Git
Github

Ola!!

Me chamo Cláudia, tenho 25 anos, moro em Campo Mourão - PR.

Sou formada em Ciência da Computação pela UTFPR, e estou cursando MBA em Gestão de Projetos pela USP/Esalq (e guitarrista nas horas vagas 😅).

Atuo profissionalmente como desenvolvedora front-end/mobile a mais de 4 anos, atualmente na empresa MetaIT, trabalhando como terceirizada para o grupo renner

Linkedin: linkedin.com/in/cláudia-sampedro-a05514170
E-mail: clp.sampedro@gmail.com

1 Versionamento

O que é e importância

Contexto

A área de desenvolvimento de softwares está em constante crescimento, e por consequência temos mais profissionais atuantes. Se pensarmos no contexto de um “time de desenvolvimento”, ou seja, um grupo de pessoas que atuam diariamente na construção de um mesmo software, logo vamos nos perguntar: “como essas pessoas sincronizam seus trabalhos?”, ou então “como irei controlar as versões deste software?”.

Sistemas de controle de versão

Sempre que os desenvolvedores criam um novo projeto eles continuam criando atualizações no código base. Mesmo depois de o projeto ser lançado é comum a atualização de versões, correção de bugs, adição de novas ferramentas, etc.

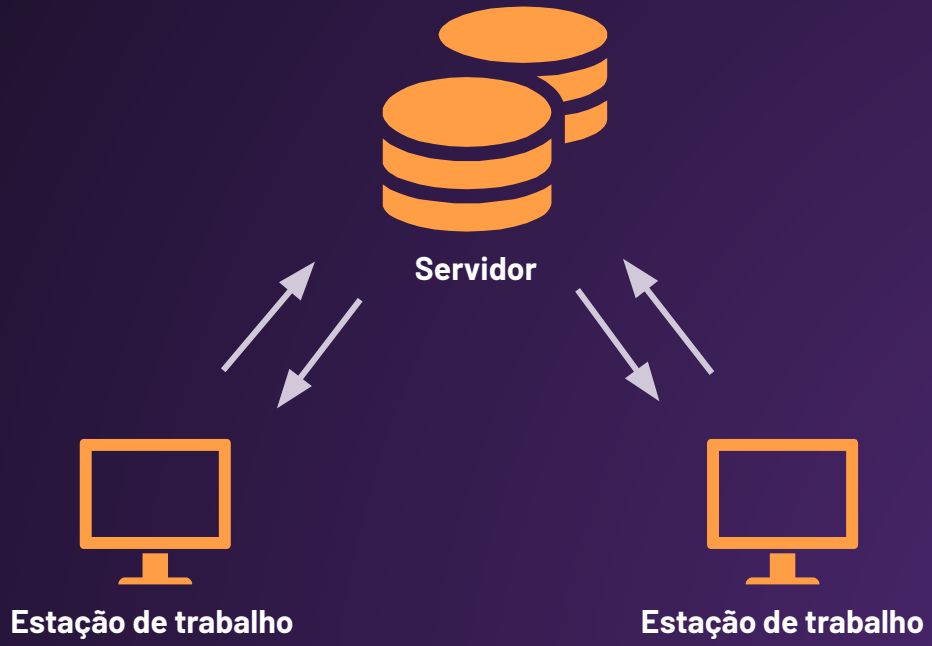
O sistema de controle de versão ajuda a acompanhar as mudanças feitas no código base. E mais, ele também registra quem efetuou a mudança e permite a restauração do código removido ou modificado.

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 última 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.

Exemplo



Como funciona?

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

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

Tipos de sistemas de controle de versão

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

Já o **distribuído,** trabalha com "diversas cópias"do projeto: cada pessoa tem uma cópia local, e o envio para o servidor central ocorre através de operações de *pull* e *push*.

Tipos de sistemas de controle de versão

Resumidamente os sistemas centralizados são mais simples de usar, mas não fornecem ferramentas mais robustas para resolução de conflitos, por exemplo.

Já os distribuídos, apesar de exigirem maior conhecimento da ferramenta, permite a resolução de problemas complexos de controle de versão

Git

O Git é um sistema de controle de versão distribuído e amplamente adotado. Seu criador principal é o mesmo que o do Linux: Linus Torvalds, e ganhou o coração das pessoas que trabalham com open source.

O Git funciona de forma distribuída: cada pessoa possui uma cópia do projeto em sua máquina, e o projeto original pode ser mantido na nuvem, para outras pessoas acessarem.

Git

O Git pode ser usado em todo e qualquer projeto que tenha arquivos de diferentes tipos, podendo ser código, texto, imagens, vídeos, áudios, entre outros. O objetivo principal é permitir o controle de histórico e versão desses projetos, melhorar o trabalho em time e o fluxo de trabalho, proporcionar a segurança dos seus arquivos e outras tantas vantagens faladas nesse post.

Github e Gitlab

GitHub e o Gitlab são plataformas para gerenciar seu código e criar um ambiente de colaboração entre devs, utilizando o Git como sistema de controle. Eles vão facilitar o uso do Git, escondendo alguns detalhes mais complicados de setup. É lá que você provavelmente vai ter seu repositório e usar no dia a dia.

O sistema web que eles possuem permite que você altere arquivos lá mesmo, apesar de não ser muito aconselhado, pois você não terá um editor, um ambiente de desenvolvimento e testes. Ambos serviços possuem versão gratuita e paga.

Instalando o Git

- *Windows*
- *Linux/MacOS*

2 Comandos e GitHub

Verificando se o git está instalado

Podemos utilizar o git apenas por linha de comandos. Primeiramente vamos verificar se temos o Git instalado nas máquinas (no Linux ele vem instalado por padrão, e no windows basta você fazer o download no próprio site do git)

Abra o terminal e digite: **git —version**

Git –help

Um dos comandos mais úteis do git é o: **git –help**

Com ele, teremos uma lista de comandos disponíveis para uso, e quais são suas funcionalidade, eliminando a necessidade de decorar todos os comandos

GitHub

Acessem o site https://github.com/ e criem uma conta

Git –config

Após, devemos configurar nossa conta no nosso computador, para que o git reconheça nossa conta do Github. Digite os comandos:

git config –global user.name “seu nome de usuário”

git config –global user.email “seu email”

Git 2FA

Hoje em dia é bem importante configurarmos autenticação por 2 fatores sempre que possível. O GitHub está gradualmente exigindo que seus usuários habilitem essa opção.

Git SSH

Podemos “interagir” com o GitHub utilizando 2 protocolos diferentes:

- HTTP
- SSH

O SSH tem a vantagem de ser mais seguro por usar estratégia de chaves pública/privada e criptografia

Git SSH

Utilizando o SSH vamos precisar gerar uma chave privada e adicionar ao agente SSH.

A chave pública será adicionada a sua conta github.

Fluxo: seu agente SSH vai conhecer sua chave privada, e o github a chave pública, assim, o github irá “codificar a mensagem” usando sua chave pública, e apenas você poderá “decodificar” usando a chave privada.

Git SSH

- Gerar a chave

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

- Adicionar ao agente SSH (pular no windows)

```
eval "$(ssh-agent -s)"
```

- Copiar chave SSH

```
pbcopy < ~/.ssh/id_ed25519.pub
```

Git SSH

Adicionar a chave no GitHub

Repositórios

- O que são
- Como criar
- Diferença entre repositórios públicos e privados
- Arquivo README
  - Linguagem de marcação Markdown
    - Headers
    - Itálico, negrito, sublinhado
    - Imagens
    - Adição de código

Clonando o repositório no PC

- Git clone
- Git init
  - git remote add origin urlQueCopiaramNoGithub

Adicionando mudanças no repositório

- Criando arquivos
- Adicionando códigos

Git status

- O comando **git status** nos ajuda a entender o estado do nosso repositório: arquivos que ainda não foram para o servidor, qual branch estamos, etc

Commit

- O commit consiste em um “pacotinho” que irá encapsular as modificações de um arquivo do projeto, para criar um marco no repositório e quando necessário enviar esse “pacotinho” para produção.

Commit

- Para adicionarmos um arquivo ao pacotinho do commit, devemos utilizar o comando chamado **git add.**
- Podemos adicionar um único arquivo modificado no commit, ou todos os arquivos modificados.
- Para adicionar um único arquivo modificado, utilizamos: **git add nomeDoArquivo**
- Para adicionarmos todas as modificações, utilizamos: **git add .**

Commit

Após utilizar o comando **git add**, podemos executar novamente o comando **git status** para vermos o que foi alterado no nosso repositório.

Agora o **git status** nos diz que há uma mudança para ser “commitada”. Voltando a nossa analogia do “pacotinho”, nesse momento estamos embalando as modificações para irem no pacote. Esse pacote depois será despachado para o servidor

Commit

Agora é hora de criarmos o nosso pacotinho de fato, o commit. Todo commit pode carregar uma mensagem com ele, a fim de identificar as modificações que estão no pacote.

Para tal, utilizamos o seguinte comando: **git commit -m “sua mensagem aqui”**

Conventional Commits

É importante utilizarmos o git commit apropriadamente. Dispor de uma linguagem coerente e padronizada ajuda a todos os envolvidos no projeto a entenderem as mudanças ocorridas e quais contextos foram afetados.

Os commits são pontos na linha do tempo de um projeto. Quando documentados propriamente nos mostram quem alterou, quando, em qual contexto e qual tipo de alteração foi feita. Diante disso, vamos conhecer o Conventional Commits Pattern.

Conventional Commits

Vantagens de uso:

- Geração automática de changelog
- Facilidade no controle das versões através de *semantic version*
- Padronização
- Torna a busca por commits específicos menos morosa

Conventional Commits

O Conventional Commits é uma convenção simples de mensagens de commit, que segue um conjunto de regras e que ajuda os projetos a terem um histórico de commit explícito e bem estruturado.

As regras são muito simples, temos um **tipo de commit** (type), o **escopo/contexto do commit** (scope) e o **assunto/mensagem** do commit (subject).

Conventional Commits

Exemplo:

git commit -m “refactor(components): change style on input component”

- A parte do assunto do commit (change style...), deve ser sempre escrita no infinitivo. A ideia é dizer o que o commit irá realizar.
- O escopo (**components**) é opcional, e diz a qual escopo do projeto esse commit pertence.
- Os tipos são padronizados, podendo ser:

Conventional Commits

- **test**: qualquer criação ou alteração de códigos de teste.
- **feat**: desenvolvimento de uma nova feature ao projeto.
- **refactor**: usado quando houver uma refatoração de código
- **style**: mudanças de formatação e estilo do código
- **fix**: correção de bugs
- **chore**: configurações de setup
- **docs**: adição ou alteração de documentação
- **build**: mudanças no processo de build
- **perf**: melhora de performance
- **ci**: mudanças nos arquivos de CI
- **revert**: reversão de commit

Praticando conventional commits

Conventional Commits

- Através de algumas bibliotecas, nós podemos forçar os commits a seguirem os padrões do conventional commits.
- Um exemplo é a biblioteca para projetos node *commitlint* https://commitlint.js.org/#/

Exemplo utilizando a lib commitlint

Commits atômicos

Uma das principais definições da metodologia é de que as tarefas devem ser divididas em átomos: atomic commits, que se consistem de unidades objetivas que implementam uma única funcionalidade, ou tratam a correção de um único erro.

Ou seja, a cada correção ou implantação individual se deve executar um processo de commit, algo que segue outro conceito do **Clean Architecture**, o **Single Responsiblity Principle** do **SOLID**.

Git push

Com o pacotinho pronto, é hora de enviá-lo para o servidor, para o nosso repositório ter as modificações que fizemos na nossa máquina. Para tal, utilizamos o seguinte comando: **git push**

Mais pra frente vamos voltar ao assunto de branchs, mas na primeira vez que formos executar o push, o próprio terminal nos dirá para executar um pouco mais complexo:

git push —set-upstream origin master

Visualizando nosso commit no GitHub

Git log

O comando git log é utilizado para visualizar os logs de um repositório: commits, modificações, autores, etc.

Esse comando apresenta uma vasta lista de opções para formatação, visualização, filtros e ordenação.

Git log

- **git log - -oneline** (formata em 1 linha)
- **—stat** (traz o numero de inserções)
- **-p** (traz o diff completo)
- **git shortlog** (agrupa por desenvolvedor)
- **—graph** (plota um grafo das branchs)
- **—follow** (mostra os commits de um mesmo arquivo mesmo se ele for renomeado)
- **—log-size**
- **—reverse**
- **—max-count=<number>**

Git log

- **—pretty**
  - **=format:""**
    - **%cn** (commiter name)
    - **%cd** (commiter date)
    - **%ce** (commiter email)
    - **%an** (author name)
    - **%h** (commit hash)
    - **%n new line**

Git log

- –pretty
  - =short
  - =full
  - =medium

Git log

- Filtros:
  - **git log -3**
  - **git log - -after ou - -before data (YYYY-MM-DD)**
  - **—author** (pode passar o \| para mais de um)
  - **—grep (msg de commit)**
  - por arquivo: **git log - -nomeArquivo**
  - **-S** por conteúdo
  - **range entre branchs: ..**
  - **—no-merges**
  - **—merges**

Restauração

Restauração

- Git checkout
  - O comando git checkout é comumente utilizado para trocar de branch. Caso utilizado com a flag -b, ele irá criar uma branch com o respectivo nome passado, caso essa não exista. Exemplo: git checkout -b novaBranch

  - Ele também pode ser utilizado para descartar alterações de um arquivo, através do uso do operador - -. Exemplo: git checkout - - index.html

Restauração

- Git restore
  - Restaura para um ponto do projeto, seja uma branch ou um commit específico. Exemplo: git restore –-source hashDoCommit

Restauração

- Git revert
  - Reverte um commit ou merge. Devemos passar o que devemos reverter.
    - HEAD: ultimo commit
    - HEAD~numero: reverte o número especificado de commits
    - Hash do commit
    - Passando intervalos:
      - git revert -n master~5..master~2
      - Git revert -n f44db3..f167fc

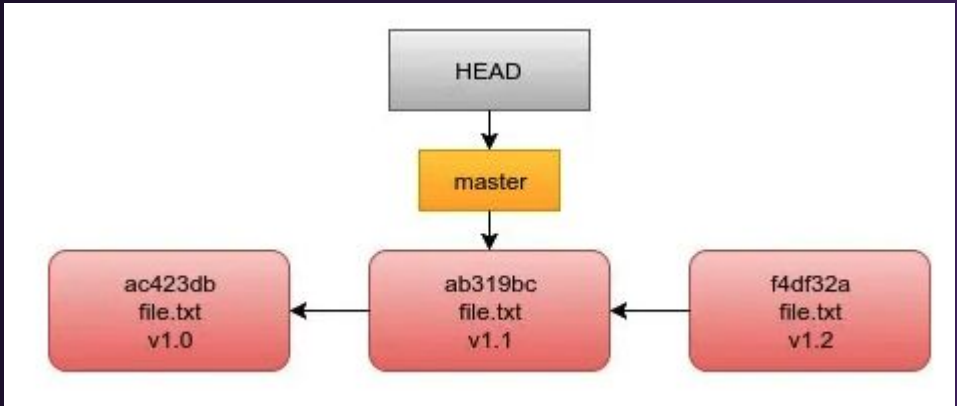
Git reset

Primeiramente alguns conceitos:

- O ponteiro HEAD aponta para o último grupo de alterações(snapshot) comitado. Podemos alterar para qual commit o HEAD aponta executando os comandos commit ou reset.
- A área de index, também chamada área de stage, contém o próximo snapshot a ser comitado. Alteramos a área de index quando executamos o comando add.
- Por fim, o diretório de working contém as alterações que ainda não foram adicionadas à área de stage.

Git reset –soft

- O primeiro tipo de reset é utilizado com a opção soft e move apenas o ponteiro HEAD para algum outro commit, sem alterar a área de stage ou o diretório de working. É importante notar que, de fato, a operação moverá o branch para o qual o HEAD aponta e, por consequência, moverá também o ponteiro HEAD.



Git reset –mixed

O segundo tipo de reset pode ser utilizado com a opção mixed ou, por ser o tipo default, somente com o comando reset. Essa opção, além de mover o ponteiro HEAD, faz com que a área de stage contenha o mesmo snapshot do commit para o qual o ponteiro HEAD foi movido, porém não afeta o diretório de working

Git reset –hard

O terceiro e mais perigoso tipo de reset é utilizado com a opção hard e não apenas descarta as alterações na área de stage como também reverte todas as alterações no diretório de working para o estado do commit que foi especificado no comando.

Git rm

Comando para excluir o arquivo de um repositório

Após a exclusão, a mudança deverá ser commitada

Exemplo: **git rm nomeArquivo**

Removendo o git de uma pasta

Para tal, basta remover a pasta oculta .git do seu projeto

Removendo o origin do repositório

Caso você deseje remover o link do GitHub/GitLab do seu repositório, basta disparar o comando **git remote rm origin**

Caso você deseje alterar a URL remote, basta digitar: **git remote set-URL origin "new URL"**

3 Branchs e git flow

Git flow

Se você utiliza o Git como ferramenta de controle de versão para seus softwares, provavelmente já deve ter observado as várias maneiras de como controlar branches de repositórios.

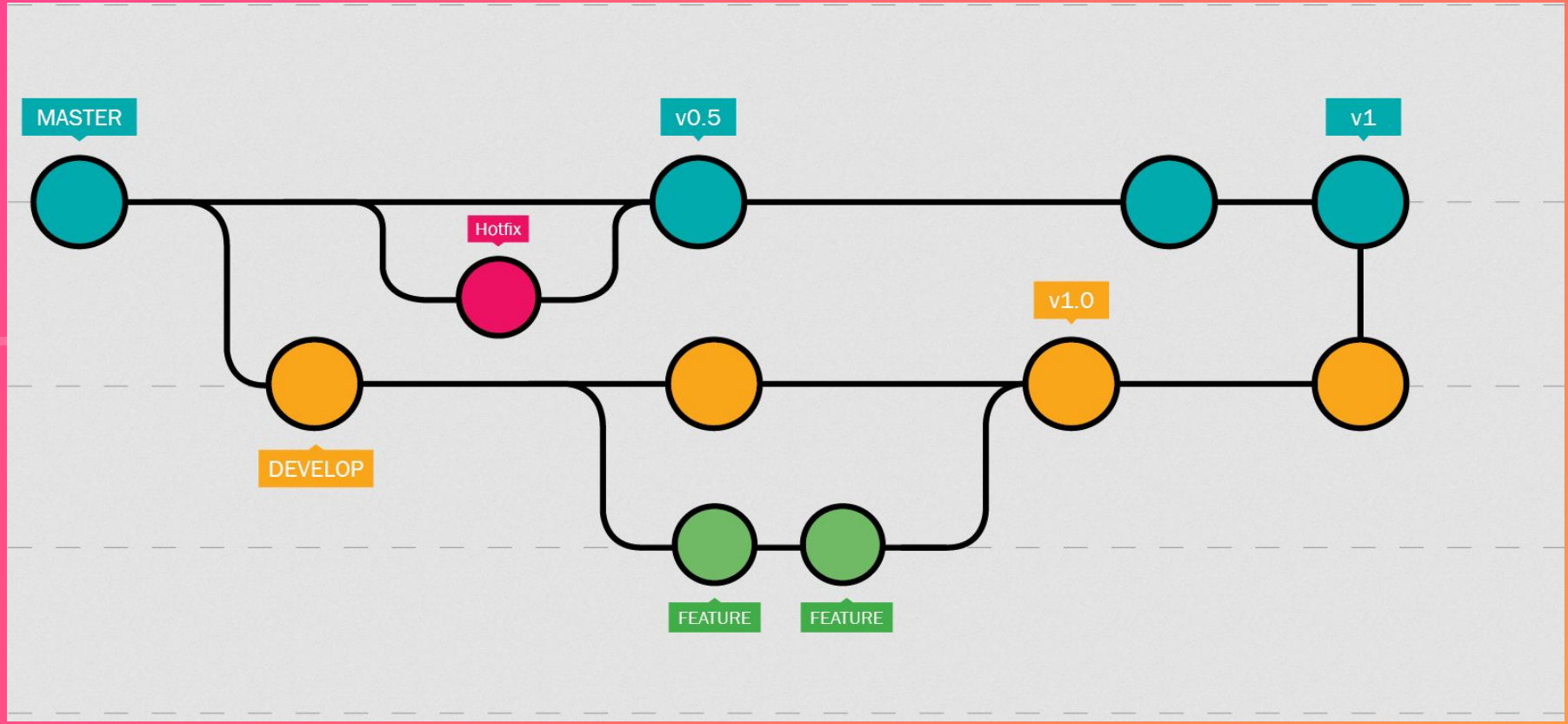
É corriqueiro pessoas utilizarem apenas uma branch para fazer commits em projetos pessoais, o que não é errado, pois quando estamos trabalhando sozinhos é muito tranquilo de se controlar tudo em um branch só.

Git flow

Mas afinal, o que são branchs?

Branchs, ou traduzindo ramos, funcionam como uma espécie de cópia paralela do projeto. Utilizando o esquema de branchs, conseguimos garantir diversas versões independentes do mesmo projeto

Isso é muito válido quando falamos de um time de desenvolvimento: cada pessoa irá implementar sua tarefa em uma branch própria, de forma a não afetar o projeto inteiro, antes da atividade ser devidamente revisada e testada.


MASTER
v0.5
v1
Hotfix
v1.0
DEVELOP
FEATURE
FEATURE

Entendendo branchs

https://git-school.github.io/visualizing-git/

- Criar outra branch a partir da develop
  - Realizar 2 commits nessa nova branch (alterar alguma coisa no código). Fazer o push
  - Voltar para a develop e criar uma outra branch
    - Realizar mais 2 commits nessa nova branch

Criando branchs na prática

- git checkout -b nomeBranch
- Utilizando VSCode
- Visualizando branchs no GitHub
  - Criar branch
  - Commit
  - Checkout
  - Git Graph

Criando branchs na prática

- git checkout -b nomeBranch
- Utilizando VSCode
- Visualizando branchs no GitHub
  - Criar branch
  - Commit
  - Checkout
  - Git Graph

Git flow

Utilizando na prática

Git Flow

O Git Flow trabalha com duas branches principais, a Develop e a Master, que duram para sempre; e três branches de suporte, Feature, Release e Hotfix, que são temporários e duram até realizar o merge com as branches principais.

Então, ao invés de uma única branch Master, esse fluxo de trabalho utiliza duas branches principais para registrar o histórico do projeto. A branch Master armazena o histórico do lançamento oficial, e a branch Develop serve como uma ramificação de integração para recursos.

Merge/pull request

O **pull request** (ou **merge request** no GitLab), consiste na solicitação para mescla de branchs.

O que isso quer dizer? Vamos supor que você deseja anexar o código da sua branch em outra branch. Esse anexo ocorre via merge request.

Você pode fazer o merge de qualquer branch para qualquer branch. Entretanto, isso pode gerar alguns problemas de conflito ou substituição indesejada de código. Mais pra frente voltaremos nesse assunto

Merge/pull request

- Realizando o pull request no GitHub
- Visualizando a mescla dos código

Git pull

- É normal precisarmos pegar as mudanças que estão no servidor para nosso computador. Ou até mesmo de outras branchs para a nossa.
- Para tal, utilizamos o comando **git pull**
- Exemplos:
    - Atualização da branch local
    - Pull de uma branch externa

Conflitos =(

- Algumas vezes quando vamos fazer o pull ou o merge request, nossas branchs apresentam conflitos. O que isso significa? Um código foi modificado de 2 maneiras diferentes e o git não sabe como lidar, ou seja, não sabe qual código deve ser aceito.
- Para resolvermos, temos algumas estratégia:
    - Git pull
    - Rebase
    - Git merge

Conflitos =(

- Como evitar conflitos?
  - SEMPRE mantenha a sua branch local atualizada com a do servidor
  - No momento de quebra das tarefas, preste atenção se suas tarefas paralelas não afetarão os mesmo arquivos
  - Caso necessário, atualize sua branch com a branch do coleguinha

Conflitos =(

- Diferença entre os métodos de resolução de conflitos
  - Criando conflitos
  - Resolvendo conflitos pelos 3 métodos
    - Pull
    - Merge
    - Rebase

4 Semantic version

Semantic version

De forma objetiva o versionamento semântico ou Semantic Versioning é um conjunto de regras e requerimentos para atribuição de versão de software. Em outras palavras, é uma tentativa de resolução de um problema antigo conhecido como “inferno de dependências” ou dependency hell que nada mais é do que problemas e complicações ao lidar com pacotes de softwares.

Esse problema ocorre quando uma versão de software prejudica a segurança e integridade de uma nova versão impedindo assim que o projeto prossiga.

Semantic version

O versionamento semântico garante um controle da versão do código utilizando um grupo de números mantendo assim sua compatibilidade e integridade nas novas publicações conforme exemplo:

Versionamento semântico 1.0.0

Semantic version

- 1 (Major) – controle de compatibilidade. Informa que existem funcionalidades/códigos incompatíveis com as versões anteriores.
- 0 (Minor) – controle de funcionalidade. Informa que novas funcionalidades foram adicionadas ao código.
- 0 (Patch) – controle de correção de bugs. Informa que um ou mais erros foram identificados e corrigidos.
- Pré-release – versão candidata. É uma versão com algumas instabilidades pois pode ter incompatibilidades no pacote.

Semantic version

Principais regras:

- Uma vez determinada e disponibilizada, a versão do software não poderá ser modificada em hipótese alguma.

- No desenvolvimento inicial de um software sua primeira versão deverá ser setada como zero, isso deve-se ao fato de que muita coisa poderá mudar a qualquer momento antes do seu lançamento e com isso sua versão inicial não pode e nem deve ser considerada estável.

Semantic version

- A versão de remendo, correção ou patch deverá indicar a correção de um ou mais bugs que se mantém compatível com a versão anterior. Ou seja, uma versão 1.0.1 corrige um problema na versão 1.0.0 e será totalmente compatível com a mesma.

- Sobre a ordem em que as versões do versionamento semântico deverão ter. No caso do versionamento semântico essa ordem será numérica (1.0.0 < 2.0.0 < 3.0.0 < 3.1.0) e assim por diante.

Semantic version

Exemplos:

- Versão 2.0.1
- Versão 2.7.0
- Versão 3.0.0
- Versão 0.67.1

Semantic version – aplicação no git

- Existem algumas formas de aplicarmos o semantic version em repositórios git:
  - Nomes de branchs
  - Tags
  - Mensagens de commit

5 Git no dia a dia

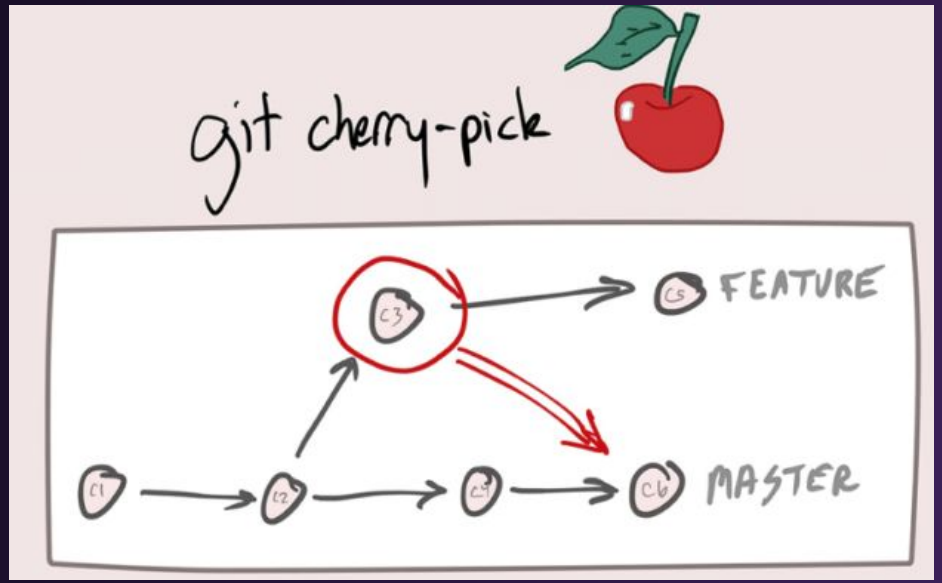
Git no dia a dia

- Setup do projeto
  - Permissão para criação dos repositórios
  - Setup do código, configuração de Husky, commitlint, eslint, etc.
  - Criação das branchs base (master, development, etc)
- Uso diário
  - Criação das branchs para as tarefas
  - Code review
  - Teste
  - Ci/Cd
- Criação de release
  - Tags
  - Ci/Cd

6 cherry pick

Cherry pick

- O cherry pick consiste basicamente em você copiar o commit de uma branch para outra
- É feito através do hash do commit



Cherry pick

- Basta escolher o commit que você deseja pegar, e na branch que deseja aplicar digitar:
  - Git cherry-pick hashDoCommit
  - Também podemos trabalhar com intervalos de commits:
    - git cherry-pick A^..B (aqui o commit A virá também)
    -
    - git cherry-pick A..B (aqui o commit A será desconsiderado)
  - É importante salientar que o commit A precisa estar após o commit B, ou seja, ser mais velho na timeline.

Cherry pick

- Caso dê conflito, devemos resolvê-los e usar o comando - -continue para dar sequência ao anexo dos commits
  - git cherry-pick - -continue
- Para abortar podemos usar:
  - git cherry-pick - -abort

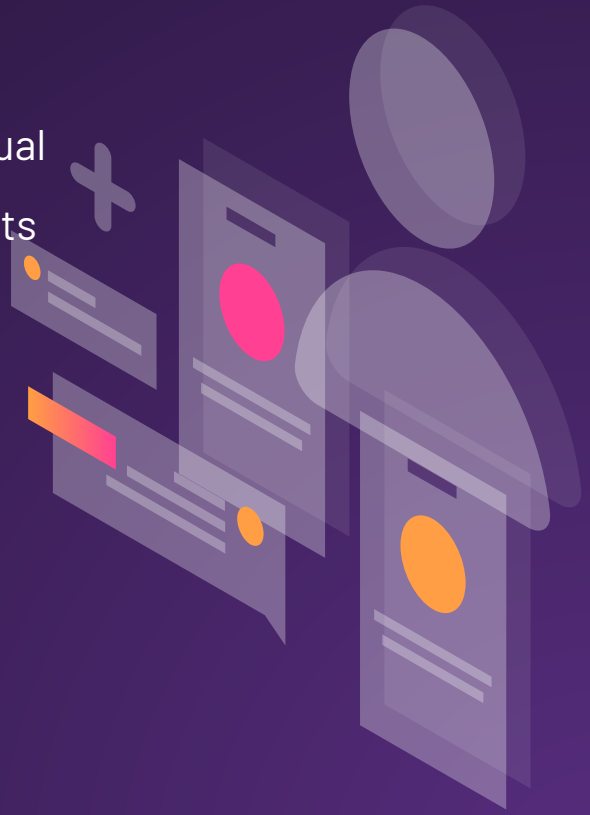
Cherry pick

- Parâmetros
  - **-e :** permite que você edite a mensagem do commit.
  - **-x:** adiciona uma mensagem no commit copiado avisando que ele é um cherry pick de um outro commit – "cherry picked from commit".
  - **–allow-empty:** por padrão, o cherry-pick não permite commits em branco, com esse parâmetro, ele sobrescreve esse comportamento.
  - **–allow-empty-message:** quando o commit não tem um título, ele é barrado. Assim como no exemplo anterior, esse parâmetro sobrescreve o comportamento.

7 rebase e merge

Merge

- Irá adicionar os commits de outra branch na sua branch atual
- Exemplo: **git merge develop**, irá adicionar todos os commits da branch develop que você não tenha na sua branch atual
- É recomendado sempre fazer o merge da branch pai na branch filha



Rebase

- Irá deslocar todos os commits da sua árvore para o ponto de rebase.
- Exameplo: **git rebase develop**, irá aplicar todos os commits da minha branch atual a partir do último commit da branch develop.

Quando usar cada um?

Ambos os comandos são muito úteis, porém, em situações diferentes. Quando seguimos corretamente o Git Flow, existe uma regra que chamamos de “Golden Rule of Rebasing”.

Basicamente, essa regra diz que o rebase não deve ser utilizado em branches públicos, já que ele tem um alto potencial catastrófico e destrói o histórico de commits, causando sempre divergências entre os branches locais e os branches remotos.

Quando usar cada um?

Caso você queria mesclar duas branchs sem criar um novo commit, pode utilizar o rebase. Exemplo, trazer alterações da branch master para a sua branch

Caso você deseje alterar uma branch pública, ou criar um commit para poder rastrear as informações que foram inseridas, um merge é melhor

8 Rename

Rename

Renomear a branch atual: **git branch -m new-branch-name**

Renomear outra branch, que não a atual: **git branch -m old-branch new-branch**

9 tags

Tags

São etiquetas que demarcam um ponto (commit) que representa alguma mudança significativa no seu código, ou seja, uma versão (ou release) do seu projeto.

Tags

São etiquetas que demarcam um ponto (commit) que representa alguma mudança significativa no seu código, ou seja, uma versão (ou release) do seu projeto.

Existem dois tipos de tag: annotaded e lightweight. De maneira bem resumida, podemos dizer que tags annotated armazenam detalhes sobre o estado do repositório naquele momento, enquanto tags lightweight armazenam apenas o checksum do commit em que foram geradas.

Tags

Criando tags *annotaded*

- git tag -a <name> -m <message>

Vendo as tags do projeto

- git tag

Informações sobre a tag

- git show name

Tags

Fazendo o push da tag

- git push origin name

Excluindo localmente a tag

- git tag -d name

Publicando a exclusão no remote

- git push —delete origin name

Tags

Criando tags lightweight

- git tag name-lw

Adicionando tags a commits já existents

- git tag -a name commitHash