma ferramenta essencial para engenheiros de software, com uma popularidade sem igual. Atualmente, ele acomoda mais de 25 milhões de usuários. Isso significa que há um número considerável de profissionais que estão procurando o GitHub para melhorar o fluxo de trabalho e a colaboração.

Em suma, o GitHub é um serviço baseado em nuvem que hospeda um sistema de controle de versão (VCS) chamado Git. Ele permite que os desenvolvedores colaborem e façam mudanças em projetos compartilhados enquanto mantêm um registro detalhado do seu progresso.

<https://www.hostinger.com.br/tutoriais/o-que-github>

# Como utilizar o Git

O GIT é consideravelmente simples de usar. Para começar, você pode criar um repositório ou conferir um já existente. Após a instalação, um simples `git-init` irá deixar tudo pronto. Da mesma maneira, o comando `git clone` pode criar uma cópia de um repositório local para um usuário.



# Como utilizar o Git

## Instalar o GIT no Windows:

Instalar o GIT no Windows é tão simples como baixar um instalador e executá-lo. Execute os seguintes passos para instalar o GIT no Windows:

1. Acesse [o site oficial](https://gitforwindows.org/) (<https://gitforwindows.org/>) e faça o download do instalador do GIT para Windows.

2. Depois de baixado, clique duas vezes no arquivo para iniciar o assistente de instalação. Basta seguir as instruções na tela, clicando em **Next**. Ao término, clique em **Finish** para concluir com êxito a instalação.

3. Abra o **prompt de comando** ou o **git bash** e digite os seguintes comandos no terminal:

```
git config --global user.name "João Silva"
```

```
git config --global user.email "exemplo@seuemail.com.br"
```

# Como utilizar o Git

## Criar/configurar/verificar um repositório

Um repositório é o maior bem de qualquer projeto controlado por versão. Para transformar qualquer diretório em um repositório GIT, o simples comando `git init <directory>` pode ser utilizado. Uma pasta chamada `.git` também deve começar a existir no diretório em que o comando foi executado.

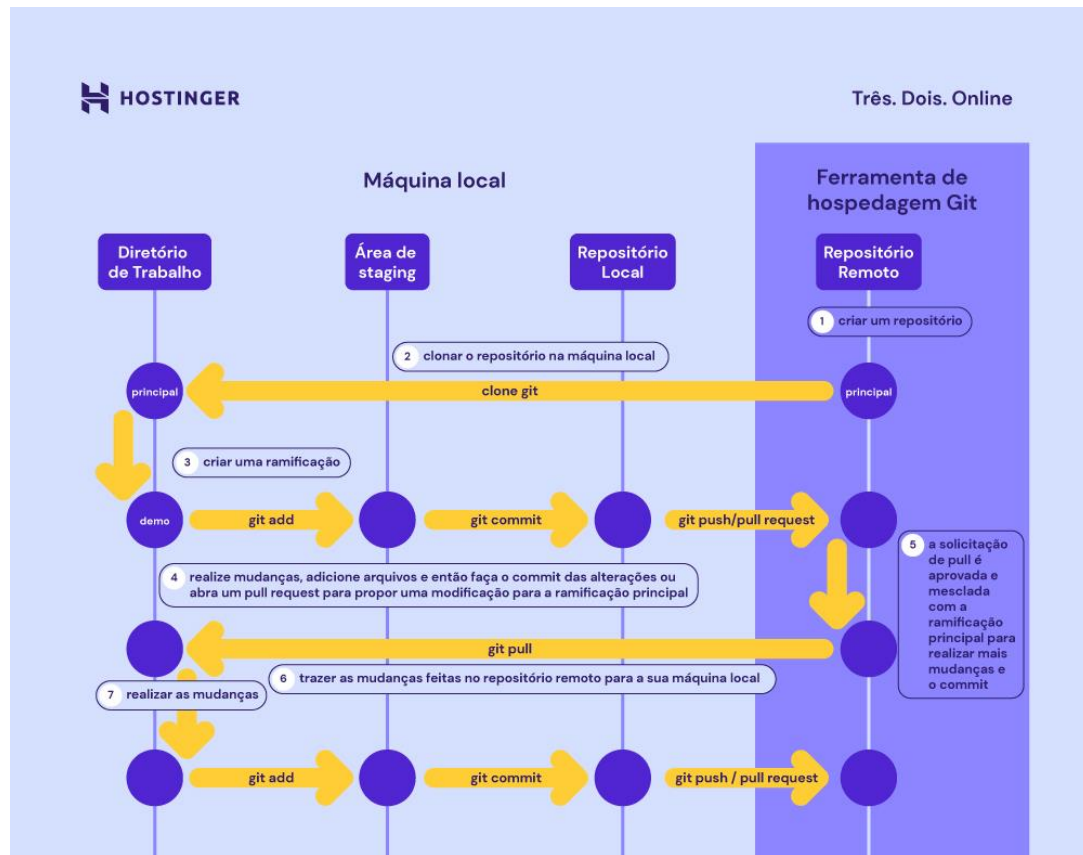
Por outro lado, se você já tem um diretório e deseja verificar (clone-lo), você pode usar o comando `git clone`. Se você estiver tentando verificar um repositório local, use o seguinte comando:

```
git clone /path/to/local/repository
```

Se você pretende verificar um repositório armazenado remotamente, use:

```
git clone user.name@host:/path/to/remote/repository (URL)
```

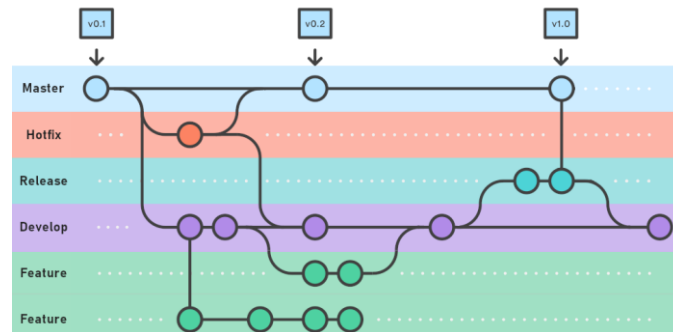
# Fluxo de Trabalho



# Fluxo de Trabalho

Cada repositório local consiste em três árvores: o **diretório de trabalho** que contém os arquivos reais; O **índice** que desempenha o papel de uma área de teste e o **HEAD** que é um ponteiro para o último *commit* feito pelo usuário.

Então, é assim que o fluxo de trabalho pode ser explicado: o usuário adiciona um arquivo ou alterações do diretório de trabalho para o índice (a área de teste) e uma vez revistos, o arquivo ou as alterações são finalmente comprometidos com o **HEAD**.



# Fluxo de Trabalho

## Os comandos *Add* e *Commit*:

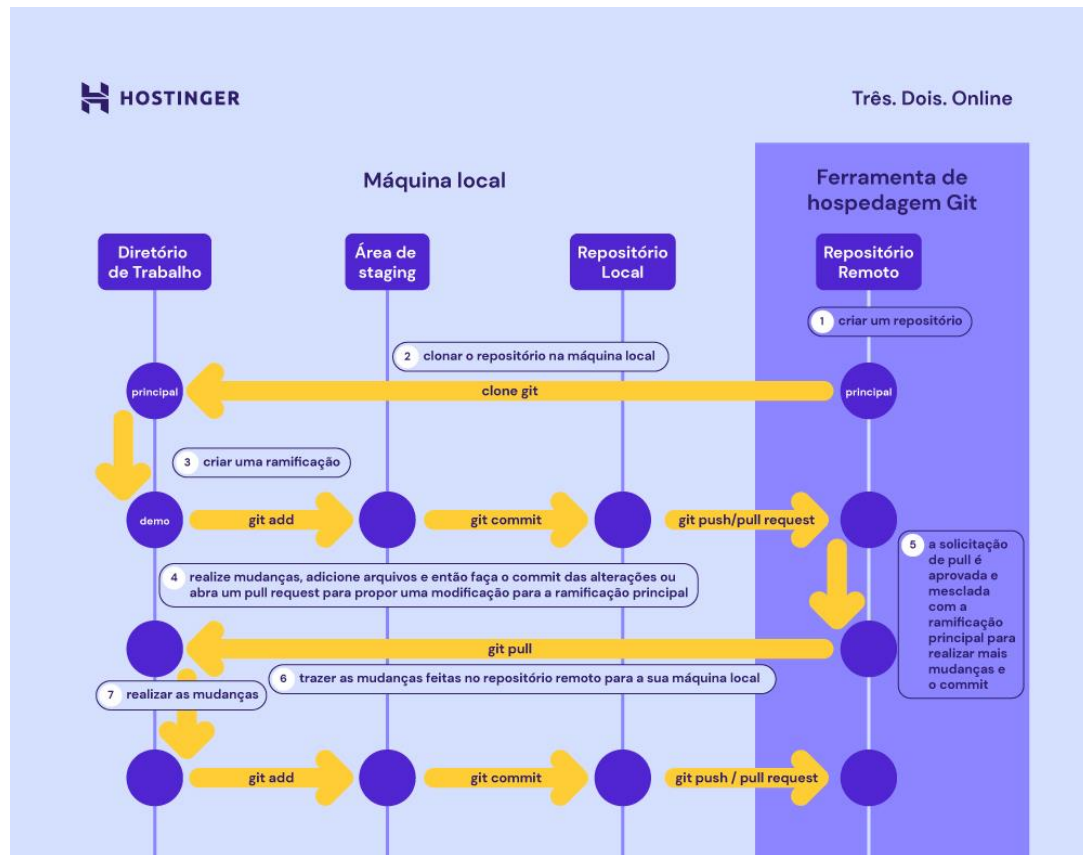
Alterações ou adições de arquivos propostas são adicionadas ao índice usando o comando *add*. Para adicionar qualquer arquivo, o comando é:

```
git add <nome_do_arquivo>
```

Se você está realmente confiante o suficiente para fazer essas mudanças no *HEAD*, então você pode usar o comando *commit*:

```
git commit -m "Adicionar qualquer mensagem sobre o commit aqui"
```

# Fluxo de Trabalho





# Fluxo de Trabalho

## Dando continuidade com as mudanças

Depois de confirmar as alterações (e acreditar que elas estão prontas para serem enviadas para o repositório original), você pode usar o comando push.

Uma vez que o `git push origin master` é executado de dentro do diretório de trabalho, as mudanças presentes no **HEAD** são enviadas para o repositório remoto. No comando acima mencionado, o **master** pode ser alterado para o nome do *branch* ao qual você deseja que as alterações sejam comprometidas.

Se, no entanto, um repositório existente ainda não tiver sido clonado e você pretende estabelecer uma ligação entre o seu repositório e um servidor remoto, execute o seguinte comando:

**git remote add origin <servidor> (URL)**

<https://docs.github.com/en/migrations/importing-source-code/using-the-command-line-to-import-source-code/adding-locally-hosted-code-to-github>

# Branches

Outra característica brilhante (mas avançada) do GIT é sua capacidade de permitir que desenvolvedores e gerentes de projeto criem vários ramos (*branches*) independentes dentro de um único projeto.

O objetivo principal de um *branch* é desenvolver novas funcionalidades, mantendo-os isolados uns dos outros. O *branch* padrão em qualquer projeto é sempre o ***master branch***. Tantos *branches* quanto necessários podem ser criados e eventualmente mesclados ao *master branch*.

Um novo *branch* pode ser criado usando o seguinte comando:

```
git checkout -b feature_n *
```

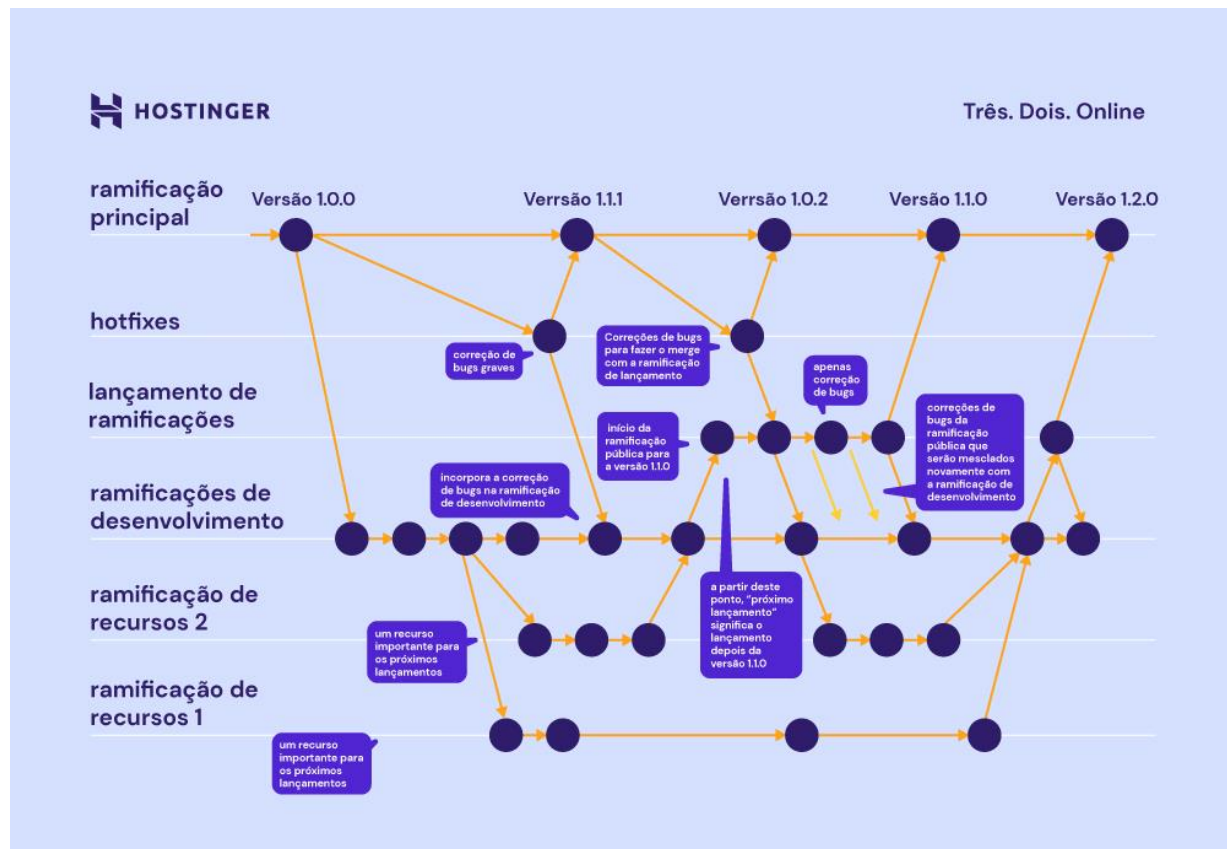
**feature\_n** é o nome do *branch*.

Se você deseja retornar ao *master branch*, o seguinte comando pode ser usado:

```
git checkout master
```

<https://www.freecodecamp.org/news/how-to-delete-a-git-branch-both-locally-and-remotely>

# Branches



# Atualizando e dando merge

Caso você queira atualizar seu diretório de trabalho local para o mais recente do repositório remoto, o simples comando `git pull` pode ser usado.

Para mesclar outro branch (dar um *merge*) no atualmente ativo, use: `git merge feature_n`

Se você der um *merge* ou *pull*, o GIT sempre tenta lidar com os conflitos por conta própria, mas as vezes não consegue. Em caso de falha devido a conflitos, o usuário tem que resolver os conflitos manualmente. Depois de editar os arquivos (para erradicar conflitos), marque-os como *merged* usando:

**`git add <nome.arquivo>`**

Se antes do *merge* você desejar visualizar as alterações, o seguinte comando pode ser executado:

**`git diff <nome_branch_origem> <nome_branch_alvo>`**

# Atualizando e dando merge

Se você acabou fazendo bagunça e precisa reverter as alterações feitas em qualquer arquivo, faça isso usando o seguinte comando

**git checkout -- <nomedoarquivo>**

Isso substituirá as alterações da árvore de trabalho pelos últimos dados presentes no **HEAD**. Quaisquer alterações que já tenham sido adicionadas ao índice não serão prejudicadas.

Por outro lado, se todas as alterações/*commits* locais devem ser eliminados e o *master branch* local for necessário para apontar para o histórico mais recente do servidor, execute os seguintes comandos:

**git fetch origin**

**git reset --hard origin/master**

**<https://www.hostinger.com.br/tutoriais/tutorial-do-git-basics-introducao>**

# Git Stash

*A ferramenta **Git** dispõe de vários comandos; porém um comando não tão conhecido e utilizado mas que pode ser aquela “mão na roda”, para o desenvolvedor é o **git stash**.*

## O que tem de tão especial no `git stash`?

No seu natural fluxo de desenvolvimento versionado você já deve ter se debatido com algumas das seguintes situações:

- Começar um desenvolvimento de tarefa na *branch* errada;
- Precisar atualizar a *branch* após já ter modificado alguns arquivos;
- Desejar criar uma *branch* a partir das suas modificações em arquivos;
- Versionar códigos incompletos *POC's\** (se for o caso), sem precisar realizar um *commit*;
- Testar o funcionamento da aplicação temporariamente desconsiderando os arquivos modificados e sem perder o que já foi feito;
- Mudar modificações não *commitadas* de *branch* sem precisar *commitar* e consequentemente realizar um *merge* e/ou PR(*pull request*);
- Dentre outras situações verbosas que envolvam *commits* e PR.

Pois bem, a utilização do `git stash` permite a você o controle sobre essas e outras situações, tornando mais prático e ágil o seu desenvolvimento de código.

# Git Stash

Falando de forma leiga: Ele cria um *backup* das modificações de seus arquivos.

De forma mais detalhada: Considerando que você tenha arquivos pendentes para *commitar*, ao utilizar o comando `git stash` no seu terminal, o **Git** vai criar uma *branch* temporária contendo a versão atual do seu projeto e após isso vai desfazer essas modificações feitas nos arquivos da sua *branch* atual, ainda não ficou claro ? Tudo bem, veja na prática um exemplo da situação descrita na prática:

```
$ git status
modified:   index.html

$ git stash
Saved working directory and index state \ "WIP on master: 049d078
added the index file" HEAD is now at 049d078 added the index file

$ git status
On branch master nothing to commit, working directory clean
```

# Git Stash

## O que aconteceu com as modificações que sumiram ?

O **Git** inteligentemente cria aquela ramificação que foi citada para você com as alterações que tinhas realizado e para recuperar basta usar o comando `git stash apply` e dessa forma, todos os arquivos que você modificou no momento do *apply* vão ser restaurado, exatamente como você deixou.

Caso você tenha realizado mais de um *stash*, você poderá escolher de uma lista qual deseja restaurar; partindo do básico, esse é o procedimento que você vai realizar:

```
$git stash list
stash@{0}: WIP on design: f2c0c72... Adjust Password Recover Email
stash@{1}: WIP on design: eb65635... Email Adjust
```

- Feito isso você pode escolher exatamente qual *stash* deseja restaurar e para isso basta utilizar o seguinte comando: `git stash apply @{id}`, sendo **id** o número identificador/posição.



# Git Stash

## Outros comandos uteis:

- `git stash show` Lista todos os arquivos modificados no ultimo *stash*;
- `git stash clear` Realiza a remoção de todos os *stash's*;
- `git stash create <mensagem>` e `git stash save <mensagem>`  
Ambos criam uma mensagem/nome para o *stash*;
- `git stash pop` Realiza a remoção em ordem de pilha.

<https://medium.com/wooza/git-stash-conhecendo-e-utilizando-um-dos-comandos-mais-pr%C3%A1ticos-para-o-versionamento-de-seu-c%C3%B3digo-a4dab3ac70da>

**Obrigado!!**



<https://www.devmedia.com.br/sistemas-de-controle-de-versao/24574>

<https://learn.microsoft.com/en-us/visualstudio/version-control/git-resolve-conflicts?view=vs-2022>

<https://www.atlassian.com/br/git/tutorials/what-is-git>

<https://www.hostinger.com.br/tutoriais/o-que-github>

<https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging>