git

GitHub

GitLab

O **Git** é um sistema de controle de versão distribuído. Desenvolvido por Linus Torvalds em 2005 e é muito usado para o rastreamento de mudanças no código-fonte em projetos de Software.

Proporciona aos desenvolvedores:

- acompanhamento das alterações realizadas em seu código;
- a reversão para versões anteriores;
- trabalho em ramificações distintas;
- colaboração em equipe;
- fácil integração de mudanças de diferentes desenvolvedores.

O GitHub é uma plataforma de hospedagem de repositórios Git na nuvem. Esta plataforma fornece um serviço online no qual é possível realizar a armazenagem, gestão e colaboração em repositórios Git.

O GitHub oferece:

- recursos de colaboração, como problemas para rastreamento de tarefas e bugs;
- solicitações de pull a fim de propor e revisar modificações no código;
- integração contínua (CI)[1] com GitHub Actions[2].

É usado principalmente para projetos de código aberto, mas pode ser usado para projetos privados (com planos pagos).

1. CI (Integração Contínua): prática de mesclagem frequente de código desenvolvido por diferentes programadores em repositório compartilhado. Mais detalhes nos próximos slides.

2. Plataforma de automação de fluxo de trabalho integrada ao GitHub, para a automação de tarefas e processos de desenvolvimento de software no repositório GitHub.

O GitLab é uma outra plataforma para hospedar repositórios, porém tem maior abrangência em relação a funcionalidades. Além da hospedagem de códigos, esta plataforma oferece ferramentas para gerenciamento de projetos e ciclo de vida de aplicativos.

A plataforma oferece hospedagem gratuita e ilimitada de repositórios Git, assim como ferramentas de gerenciamento de projetos, rastreamento de problemas, integração contínua e entrega contínua (CI/CD) integradas.

Possui versão de código aberto além da versão empresarial hospedada da nuvem.

git 

GitHub 

GitLab

Git x GitHub x GitLab

O Git é o sistema de controle de versão. Enquanto o GitHub e o GitLab são plataformas que fazem uso do Git para hospedagem de repositórios e fornecem recursos adicionais.

GitHub x GitLab

O GitHub é popularmente conhecido devido à sua grande comunidade de código aberto e é muito usado justamente para projetos de código aberto. Já o GitLab tem maior abrangência em relação a funcionalidades, fornecendo ferramentas de gerenciamento de projeto, hospedagem gratuita e ilimitada de repositórios privados.

A escolha da plataforma depende da necessidade de cada projeto em específico e equipe, assim como preferências relacionadas a integrações de códigos e políticas de privacidade. Todas essas ferramentas são muito utilizadas e trabalham com eficácia para controlar versões de projetos de desenvolvimento de Software.

Alguns conceitos...

CI/CD (Continuous Integration/Continuous Delivery):

CI/CD é uma prática para o desenvolvimento de software baseado em automatização de integração e entrega contínua de código em um ambiente de produção, envolvendo a integração frequente de novas alterações de código, automatização da execução de testes e entrega automática de novas versões de sistemas ou serviços aos usuários finais.

Integração Contínua (CI): prática de mesclagem frequente de código desenvolvido por diferentes membros de uma equipe de desenvolvimento em um repositório compartilhado. A cada vez que um código é alterado ou um novo código é integrado, uma série de testes automatizados é executada, garantindo, assim, que as integrações realizadas não impactem negativamente o código já existente.

Entrega Contínua (CD): automatização do processo de implantação de código em ambientes de teste ou de produção após uma integração bem-sucedida e aprovação dos testes. Esta prática permite uma entrega rápida de novas versões de software ao usuário final, mantendo um Sistema estável.

Git Bash

O Git Bash é um emulador de terminal e shell para sistemas Windows, permitindo que os usuários executem comandos Git e outros comandos do Unix diretamente no ambiente Windows. É útil para desenvolvedores que trabalham com o sistema de versão Git e estão acostumados ao uso de um terminal Unix.

Emulador de Terminal: fornece uma interface de linha de comando para digitação de comandos e interação com o sistema de arquivos.

Suporte ao Git: atua juntamente ao Git, portanto é possível clonar repositórios, criar branches, fazer commits, dentre outras ações no Git Bash.

TortoiseGit

Ferramenta de interface gráfica de usuário (GUI) para o sistema de controle de versão Git. Assim como o Git Bash é uma interface de linha de comando para o Git, o TortoiseGit é uma interface gráfica que torma o uso do Git mais fácil e intuitivo em sistemas Windows.

Fornece uma maneira visual de interação com repositórios Git e realização de operações relacionadas ao controle de versão

Algumas funções:

1. **Integração com Windows Explorer:** o TortoiseGit se integra diretamente no Gerenciador de Arquivos do Windows, adicionando ícones e menus contextuais para a realização de ações Git diretamente no sistema de arquivos.
2. **Histórico Visual:** pode-se visualizar o histórico de commits de um repositório Git em uma interface gráfica.
3. **Sincronização:** pode-se sincronizar o repositório local com um repositório remoto (como no GitHub ou GitLab) para buscar as últimas alterações ou enviar as alterações locais.
4. **Resolução de Conflitos:** fornece ferramentas visuais para ajudar a resolver conflitos de merge quando o Git detecta que há alterações conflitantes em diferentes branches.

+ Conceitos Gerais

BRANCH: é uma ramificação de desenvolvimento que deriva de uma branch principal (como a trunk ou master) em um sistema de controle de versão. Cada branch é uma cópia independente do código-fonte do projeto, que permite que o desenvolvedor trabalhe em alterações específicas sem afetar diretamente a versão principal do software.

1. Branch Principal (Master ou Trunk): é a branch central do projeto, que representa a versão estável e funcional do Software. Versão que está pronta para uso em produção.

2. Branches Secundárias (Feature Branches, Bugfix Branches, etc.): são criadas a partir da branch principal e são usadas para desenvolver novas funcionalidades, corrigir bugs ou realização de tarefas específicas de desenvolvimento. Cada branch secundária é isolada das outras, permitindo que os desenvolvedores trabalhem de forma independente nas tarefas.

3. Merge (Mesclagem): ao concluir o trabalho em uma branch secundária, os desenvolvedores podem mesclar (merge –*ou, informalmente, mergear)* suas alterações de volta à branch principal, incorporando as modificações na versão principal do software.

+ Conceitos Gerais

BRANCH: é uma ramificação de desenvolvimento que deriva de uma branch principal (como a trunk ou master) em um sistema de controle de versão. Cada branch é uma cópia independente do código-fonte do projeto, que permite que o desenvolvedor trabalhe em alterações específicas sem afetar diretamente a versão principal do software.

4. Histórico dos Fontes: cada branch mantém seu próprio histórico de desenvolvimento, registrando todos os commits e alterações específicas feitas nessa branch, facilitando o rastreamento das modificações e a identificação de qual desenvolvedor trabalhou naquela tarefa.

5. Tags (etiquetas): após uma mesclagem bem sucedida de uma branch secundária na branch principal, é comum criar tags para marcar versões específicas do software, ajudando a identificar versões estáveis e importantes do projeto ao longo do tempo.

6. Fluxos de Trabalho: há várias estratégias e fluxos de trabalho para o uso de branches em projetos de Software, incluindo o Git Flow que será abordado posteriormente.

+ Conceitos Gerais

TRUNK: refere-se a uma das principais branches (ramificações) de desenvolvimento no sistema de controle de versão Git. A branch "trunk" também é conhecida como "branch principal" ou "branch mestre (master)".

É importante observar que o uso do termo "trunk" e a estratégia de desenvolvimento em branches podem variar entre diferentes equipes e projetos. Algumas equipes podem preferir usar um nome diferente, como "main" ou "master", em vez de "trunk", para denotar a branch principal. Além disso, a adoção de práticas como Git Flow pode influenciar na maneira em que as branches são gerenciadas e nomeadas.

CHAVE SSH: chave criptográfica do tipo SSH usada para autenticar e estabelecer conexões seguras entre o computador local do desenvolvedor e um servidor GitLab ou repositório Git remoto. As chaves SSH são usadas para dar garantia da segurança das transações e comunicações entre o sistema local e o servidor remoto.

É útil para operações como clonar repositórios, fazer push e pull de código e interagir com os recursos do GitLab.

Para usar o TortoiseGit com repositórios Git que requerem autenticação SSH, é necessário configurar chaves SSH no Windows. Envolvendo a geração de um par de chaves SSH: uma chave pública que é compartilhada com o servidor GitLab e uma chave privada que é mantida no sistema local. É necessário fornecer a chave pública ao GitLab e garantir que a chave privada esteja devidamente configurada no TortoiseGit para que ele possa usá-la para autenticação ao acessar repositórios GitLab.

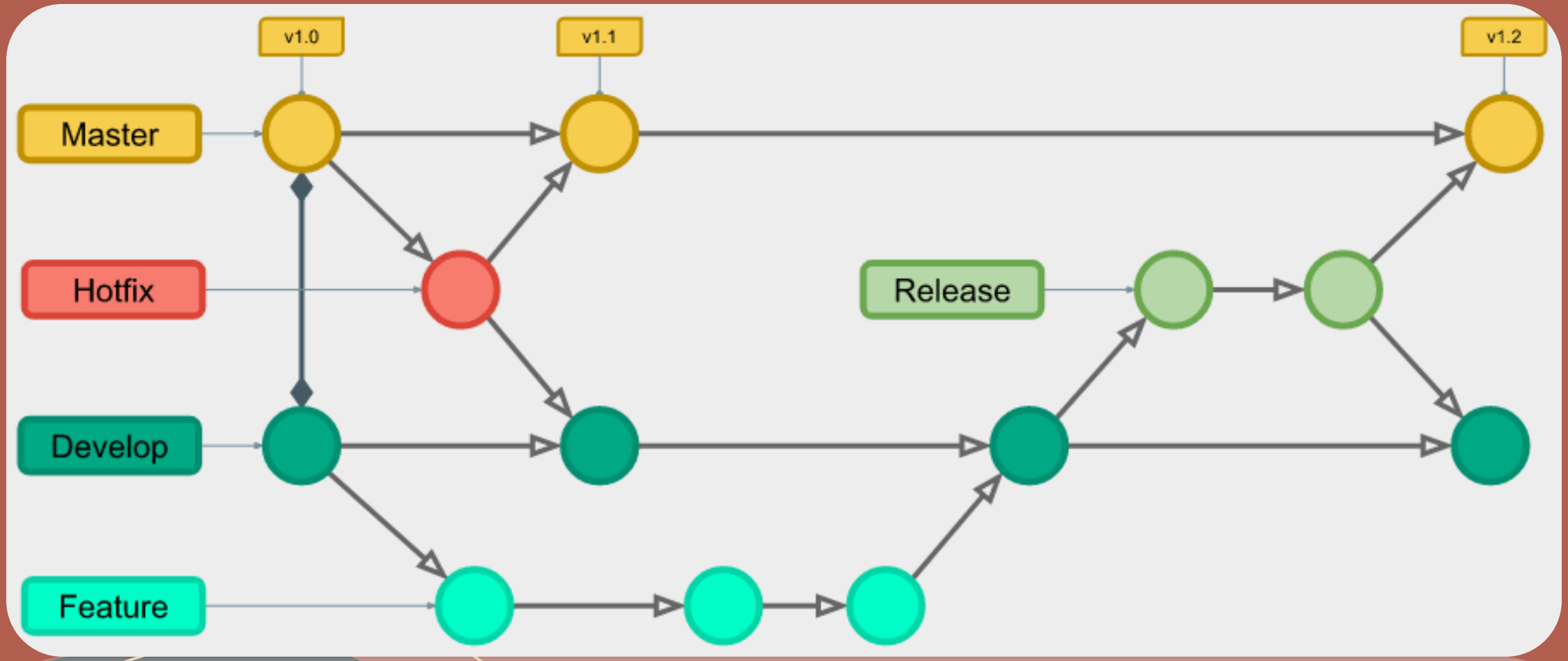
Git Flow

O Git Flow é uma estratégia de fluxo de trabalho muito utilizada entre equipes de desenvolvimento de software. Este fluxo proporciona uma melhora nas organizações das branches dentro dos repositórios e, desta forma, aprimora a fluidez ao processo de desenvolvimento das novas versões, correções de bugs e funcionalidades do Sistema.

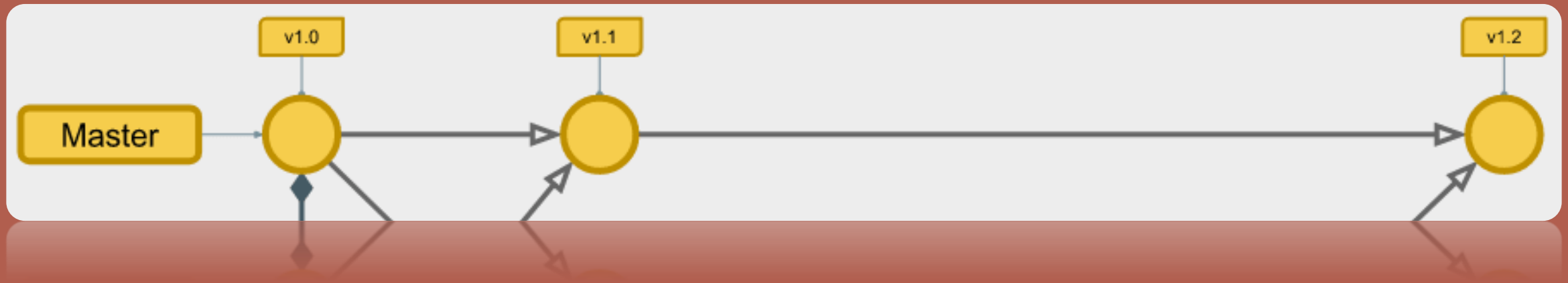
O Git Flow atua com as branches **develop** e **master** como as principais, além de três branches de apoio: **release, feature** e **hotfix** que são branches temporários.

As duas branches principais são usadas para registrar o histórico do projeto. Na **master** está o histórico do lançamento do projeto e a **develop** funciona como uma ramificação de integração para recursos.

Git Flow



Git Flow

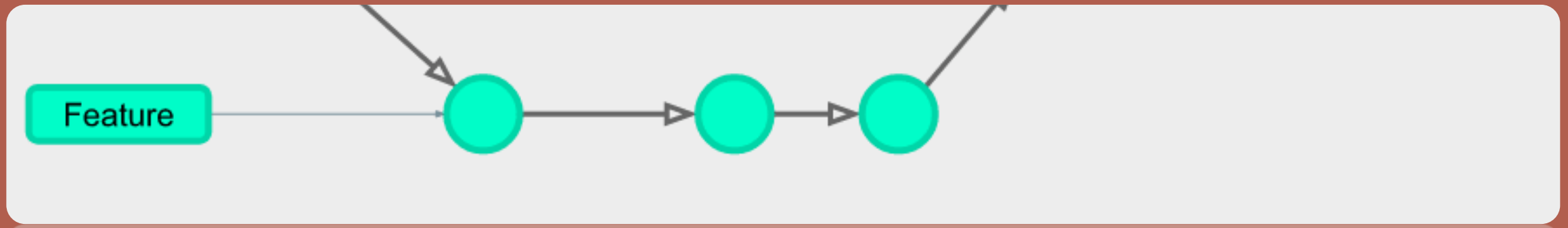


A branch master contém o código pronto para produção que pode ser liberado. É criada no início do projeto e é mantida em todo o processo de desenvolvimento. Pode ser marcada em vários commits para significar diferentes versões ou lançamentos e outras branches serão mescladas na master após serem suficientemente testadas.

Git Flow

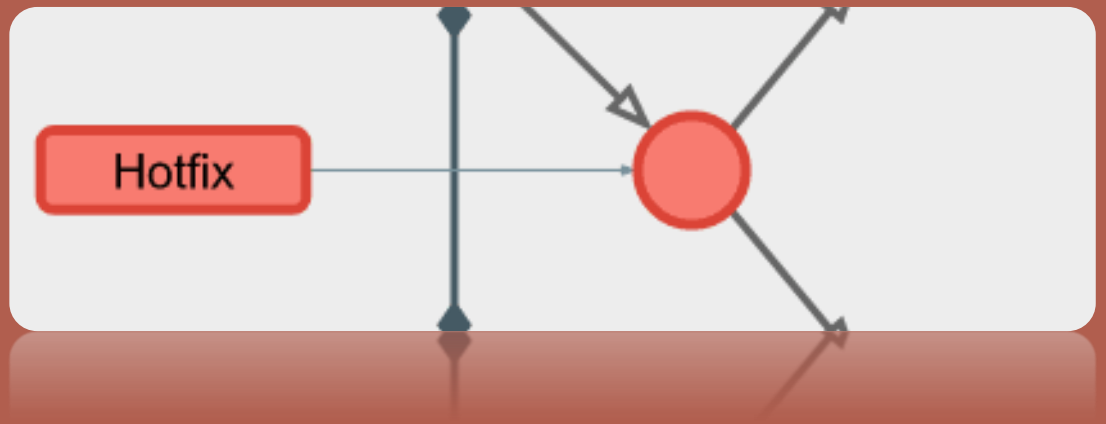
A branch develop contém o código de pré-produção com os desenvolvimentos recém realizados que estão em processo de testagem. Quando os testes são concluídos, são mesclados na master. É criada no início do projeto e é mantida em todo o processo de desenvolvimento.

Git Flow



A branch feature contém o código trabalhado na correção de pequenos bugs ou retoques finais específicos para liberação de um novo código, que deve ser tratado separadamente da ramificação de desenvolvimento principal. Desenvolvimento de recursos para os próximos releases.

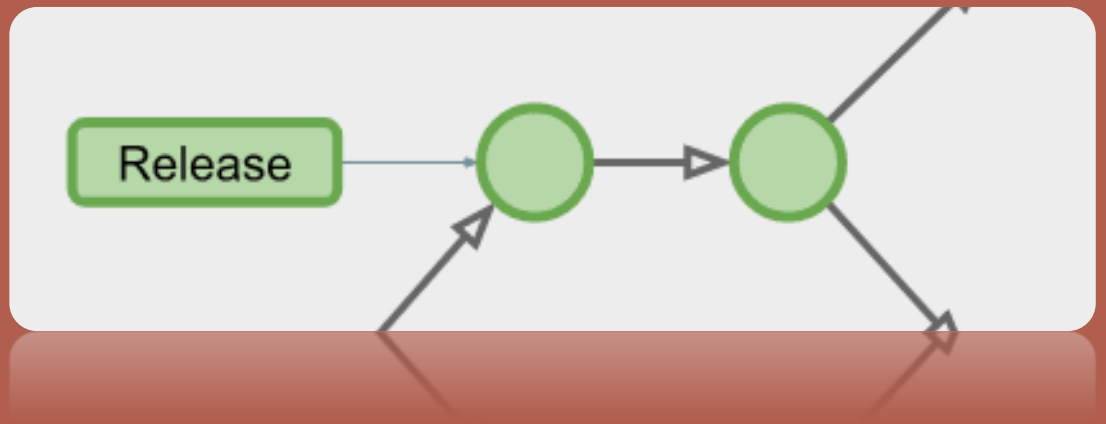
Git Flow



A branch hotfix é usada para abordagem rápida de alterações necessárias na master. A base do hotfix deve ser a master/main e deve ser mergeada de volta para a master e a develop.

A mesclagem do hotfix de volta na develop é essencial para que a correção feita persista no próximo lançamento da master.

Git Flow



A branch release é usada para o preparo de novas versões de produção. Caso sejam adicionados todos os recursos necessários, tornando o projeto pronto para ser lançado, mas for encontrado algum bug, é possível resolver diretamente no mesmo branch, depois mergear as alterações de volta para a develop e, por fim, enviar para a master.

Principais Comandos

git init: Esse comando é responsável por inicializar um novo repositório. Em um processo de versionamento de código, esse é o primeiro comando básico.

git config: O comando git config é utilizado para configurar as opções de instalação e de usuário do git.

git add: Este é responsável por adicionar supostas mudanças no diretório do projeto à área de staging, dando uma oportunidade de preparar a snapshot antes de submeter o arquivo ao projeto oficial, ou seja, antes de fazer o commit. (“*git add .” adiciona todas as mudanças disponíveis de uma só vez*).

git status: Verifica o estado da área de staging.

git commit: Para submeter as mudanças inclusas no snapshot da área de staging. Ex.: *git commit -am “Alteração do método de inserção de cadastro de pacientes“*. Onde a flag **-am** indica a inserção de uma mensagem para identificação da alteração.

git fetch: Faz o download a partir de uma branche de outro repositório com os outros commits e arquivos associados. Mas não integra nada no repositório local, portanto o desenvolvedor pode inspencionar as alterações antes de mergeá-las para o projeto.

Principais Comandos

git pull: é a versão automatizada do git fetch. Baixa um branche de um repositório remoto e faz o merge imediatamente no branch atual.

git push: fazer o push (empurrar) é o oposto do fetch. Transfere commits de um repositório local para um repositório remoto.

Arquivos .gitignore

Este tipo de arquivo serve para o Git saber quais arquivos deverão ser ignorados na hora de realizar um commit. Dessa forma se agiliza o trabalho do usuário que não precisa ficar tirando de todo commit arquivos indesejados como, por exemplo, senhas, arquivos de configurações de IDE pessoais, etc.

Para utilizar esse tipo de arquivo, é necessário que após tê-lo criado e colocado dentro do repositório local se escreva dentro dele quais arquivos serão ignorados. Um exemplo de arquivo .gitignore é o seguinte:

senhas.txt
iniciarExe.bat
testes/

Existem, também, os caracteres coringas, que facilitam a seleção de arquivos para o .gitignore:

*	Substitui qualquer coisa
?	Substitui apenas um caractere
[]	Para intervalos
!	Para negação
#	Indica um comentário

```
#Ignora todos os arquivos .txt
*.txt
#Ignora os arquivos que começam com erro, tendo qualquer
   carácter a frente e sendo .log
erro?.log
```

Ferramentas de configuração (define quem fez as atualizações)

$ git config –global user.name "[nome/nick do desenvolvedor]"
Define o nome ou o apelido de quem realiza as alterações, o nome é enviado ao GIT e descreve quem realizou a alteração de maneira simples.
$ git config –global user.email "[e-mail do desenvolvedor]"
Define o e-mail de quem realiza as alterações, o e-mail é enviado ao GIT e descreve quem realizou a alteração de maneira simples.
$ git config –global color.ui auto
As interações por linha de comando ficam coloridas com esse comando.

Criar Repositórios (novo repositório ou obtém de um já existente)

$ git init [nome do projeto]
Cria um novo repositório local com um nome já definido
$ git clone [url]
Baixa um projeto e os históricos de versões

Fazendo atualizações para entregar uma funcionalidade

$ git branch
Lista todos os branches (termo para definir versões funcionais)
$ git branch [nome de uma nova branch]
Cria uma nova branch com o nome informado
$ git checkout [nome de uma branch existente]
Altera conteúdo do local de trabalho para o snapshot da branch informada
$ git merge [nome de uma branch existente]
Combina a branch informada com a branch atual do git.
$ git branch -d [nome de uma branch existente]
Remove a branch informada

Fazendo atualização pontual (revisão de alterações commit manual)

$ git status
Lista todas as alterações que precisam ser comitadas
$ git diff
Apresenta as diferenças entre arquivos que não estão staged
$ git add [files]
Adiciona o arquivo para o grupo (staged) que será comitado
$ git diff –staged
Apresenta as diferenças entre os arquivos staging e a ultima versão.
$ git reset [file]
Remove arquivo do grupo (staged) comitado mas preserva conteúdo.
$ git commit -m "[descrição da alteração]"
Gera um snapshot que pode ser utilizado futuramente para roolback por exemplo. Isso não gera uma versão apenas um snapshot, isso é importante para que as alterações sejam documentadas. Essa operação é realizada apenas localmente.

Remover arquivos

$ git rm [file]
Remove o arquivo do local de trabalho e do stage
$ git rm –cached [file]
Remove o arquivo do stage mas não remove do local de trabalho
$ git mv [nome do arquivo original] [novo nome do arquivo]
Altera o nome do arquivo local e na area do stage, pronto para comitar.

Removendo arquivos e diretórios de commit acidental/indevido

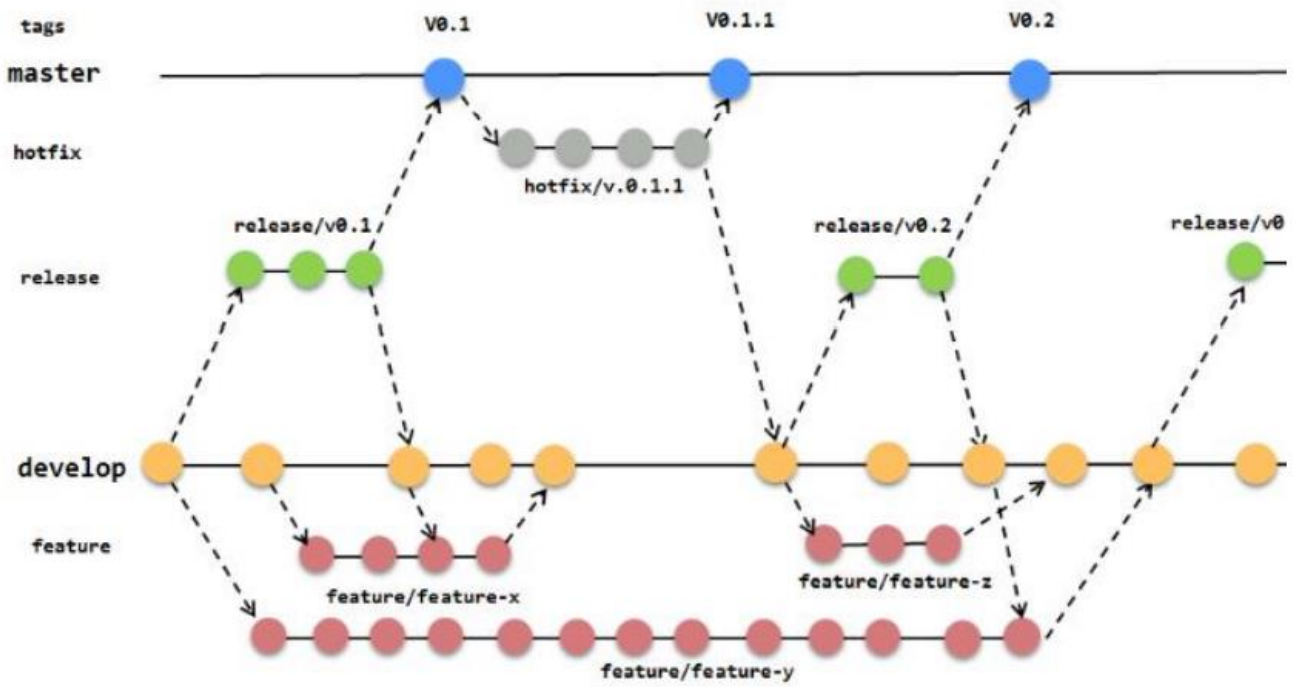
*.log
build/
temp-*
Esse deve ser o conteúdo do .gitignore, nesse arquivo ficam todos os diretórios, arquivos ou padrões de arquivos que não devem ser comitados.
$ git ls-files –other –ignored –exclude-standard
Lista todos os arquivos ignorados nesse projeto

Como salvar alterações incompletas sem comitar

$ git stash
Seria como um commit temporário não oficial.
$ git stash pop
Recupera os arquivos que foram salvo pelo stash
$ git stash list
Lista todos os stash realizados e não recuperados
$ git stash drop
Descarta os stash mais recente realizado.

Histórico

$ git log
Lista o histórico das versões da branch atual
$ git log –follow [file]
Lista histórico de versão de arquivo, incluindo se também foi renomeado
$ git diff [nome da primeira branch] [nome da segunda branch]
Apresenta a diferença entre as duas branch
$ git show [nome ou código de um commit]
Apresenta todos os detalhes as mudanças de um commit especifico



Rollback dos commits realizados

$ git reset [nome do commit_1]
Desfaz todos os commits após o commit_1, mantendo as alterações locais
$ git reset –hard [nome do commit_1]
Descarta histórico de mudanças após o commit_1 e atualiza a área de trabalho com a versão do commit_1

Atualizando o servidor com os commits locais

$ git fetch [geralmente utilizam o nome “origin”]
Faz download do histórico do repositório, na imagem seria do “origin”
$ git merge [geralmente utiliza o nome “origin”]/[nome de uma branch]
Combina o branch do origin com a branch local
$ git push [geralmente o nome “origin”] [o nome da branch remota]
Faz upload de todo o conteúdo local para o servidor do Git
$ git pull
Faz o download de todo histórico do repositório incluindo as mudanças

Na Prática

Antes de iniciar a manutenção/incremento no código, é necessário criar uma nova branch específica para esta tarefa, para que posteriormente esta seja integrada à branch develop (passando por testes em homologação e, após isso, seja feito o merge da develop com a master).

Chamando o git bash na pasta DEVELOP, dentro do projeto, serão utilizados os comandos abaixo:

git checkout –b ***nome da branch*** O nome da branch é de escolha do usuário/projeto (geralmente um número de tarefa associada à nova implementação).

Após este comando, será criada uma nova branch local. A partir de então o desenvolvedor poderá começar a realizar suas alterações nos fontes do projeto.

Após as alterações/implementos terem sido realizados, o desenvolvedor pode verificar se o git está enxergando as alterações realizadas, com o comando abaixo no git bash:

git status *Este comando apresentará os arquivos alterados em vermelho, estes precisam ser adicionados à área de staging.*

git add . *Este comando adiciona todos os arquivos alterados para a área de staging, então o usuário pode dar um git status.*

git status *Este comando apresentará os arquivos alterados em verde, pois já foram adicionados à área de staging, estão prontos para serem commitados.*

Na Prática

Após o git status, é necessário realizar o merge das alterações, para que a branch local as enxerge como novas versões:

git commit -m "Descrição" *Este comando realiza o commit e o "-m" permite que o desenvolvedor adicione uma descrição.*

Após o git commit, é necessário realizar o git push, para que as alterações da branch local subam para a branch remota.

git push origin Nome_da_branch_criada

Ex.: git push origin clinica001

Após isso, deverá acessar o projeto no GitLab. Será exibida automaticamente uma notificação de merge.
Criar um novo merge request, com a origem da branch criada e o destino develop.

Assignee: nome do desenvolvedor

Criar merge request.

Na Prática

Na página do merge request, pode-se visualizar as alterações realizadas e, por fim, realizar o merge da branch criada para a develop.

Conceitualmente o que está na develop só é integrado à master após os testes em homologação do Software. Para isso, basta criar um novo merge request, com origem na develop, destino à master/main e aplicar o merge.