



Git



Gerenciamento de Código Fonte



Sumário

1. Conceitos

- a. Gerenciamento de Código Fonte

2. Git

- a. Introdução e História
- b. Instalação e configuração
- c. Repositórios
- d. Comandos Básicos
- e. Branches e Tags
- f. Merge e Rebase
- g. Fluxos de Trabalho

3. GitHub

- a. Criação de Conta
- b. Gerenciamento de Repositórios
- c. Pull Request



Conceitos

- A. Gerenciamento de Código Fonte



Gerenciamento de Código Fonte

“Controlar cada peça de código”

- Fonte de Programas
- Arquivos de configuração
- Binários
- Dependências (bibliotecas, etc)

=> CI (Configuration Items)

Gerenciamento de Código Fonte

Principais Objetivos

Salvaguardar qualquer recurso de um projeto de software.

Também envolve manter um registros dos marcos (milestone) específicos no processo de desenvolvimento - *baseline* do código.

Gerenciar desenvolvimento paralelo, correção de bugs e desenvolvimento distribuído.

Gerenciamento de Código Fonte

Conceitos Importantes

- Equipes e Paralelismo
- Branches
- Merges
- Conflitos
- Labels (Tags)

Git

- A. Introdução e História
 - B. Instalação e configuração
 - C. Repositórios
 - D. Comandos Básicos
 - E. Branches e Tags
 - F. Merge e Rebase
 - G. Fluxos de Trabalho
-

Introdução e História

O que é o Git?

Ferramenta para rastrear mudanças em um conjunto de arquivos ao longo do tempo.

Controle de Versão.

Outras: SVN, CVC, VSS, Mercurial, ClearCase,
...

O que se pode fazer?

Examinar o estado do projeto (repositório).

Exibir as diferenças entre estados do projeto.

Dividir o projeto entre linhas independentes que evoluem separadamente - *branches*.

Periodicamente recombinar branches, reconciliando diferenças - *merge*.

Permitir que muitas pessoas trabalhem simultaneamente, compartilhando e combinando seus trabalhos quando necessário

Introdução e História

Controle Versão Clássico

Centralizado: Existe apenas 1 cópia do repositório.

Acesso via rede. Se o acesso falhar, todos os usuários não terão acesso.

Controle Versão Distribuído

Descentralizado: Cada usuário tem 1 cópia (independente) do repositório.

Acesso via rede só é necessário quando deseja-se compartilhar alterações entre os repositórios.

2 passos: 1) adicionar as alterações para a área de *stage* chamada “index”, 2) *commit* as alterações para o repositório (local)

Facilita a utilização de diversas formas de trabalho (flows)

Introdução e História

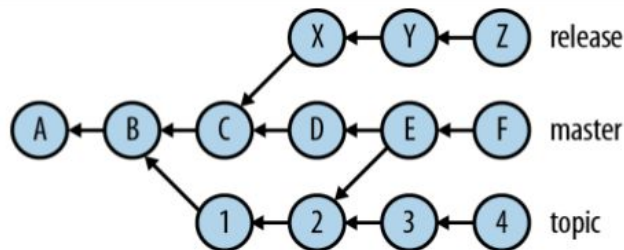
Terminologia

Projeto Git: Repositório (pasta no sistema de arquivos). Conjunto de snapshots de conteúdo da pasta - coleções de arquivos e diretórios - *commits*

Commit:

- Snapshot de conteúdo: *tree*
- Author Identification: Nome, email e data/hora da alteração
- Committer Identification: Idem Author (mas pode ser diferente)
- Commit Message: Comentário do commit

- Parent Commits: Lista de referência a outros commits identificando precedência nos estados dos conteúdos



Introdução e História

Terminologia

Branches: Coleção de todos os commits no grafo que é alcançável a partir do último commit seguindo as setas ao longo da história (até o início)

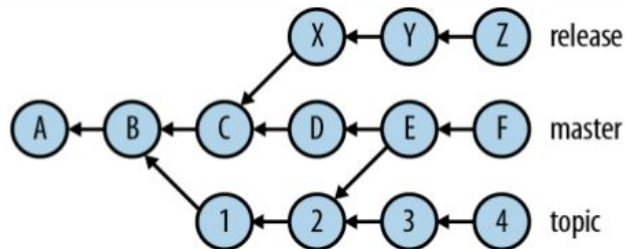
release: {A, B, C, X, Y, Z}

master: {A, B, C, D, E, F, 1, 2}

topic: {A, B, 1, 2, 3, 4}

Branches: release, master, topic

Commits: (last) Z, F e 4



Introdução e História

Linus Torvalds

Criou o Git em 2005 para desenvolvimento do Kernel do Linux

BitKeeper tool

“an unpleasant or contemptible person”

“the stupid content tracker”

Junio Hamano

Mantenedor atual do Git desde 2005, após a primeira release

Introdução e História

The Object Store

Banco de dados do Git

Mantém 4 tipos de itens: *blobs*, *trees*, *commits* e *tags*

Blob

- Massa de bytes sem estrutura
- Conteúdo de um arquivo sob controle de versão é representado como um blob
- Técnicas sofisticadas de compressão e transmissão para tratar blobs eficientemente

-
- Cada arquivo é representado como um todo, com seu próprio blob, completo
 - Diferente de outros gerenciadores de versão onde um arquivo é representado como uma série de diferenças de uma revisão à outra
 - Por isso, Git ocupa mais espaço
 - Porém mais rápido pois não precisa reconstruir os arquivos e aplicar camadas de diferenças

Introdução e História

The Object Store

Blob

- Este modelo aumenta a confiabilidade e incrementa a redundância: corrupção de um blob afeta apenas aquela versão do arquivo, enquanto no modelo de “diferenças” corrompe todas as versões dependentes da corrompida

Tree

- Representa uma porção do conteúdo do repositório num ponto do tempo: snapshot do conteúdo de um particular diretório, incluindo todos os subdiretórios

Introdução e História

The Object Store

Commit

- Unidade fundamental de alterações
 - Snapshot do conteúdo do repositório, junto com informações de identidade e relações e histórico
 - Ponteiro para a *tree* contendo o estado completo do conteúdo do repositório num determinado momento
 - Informações: author, commiter, data/hora
-
- Lista de “*parent commits*”
 - Author: Responsável pelo conteúdo do commit
 - Committer: Responsável por adicionar o commit no repositório
 - Author e Committer inicialmente são iguais, mas alguns comandos podem tornar estas informações distintas. Ex: cherry-pick

Introdução e História

The Object Store

Tag

- Permite marcar um “*Label*” para um determinado commit, permitindo que mesmo fique registrado na lista de tags
- Normalmente é utilizado para definir marcos importantes do desenvolvimento de um software, por exemplo, as releases

- Tipos

- Annotated
- Lightweight
 - Simplesmente um nome apontando para um commit

Introdução e História

The Object Store

SHA-1

- Um commit é identificado por uma função hash que utiliza o algoritmo SHA-1 de 160 bits
- O hash é calculado a partir do conteúdo do commit
- Permite grande performance

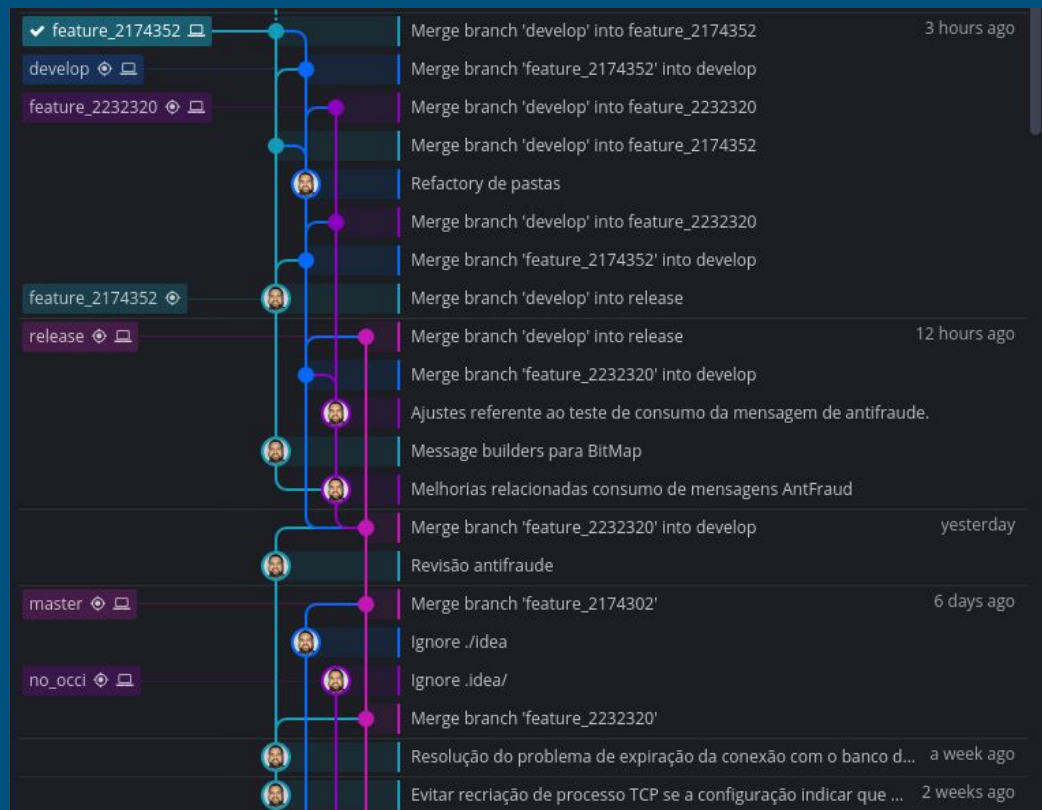
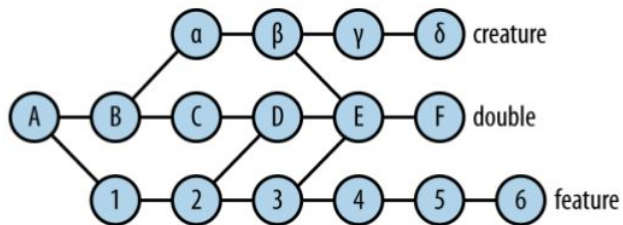
Outros Versionadores

Outros versionadores são baseados num identificador numérico, normalmente baseado na ordem com que os commits vão acontecendo.

Exemplo: CVS e SVN

Introdução e História

Commit Graph



Introdução e História

Refs

Tipos de Referência

- Ref simples, que aponta diretamente para um ObjectID (usualmente um commit ou tag)
- *Symbolic Ref* (symref), que aponta para outra ref (simples ou symref)
- Pasta /refs
- Um novo repositório tem pelo menos refs/tags/ e refs/heads para manter as tags e branches locais

-
- refs/remotes que mantem nomes referenciando outros repositórios

Introdução e História

Branches

Um ponteiro para um commit, tal como uma ref

- HEAD: ref especial que aponta para a branch que estamos, se o HEAD é uma symref para uma branch existente.
- Se o HEAD é um ref apontando diretamente para commit pelo SHA-1 ID, então não estamos na branch, mas sim no modo “*detached HEAD*”. Isso acontece quando fazemos checkout de algum commit para examina

```
# HEAD points to the master branch
$ git symbolic-ref HEAD
refs/heads/master
```

```
# Git agrees; I'm on the master branch.
$ git branch
* master
```

```
# Check out a tagged commit, not at a branch tip.
$ git checkout mytag
Note: checking out 'mytag'.
```

```
You are in 'detached HEAD' state...
```

```
# Confirmed: HEAD is no longer a symbolic ref.
$ git symbolic-ref HEAD
fatal: ref HEAD is not a symbolic ref
```

Introdução e História

Branches

- HEAD geralmente referenciado como o commit “corrente”, e se estamos numa branch, pode ser chamado de commit “*last*” ou “*tip*”

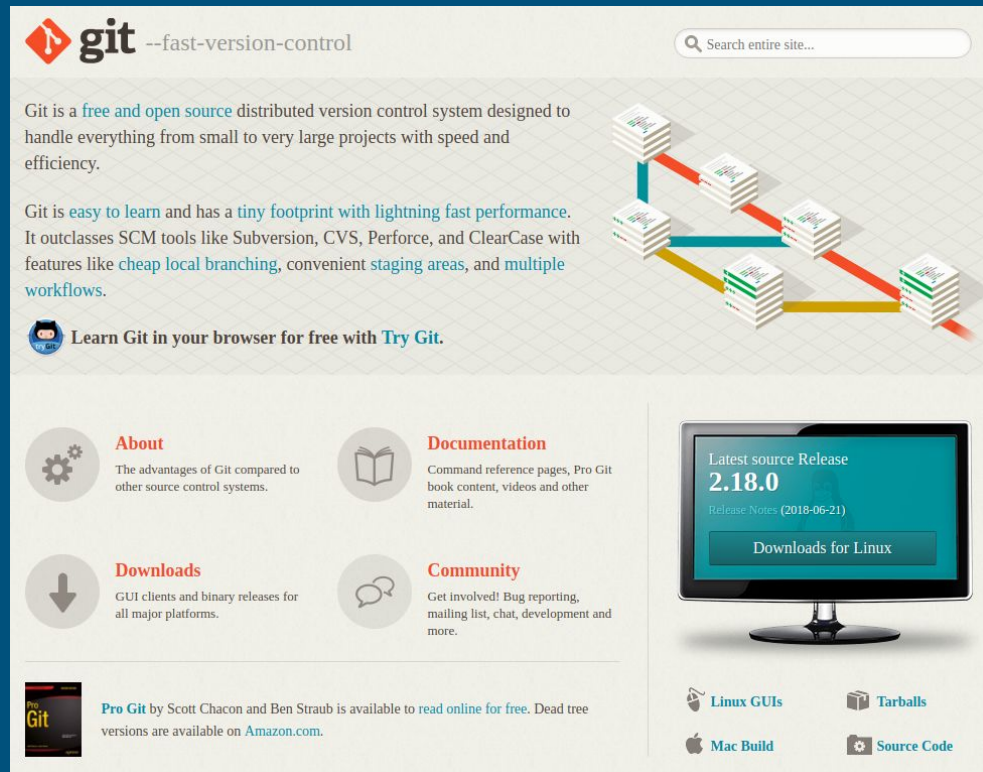
Quando fazemos commit

- Cria um novo commit com nossas alterações no conteúdo do repositório
- Torna o commit corrente da branch master o “*parent*” do novo commit
- Adiciona o novo commit para o *object store*
- Altera a branch master (refs/heads/master) para apontar para o novo commit

Instalação e Configuração

Instalação

<https://git-scm.com>




The screenshot shows the Git website homepage. At the top, the Git logo is followed by the tagline "--fast-version-control". A search bar is located in the top right corner. The main content area describes Git as a free and open source distributed version control system. It highlights features like easy learning, tiny footprint, lightning fast performance, and outclasses other SCM tools. A diagram on the right shows a branching model with stacks of code and colored lines representing branches. Below the main text, there's a section for "Learn Git in your browser for free with Try Git." and four icons representing "About", "Documentation", "Downloads", and "Community". On the right, a monitor displays the "Latest source Release 2.18.0" and "Downloads for Linux". At the bottom, there's a section for "Pro Git" by Scott Chacon and Ben Straub, and a footer with links for Linux GUIs, Tarballs, Mac Build, and Source Code.

git --fast-version-control

Search entire site...

Git is a [free and open source](#) distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is [easy to learn](#) and has a [tiny footprint with lightning fast performance](#). It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like [cheap local branching](#), convenient [staging areas](#), and [multiple workflows](#).

 Learn Git in your browser for free with [Try Git](#).

About
The advantages of Git compared to other source control systems.

Documentation
Command reference pages, Pro Git book content, videos and other material.

Downloads
GUI clients and binary releases for all major platforms.

Community
Get involved! Bug reporting, mailing list, chat, development and more.

Latest source Release
2.18.0
[Release Notes \(2018-06-21\)](#)
[Downloads for Linux](#)

Pro Git by Scott Chacon and Ben Straub is available to [read online for free](#). Dead tree versions are available on [Amazon.com](#).

[Linux GUIs](#) [Tarballs](#)
[Mac Build](#) [Source Code](#)

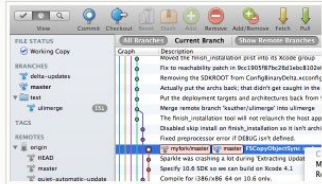
Instalação e Configuração

GUIs

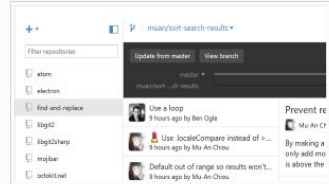
<https://git-scm.com/download/gui/windows>

AllWindowsMacLinuxAndroidIOS

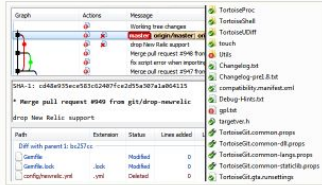
19 Windows GUIs are shown below ↓



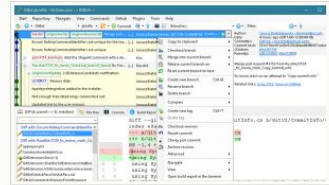
SourceTree
Platforms: Mac, Windows
Price: Free
License: Proprietary



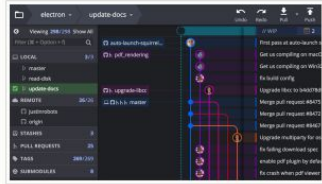
GitHub Desktop
Platforms: Mac, Windows
Price: Free
License: MIT



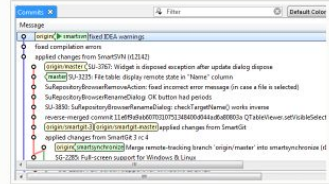
TortoiseGit
Platforms: Windows
Price: Free
License: GNU GPL



Git Extensions
Platforms: Linux, Mac, Windows
Price: Free
License: GNU GPL



GitKraken
Platforms: Linux, Mac, Windows
Price: Free for non-commercial use
License: Proprietary



SmartGit
Platforms: Linux, Mac, Windows
Price: \$79/user / Free for non-commercial use
License: Proprietary

Repositórios

O que é?

- Pasta gerenciada pelo Git, ou seja, um projeto Git
- Pode ser local ou remoto
- Inicializamos uma pasta para que a mesma torne-se um repositório Git, exemplo:
 - `mkdir myproj-git`
 - `cd myproj-git`
 - `git init`

-
- Podemos “*clonar*” um repositório remoto
 - Gera uma cópia do repo remoto
 - Permite atualizações entre os repositórios
 - `pull: local <- remoto`
 - `push: local -> remoto`
 - Podemos relacionar um repo local a um remoto
 - `git remote add ...`

Comandos Básicos

git init

git config --global user.name "Your full name"

git config --local user.name "Your full name"

git config --global user.email "xxx@server.com"

git config --local user.email "xxx@server.com"

git config -l

```
eduardo@sylvia ~/Private/dev-course $ git config -l
user.name=Eduardo Ribeiro da Silva
user.email=filosofisto@hotmail.com
push.default=simple
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
remote.origin.url=https://github.com/filosofisto/dev-course.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
```

Comandos Básicos

`git status`

`git add .`

`git add *.java`

`git add <file>`

`git reset <file> //move <file> from Stage area to Unstage area`

`git commit -m "Message commit"`

`git log`

`git checkout <commit id>`

Comandos Básicos

git checkout <branch>

git reset --hard <commit id> //Aborta commit após <commit id>

git clone <repo address>

git fetch

git merge

git push

git pull

git remote

Fluxos de Trabalho

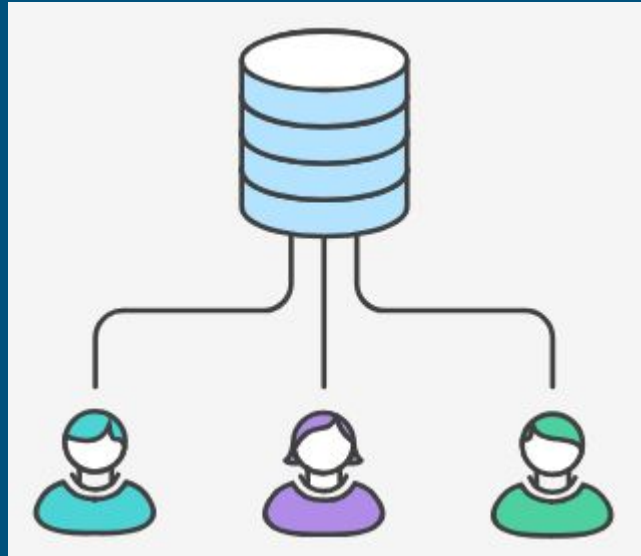
O que são?

Um fluxo de trabalho é uma recomendação ou receita de como utilizar o Git para trabalhar de forma consistente e produtiva

Processos mais usados

- Workflow Centralized
- GitFlow
- Fork Flow

Workflow Centralized



Workflow Centralized

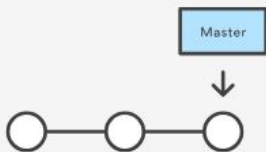
Características

- Ideal para equipes que estão migrando do SVN para o Git
- Repositório Remoto
- Apenas a branch *master*

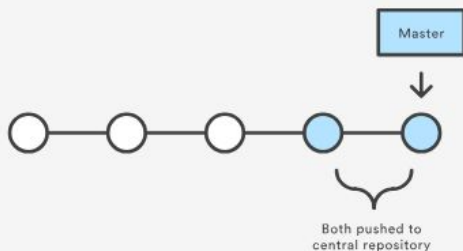
Workflow Centralized

Initialize the central repository

Central Repository



Local Repository

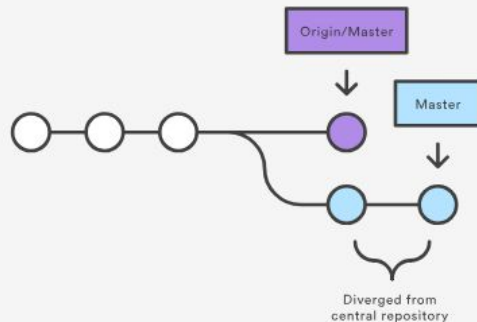


1. Inicializa repositório remoto
2. Clona repositório
3. Faz alterações e commit
4. Push novos commits ao repositório remoto
5. Gerencia conflitos

```
git status # View the state of the repo
git add <some-file> # Stage a file
git commit # Commit a file</some-file>
```

```
git push origin master
```

Local Repository

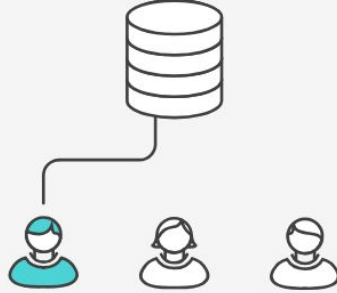


Workflow Centralized

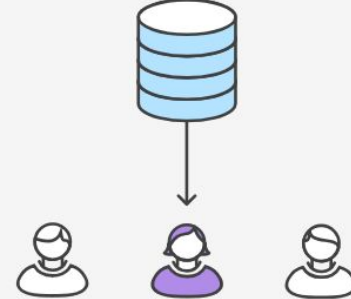
John works on his feature



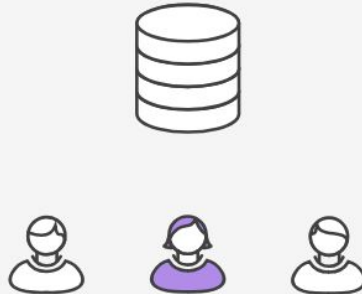
John publishes his feature



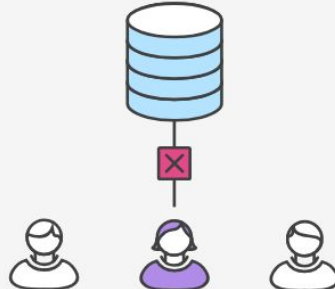
Mary rebases on top of John's commit(s)



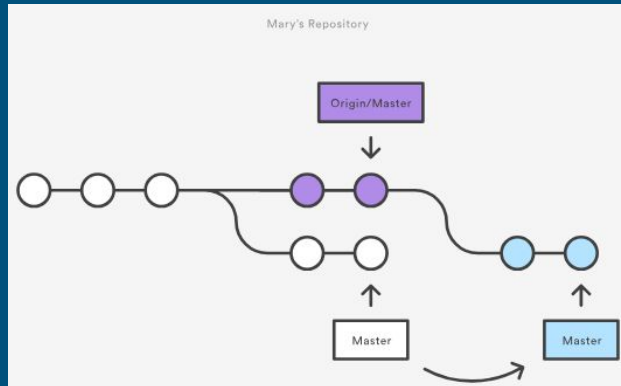
Mary works on her feature



Mary tries to publish her feature

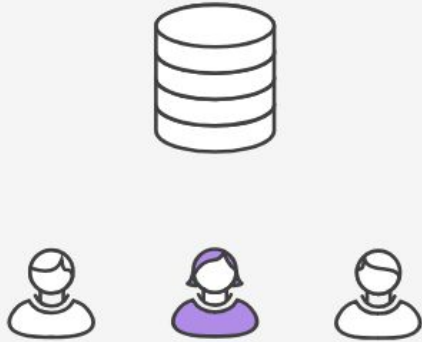


```
git pull --rebase origin master
```

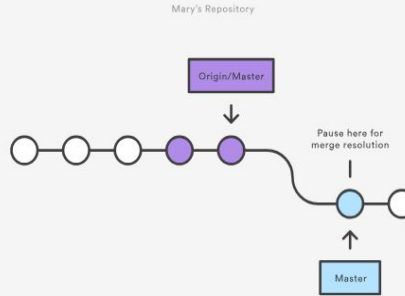


Workflow Centralized

Mary resolves a merge conflict



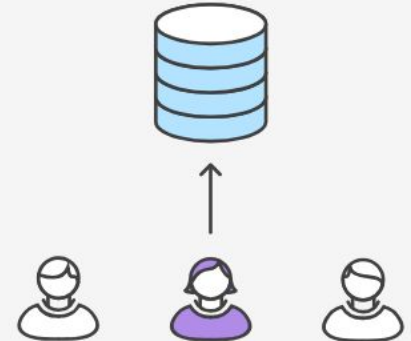
```
CONFLICT (content): Merge conflict in <some-file>
```



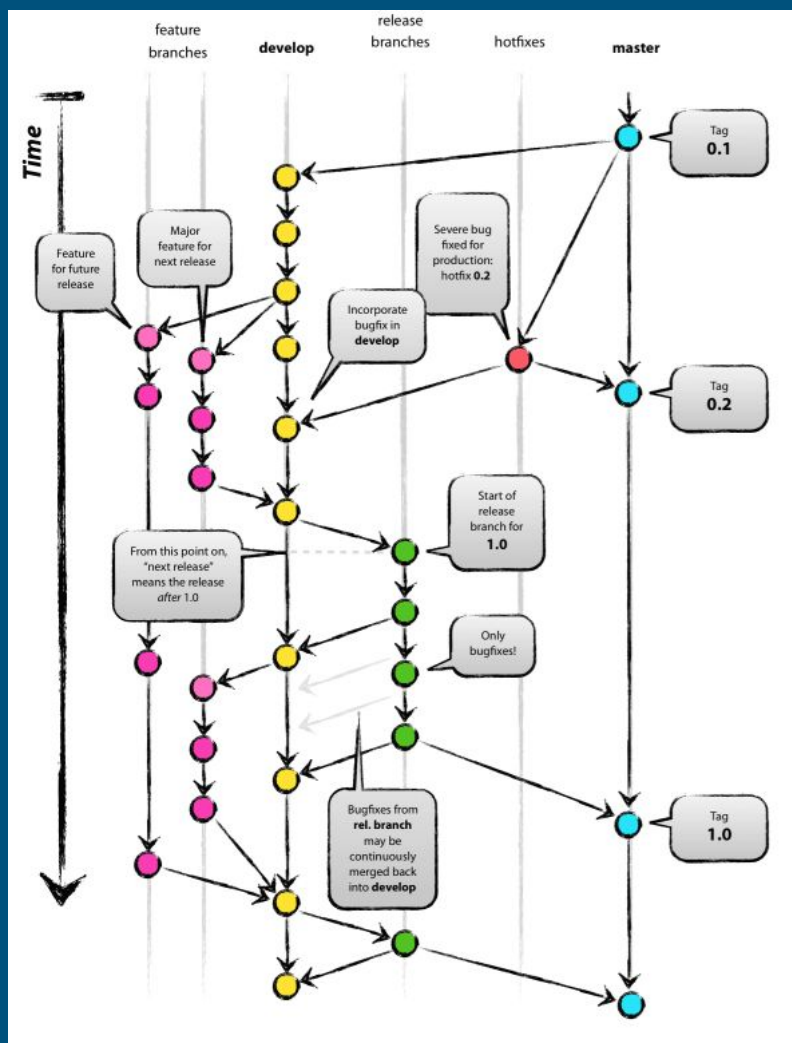
```
git add <some-file>  
git rebase --continue
```

```
git rebase --abort
```

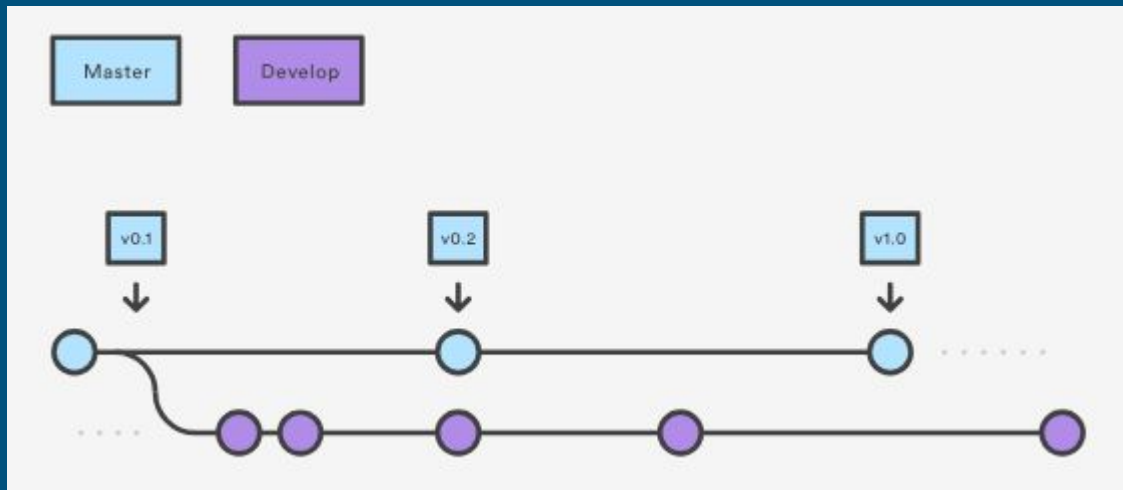
Mary successfully publishes her feature



GitFlow



GitFlow



```
git branch develop
git push -u origin develop
```

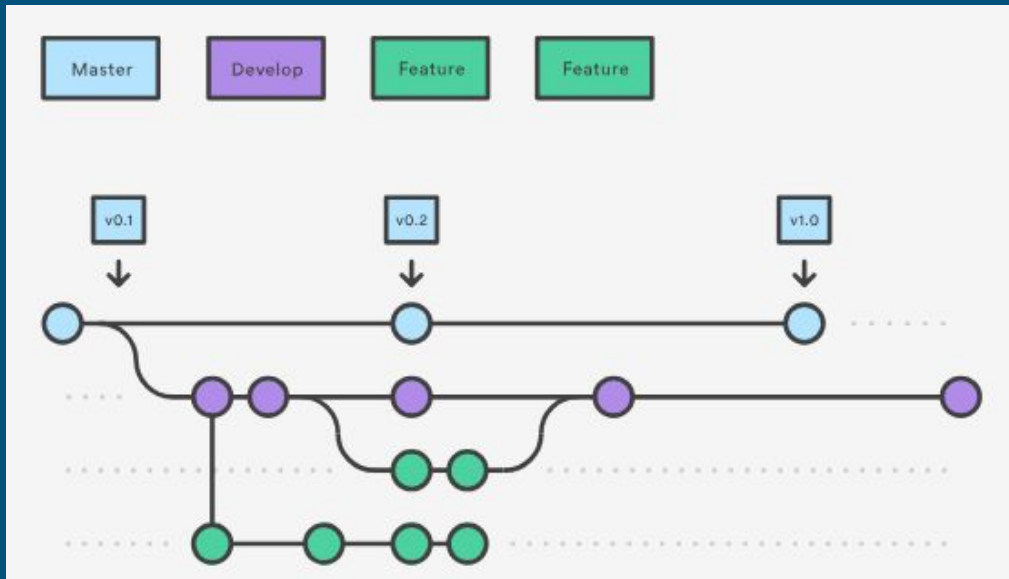
```
$ git flow init
```

```
Initialized empty Git repository in ~/project/.git/
No branches exist yet. Base branches must be created now.
Branch name for production releases: [master]
Branch name for "next release" development: [develop]
```

```
How to name your supporting branch prefixes?
Feature branches? [feature/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
```

```
$ git branch
* develop
master
```

GitFlow



```
git checkout develop  
git checkout -b feature_branch
```

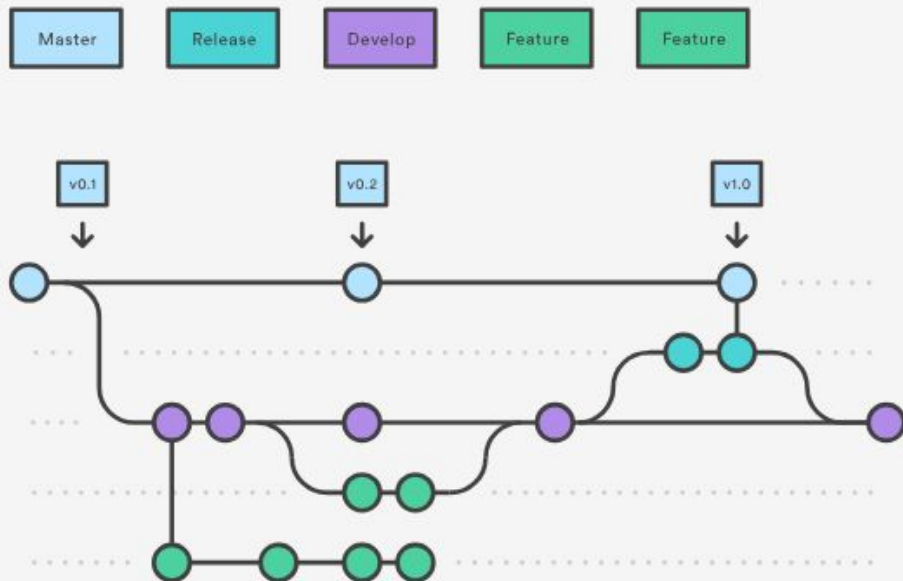
```
git flow feature start feature_branch
```

```
git checkout develop  
git merge feature_branch
```

```
git flow feature finish feature_branch
```

GitFlow

Release Branches



```
git checkout develop  
git checkout -b release/0.1.0
```

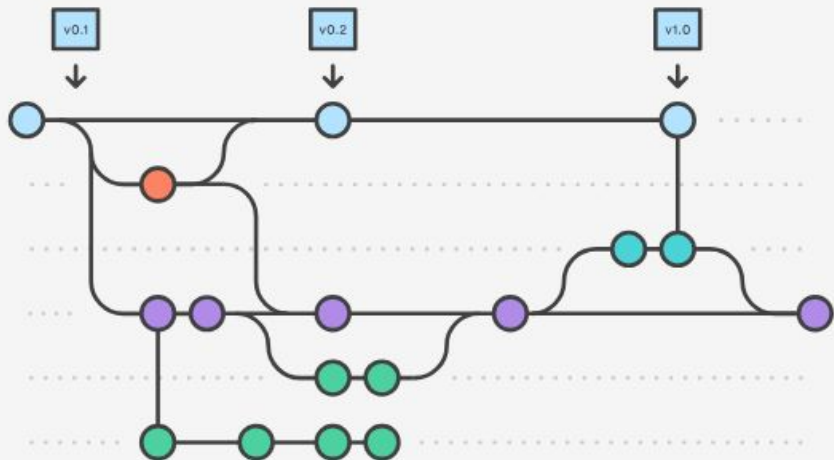
```
$ git flow release start 0.1.0  
Switched to a new branch 'release/0.1.0'
```

```
git checkout develop  
git merge release/0.1.0
```

```
git checkout master  
git checkout merge release/0.1.0  
git flow release finish '0.1.0'
```

GitFlow

Hotfix Branches



```
git checkout master  
git checkout -b hotfix_branch
```

```
git flow hotfix start hotfix_branch
```

```
git checkout master  
git merge hotfix_branch  
git checkout develop  
git merge hotfix_branch  
git branch -D hotfix_branch
```

```
git flow hotfix finish hotfix_branch
```