

Semana 8 - Aula 3

# Git Flow

FUTURE4

# *Sumário*

# O que vamos ver hoje? 🙄

- A ideia de hoje é aprofundar um pouco o conceito de **git** de vocês!
- Falaremos sobre:
  - O arquivo .gitignore;
  - Outros comandos;
  - Conflitos, em geral; e
  - Git Flow

*Retomando alguns  
comandos*

# Comandos já vistos

- **git init**
  - É o comando usado para transformar um diretório qualquer em um repositório do git
- **git status**
  - Usado para verificar o status atual do repositório
  - Indica se está atualizado; quais forem as modificações e mais

# Comandos já vistos

- **git add**
  - Adiciona os arquivos na área de staging
  - Podemos usar a opção --all para pegar todos os arquivos
- **git commit -m**
  - Persiste as modificações que estão em staging
  - É necessário passar uma mensagem, identificando o que foi feito pelo desenvolvedor

# Comandos já vistos

- **git push**
  - Manda modificações para um repositório remoto
- **git pull**
  - Pega as modificações de um repositório remoto

# Comandos já vistos

- **git branch**
  - Permite ver uma lista com as branches
  - Se colocarmos um 'nome' depois, ele cria uma nova branch com esse nome
- **git checkout**
  - Muda de branch



*.gitignore*

# .gitignore 🤚

- O .gitignore (o ponto no início é importantíssimo) é onde colocamos os arquivos que devem ser ignorados pelo git
  - Ou seja, é a lista dos arquivos, cujas modificações, **não entrarão nos commits**, nem no repositório remoto

# .gitignore 🖐️

- A sintaxe dele é bem simples, basta colocar o arquivo em questão em uma determinada linha
- O símbolo **\*** indica generalização:
  - a/\*/b manda o git ignorar: a/x/b; a/x/y/b; etc.
- O símbolo **#** é usado para indicar comentários
- Mais informações sobre a sintaxe encontram-se neste [link](#)

## .gitignore 🖐️

- Se nós adicionarmos um novo arquivo, precisamos pedir para **resetar as configurações** do que está sendo trackeado ou não:

**git rm --cached**

# .gitignore 🖐️

- Usamos isto quando **não** queremos que certas informações estejam no repositório remoto:
  - node\_modules
  - tokens para ambientes
  - arquivos específicos da sua máquina

Vamos ver na prática! 🧐

*Novos comandos*

# Novos comandos


- **git remote**

- Todos os repositórios fora da nossa máquina são chamados de remote
- É possível ter mais que um repositório remoto. O comando git remote permite configurar todos os repositórios remotos

# Novos comandos

- **git remote**

- git remote add referência-ao-repo URL
  - Permite adicionar um novo remote ao projeto
- git remote remove referência-ao-repo
  - Permite retirar um remote do projeto


Vamos ver na prática! 



# Novos comandos

- **git diff**


- Permite que verifiquemos as diferenças da branch local com a sua respectiva no repositório remoto
- Conseguimos especificar, inclusive, qual arquivo queremos ver as diferenças

Vamos ver na prática! 

# Novos comandos

- **git fetch**

- Ele permite que baixemos todas as alterações feitas no repositório remoto
  - commits novos
  - branches novas
- Ele não atualiza as branches locais
  - usamos git pull para isto

Vamos ver na prática! 

# Novos comandos

- **git tag**

- git tag nome-da-tag: Permite criar uma nova tag
- Tags são maneiras de se identificar o código facilmente
- Ex.: v1.3.4

# Novos comandos

- **git tag**

- Elas permitem identificar algum ponto importante da árvore de commits
- É possível dar checkout para uma tag:

**git checkout nome-da-tag**

**Vamos ver na prática! **

# Pausa para relaxar

10 min

- **PONTOS IMPORTANTES:**

- git diff
- git fetch
- git remote
- git tag

*Conflitos*

# Conflitos

- Imagine a seguinte situação: você está trabalhando em um projeto grande em uma empresa.
- Pedem, então, para você resolver um bug muito importante na tela do Login de um projeto
- Ao mesmo tempo, um colega seu tem que implementar um novo tipo de Login: através do Facebook

# Conflitos

- Bem, no meio do caminho, você descobriu o problema do Bug e resolveu
- Enquanto seu colega implementou toda a funcionalidade sem consertar o bug; mas deu um “migué” para que tudo que ele fez funcionasse
- Você subiu o PR, foi aprovado, e mergeou




# Conflitos

- **O que vai acontecer com o PR do seu colega?**
  - Vai estar com CONFLITO
- **O que é um conflito?**
  - Conflito é uma situação em que o git “não sabe” quais modificações de um mesmo arquivo ele deve levar em consideração

# Conflitos

- **Como acontece um conflito?**
  - Quando as modificações partem de um mesmo ponto em comum
  - Quando as modificações ocorrem nas mesmas linhas de um arquivo

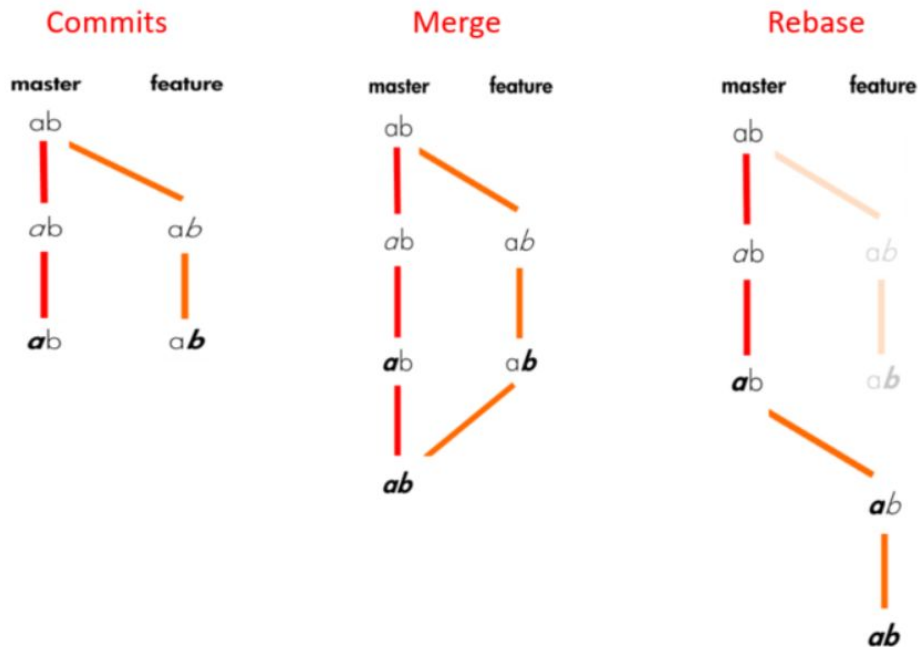
Vamos ver na prática! 

# Resolvendo Conflitos

- **Como resolvemos conflitos?**
  - Para isto, precisamos fazer com que os nossos commits partam dos novos commits da master
    - E, não mais da versão antiga da master
  - Quando isso acontece, dizemos que fazemos um *rebase*

# Resolvendo Conflitos 🕊

- git rebase



# Resolvendo Conflitos

- **git rebase**

- git rebase <branch-2>: Dá o rebase na branch original em relação a branch-2
- git rebase --continue: Continua o rebase (tem que ser feito depois de um **git add**)
- git rebase --abort: Interrompe o processo de rebase

Vamos ver na prática! 

# Resolvendo Conflitos

- **Problema**

- Quando fazemos o rebase, o git vai entender que a nossa branch está desatualizada em relação a branch remota
- Desta forma, ele não vai deixar que subamos a nossa branch

**git push origin <branch> --force**

**git push origin <branch> -f**

Vamos ver na prática! 

FUTURE 

# Pausa para relaxar

5 min

- **PONTOS IMPORTANTES:**

- Conflitos tem que ser resolvidos manualmente
- Processo para corrigir conflitos:
  - 1. **git rebase** branch-para-rebase
  - 2. (resolve os conflitos na mão)
  - 3. **git add --all**
  - 4. **git rebase --continue**
  - 5. (refazer 2, 3 e 4 até acabarem os conflitos)
  - 6. **git push -f**

# *Git Flow*



# Git Flow

- Git Flow são diretrizes que estabelecem **padrões** para **nomes** e **funções** de cada branch do nosso projeto
- Branch Principal
  - **master**: é o código que está no ambiente de produção. Tudo que estiver nesta branch, com certeza, é o código que está na mão dos nossos usuários

# Git Flow

- Branches de Desenvolvimento
  - **develop:**
    - é a branch principal de desenvolvimento. Tudo que já foi **testado** no nosso ambiente de desenvolvimento fica nesta branch
    - Quando um conjunto de features de desenvolvimento estiver pronto, nós fechamos/mergeamos a branch develop na **release**

# Git Flow

- Branches de Desenvolvimento
  - **feature/**:
    - É o sufixo de todas as branches que contém códigos de uma nova funcionalidade

**feature/user-signup**  
**feature/user-signin**  
**feat/feed**

# Git Flow

- Branches de Desenvolvimento
  - **feature/**:
    - Todas as branches deste tipo são criadas a partir da **develop**
    - Assim que terminada a funcionalidade, a branch em questão é fechada/juntadas com a **develop**

# Git Flow

- Branches de Erro
  - **hotfix/**:
    - São branches que significam um erro crítico que esteja em **produção**
    - Criadas, sempre, a partir da **master**. Quando concluídas são fechadas/juntadas com a **master**

# Git Flow

- Branches de Erro
  - **release/**:
    - É uma branch intermediária entre a develop e a master.
    - Criadas, sempre, a partir da **develop**. Quando concluídas são fechadas/juntadas com a **master**

Vamos ver na prática! 

FUTURE 

# Resumo

- Vimos hoje os seguintes comandos:
  - git fetch
  - git remote
    - git remote add
    - git remote remove
  - git tag
    - git tag <valor> -m

# Resumo

- **Conflitos**

- Acontecem quando mexemos na **mesma linha** do **mesmo arquivo**
- Tem que ser resolvidos **manualmente**

- **git rebase**

- --continue
- --abort

- **git push -f**



# Resumo

- Git Flow
  - **master**: código em produção
  - **develop**: código principal de desenvolvimento
  - **feature/**: branches com novas funcionalidades
  - **hotfix/**: branches com correções da master
  - **release/**: branches com o código pronto para ir para master

# Dúvidas?

Obrigado!