Software Engineering

Part 3: Versionning & Outils de développement

Guillaume Swaenepoel & Thomas Aubin

Git

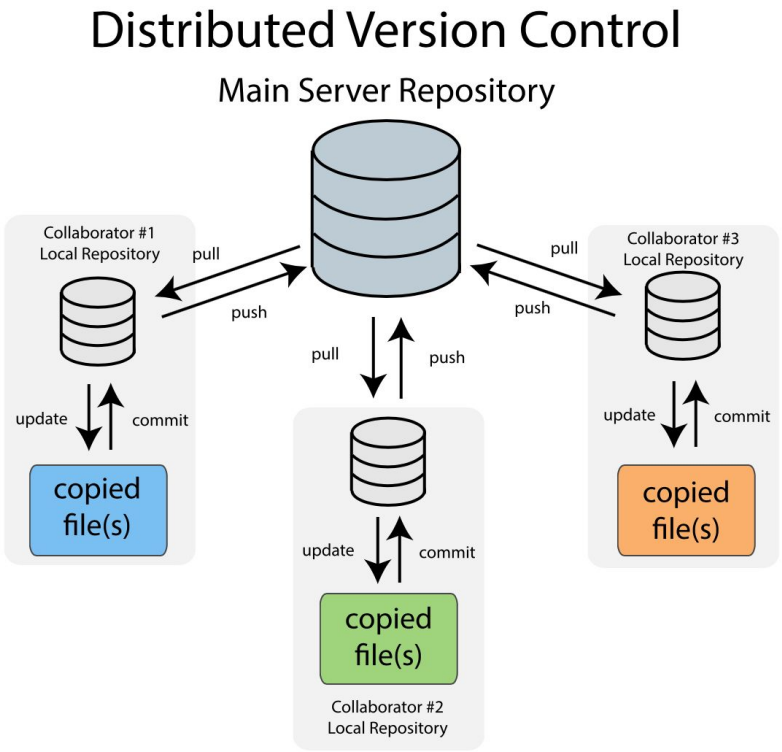
software versioning

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.

https://git-scm.com/

Distributed Version Control



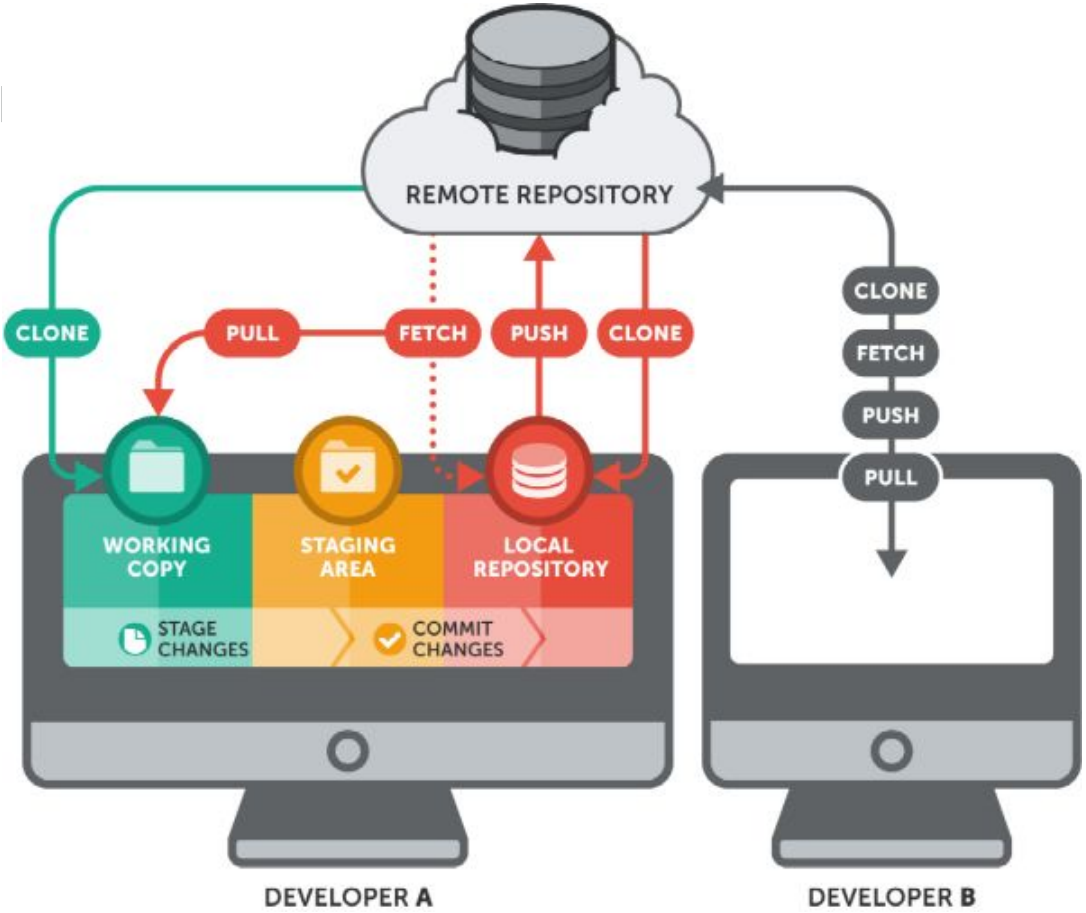
Asynchronous version

Working from a local copy of the server version.

Modification, new branches, commit, etc done locally.

Push operation to copy your current local version to the server

Pull operation to get modification from server



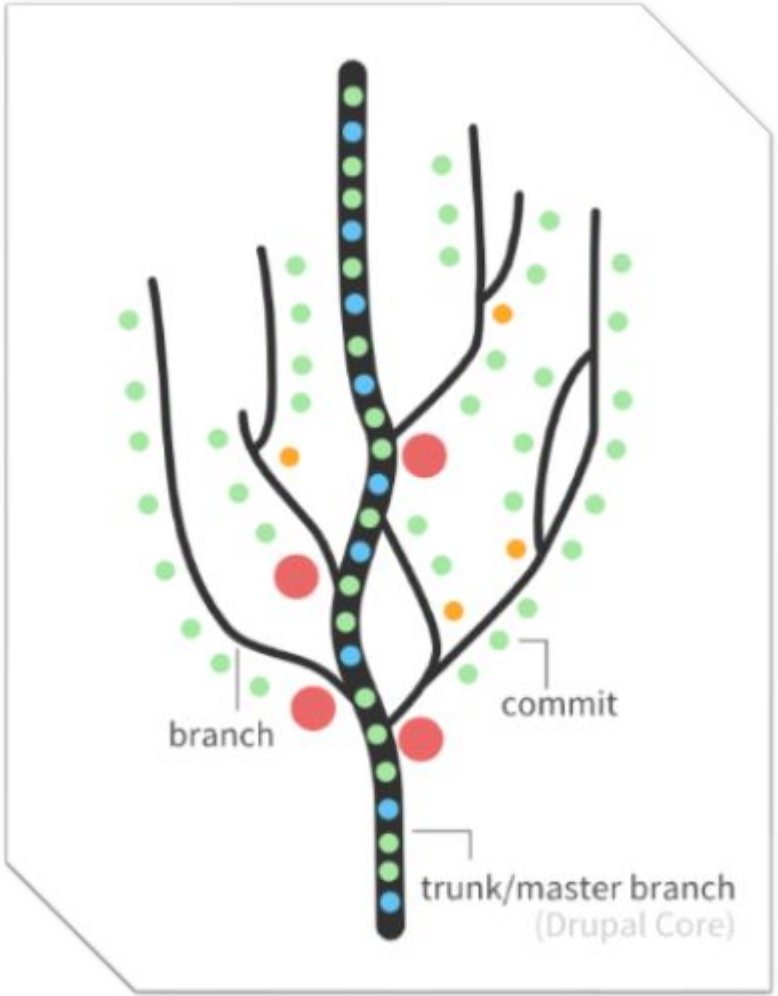
Repository

Complete tree of your project history

Split by branches

Composed by a main branch called “master”

Composed of “commit” : save of a version of a project.



Branches

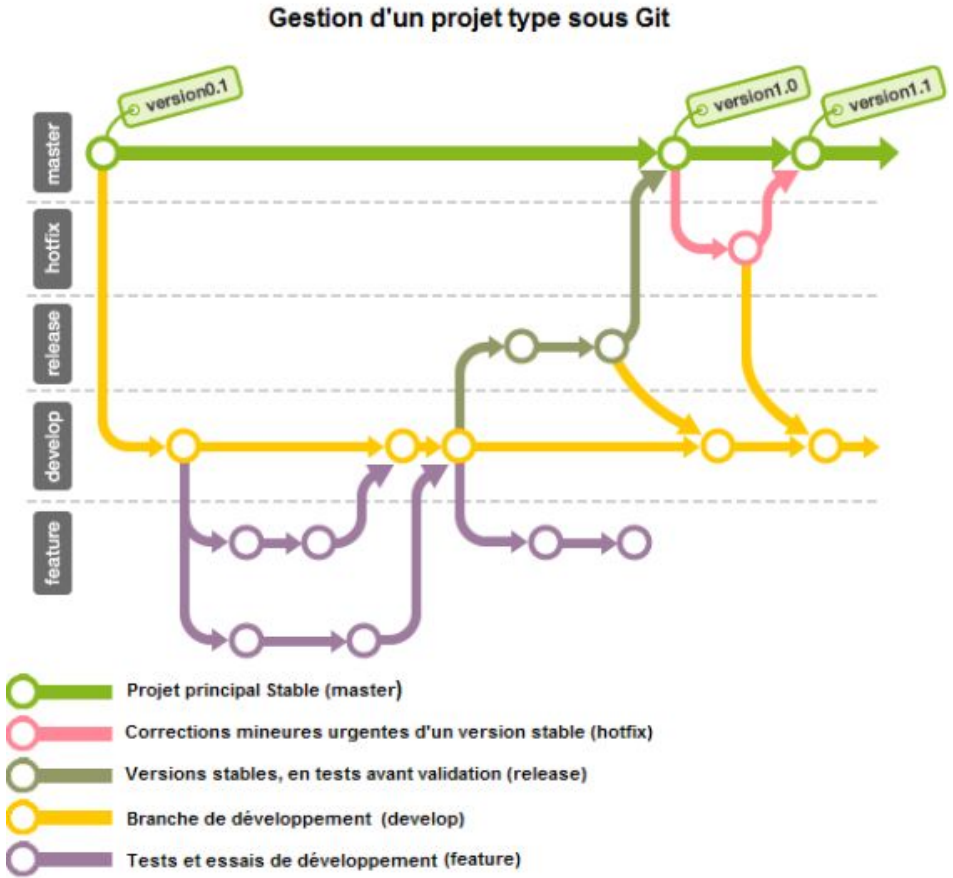
Branch : versioning path from a certain commit.

A branch can have has many commit/branch has necessary

Can be abandoned/deleted

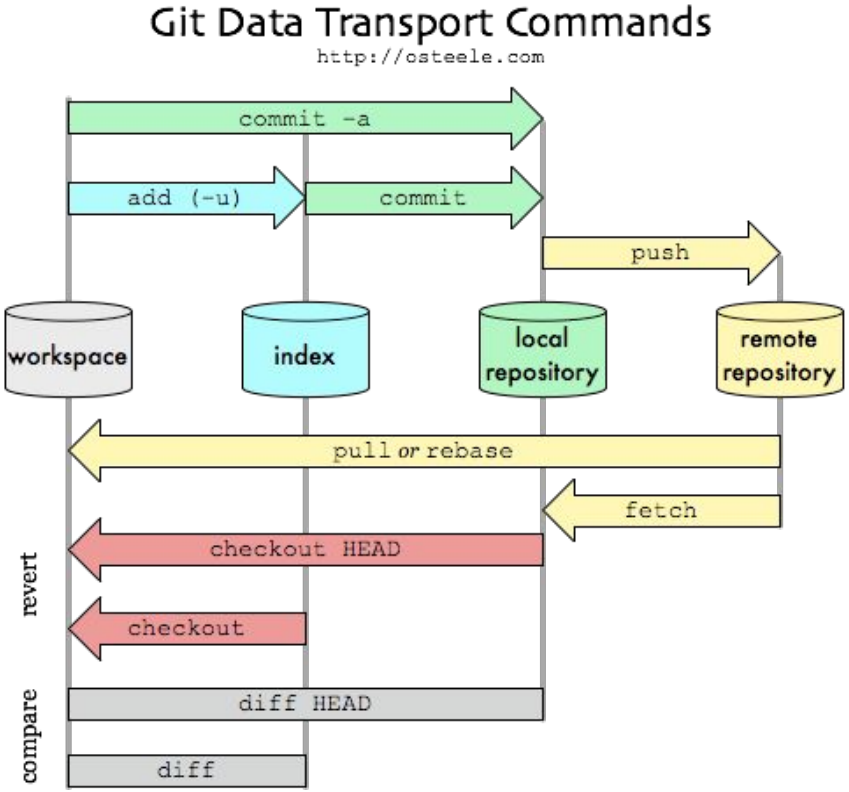
Can merge to other branch, even it’s own based

Branches names depends on the working organization



4 stage:

- Workspace
- index (files in cache waiting for commit)
- local repository
- remote repository (gitlab/github/...)

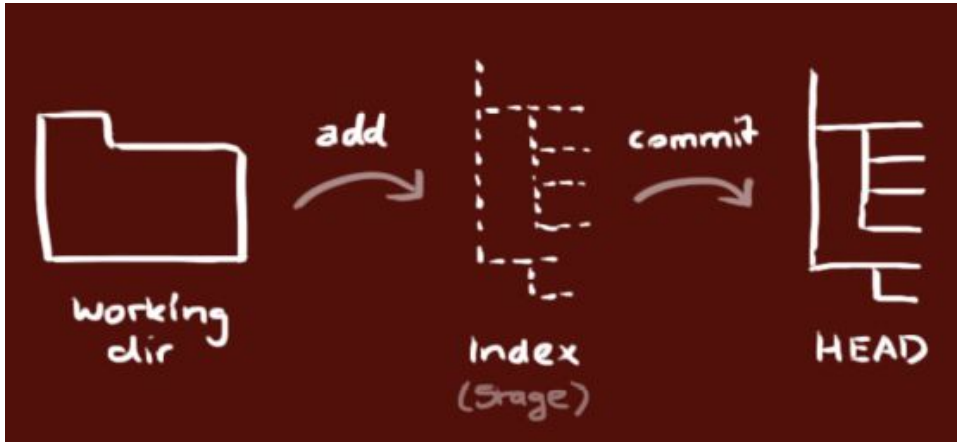


Add, commit & tags

Add : Add modifications to tree index. Modifications are recorded, but not saved -> needed for commit, but need a commit after, to validate your modification and comment them.

Commit : Validate the modification added to the index. Commit are linked to a user, a message and an ID (SHA). This modification will create a node on your project tree.

Tag : Add a label to a specific commit

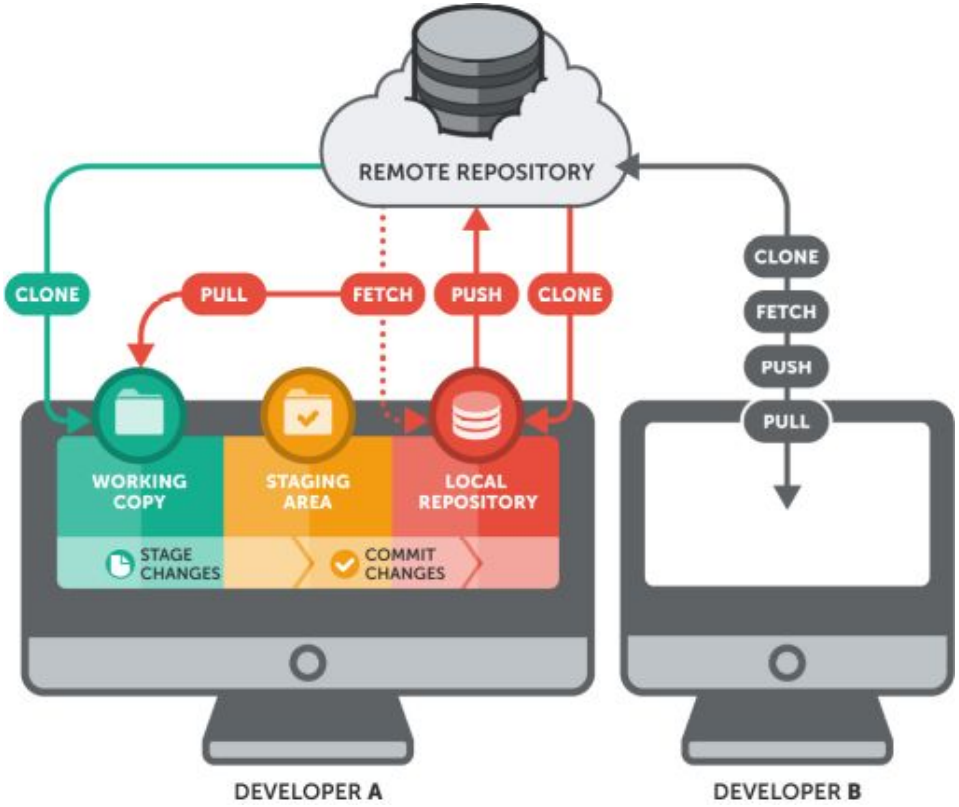


Clone & view tree

Clone : Create a local copy of distant repository (historical, branches, etc included).

Show log : print the entire historical repository with comment, commit, messages, tags, etc.

Many software are great to show your project graphically, for a better view of the organization.



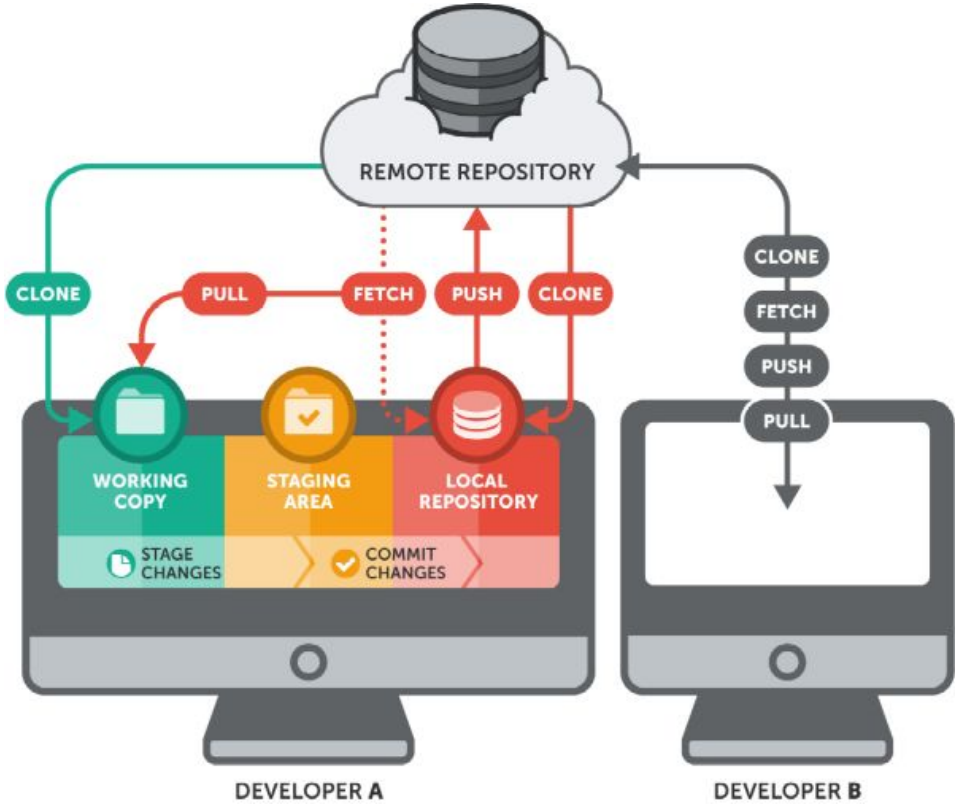
Push & Pull

Pull : Synchronize your branch with the distant one. Update your files.

!!! Your tree must have the same history to synchronised !!!

Merge : meld the modification to create a new commit. Use to synchronized branches or distant/local changes. Conflict needs to be resolved ! Test before push the merged commit.

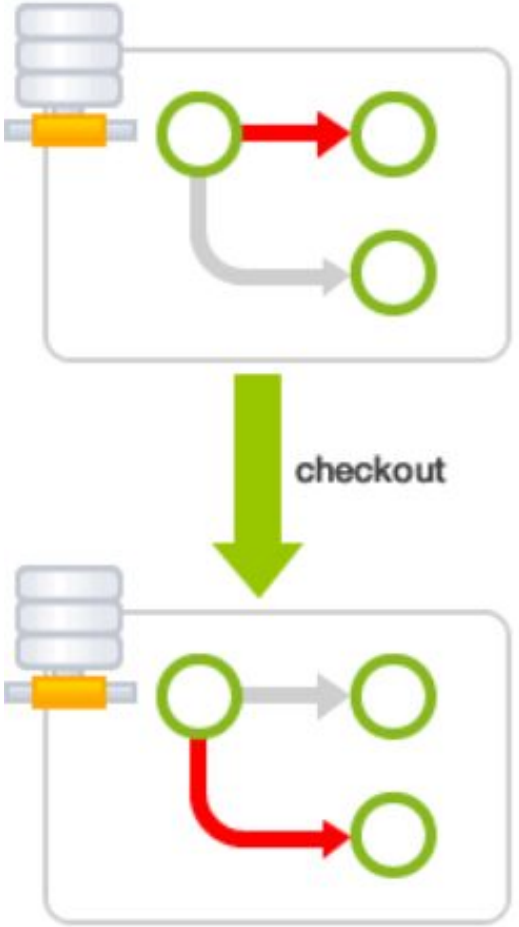
Push : Synchronized your local branch with the distant one. Your history must be the same.



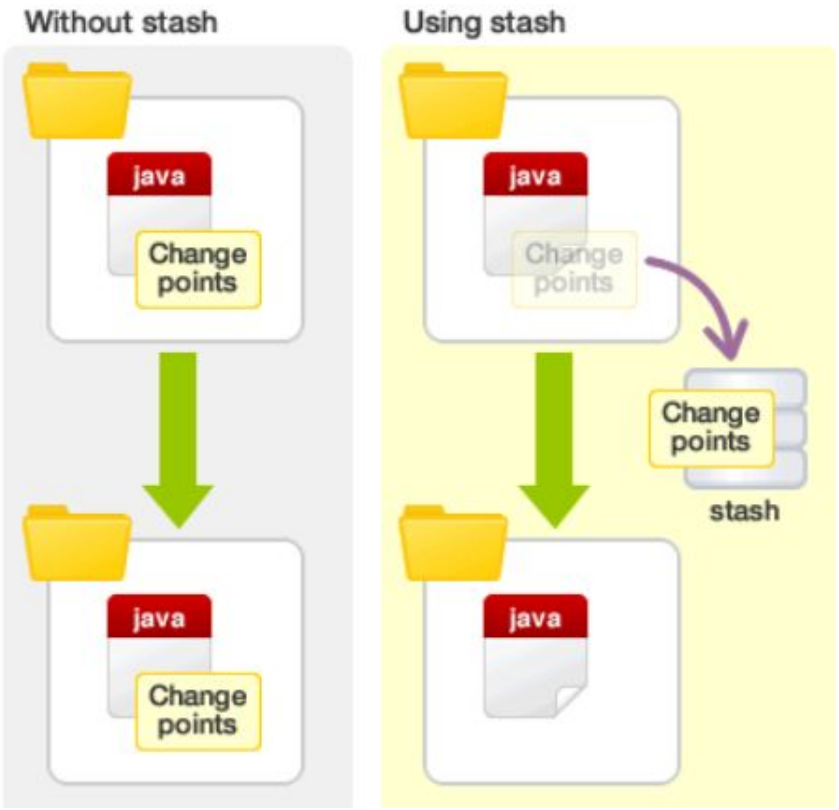
Checkout & branches

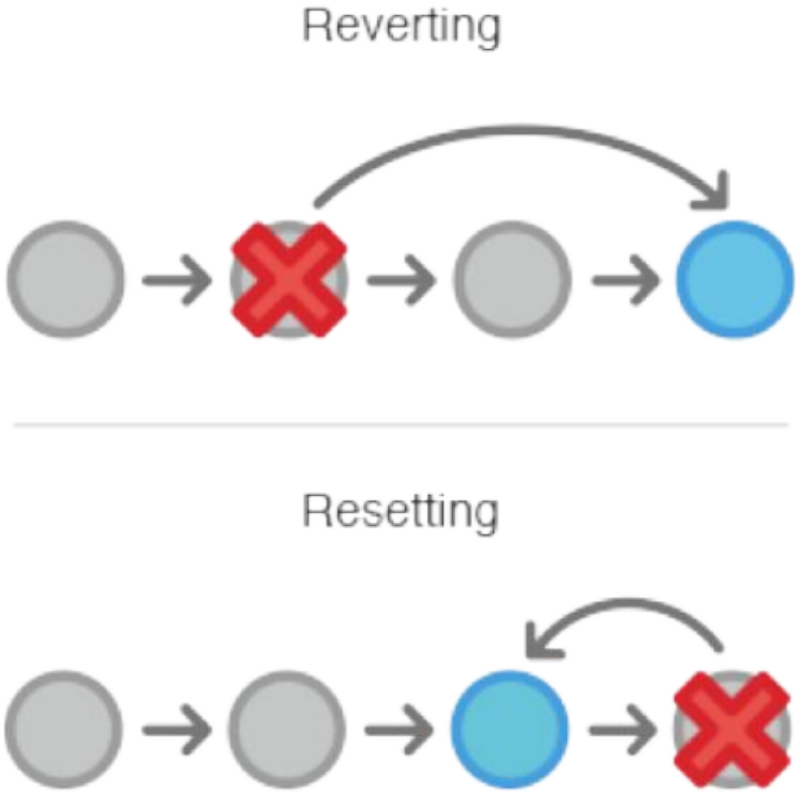
Checkout : used to navigate through commits and branches

Branch : branch can be listed, delete, etc.



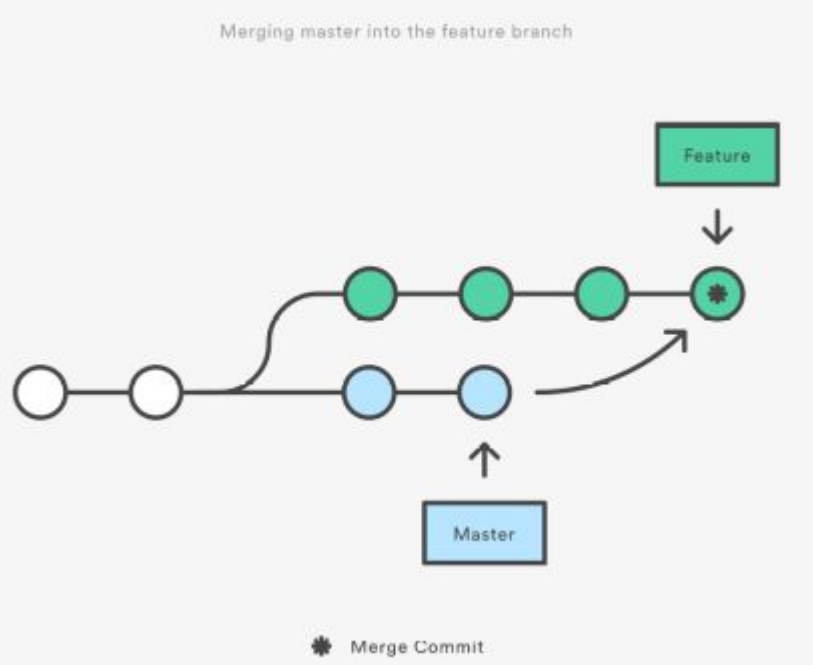
Stash, revert & Reset



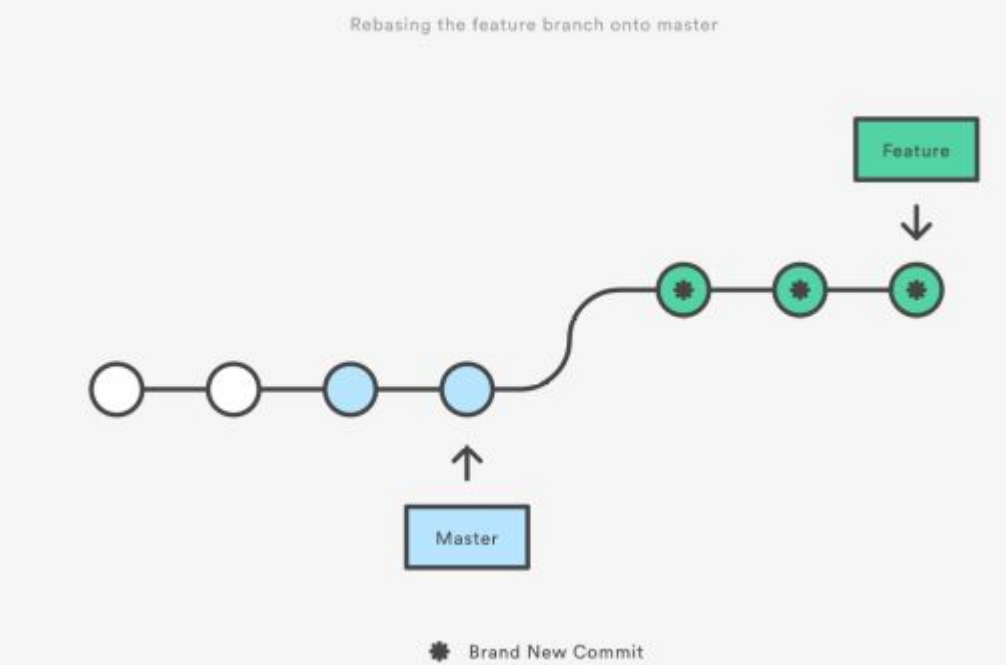


Merge & Rebase

Branches merge



Rebase



Tools

Clients :

Git Extension

Git Gui

Git Kraken

Turtoise git

Source tree

Sublime Merge

...

Server :

BitBake,

Bitbucket,

Gitlab,

Github

...

Ressource utile : http://rogerdudler.github.io/git-guide/

Exercices Git

1) Installer git
2) Créer un dossier vide, aller dedans et clone du repo Github
3) 2 par groupe (A et B) :
    a. Création d'une branche par A + ajout d'un fichier test avec « init » comme message.
    b. A ajoute à l'index, commit et push.
    c. B se synchronise à la branche
    d. Modification du fichier par A (message quelconque)
    e. Modification du fichier par B (message quelconque différent que celui de A)
    f. Chacun ajoute le fichier à son index et commit
    g. Afficher l'arbre pour vérifier
    h. A push sur le serveur
    i. B pull et résout les conflits (en mettant le sien en tête)
    j. Afficher l'arbre pour vérifier
    k. Tag la tête comme « avant échange »
    l. Afficher l'arbre pour vérifier
    m. Revert à « origin »
    n. Afficher l'arbre pour vérifier
    o. Modifier localement et supprimer ces modifications (pas d'ajout ou de commit)

Outils de développement

LES IDE

IDE

- Integrated development tool
- Software which include tools and functionalities created to improve and facilitate the development/maintenance of software.
- Liste of the included tools (for the most completes...)
  - Auto-completion tools (detection of libraries, class, object, etc...)
  - Auto-indentation tools
  - Coding rules tool
  - Debug tools
  - Building and packaging tools
  - Versionning tool and git integration

  ...

IDE List

- Jetbrains
- Android studio
- Visual Studio Code: Developed by Microsoft. Used for many languages. A lot of extension !
- Eclipse
- Atom: OpenSource editor developed by Github (with additional packages)
- Microsoft visual studio
- Sublime text (with additional packages)

...

Python POO

Created in 1991 by Guido van Rossum

Why Python:

- Easy to learn
- Big community
- Lots of complete libraries (https://pypi.org)
- One of the most popular language
- ! Syntax sensitive !
- No types

Declare a function

Simple

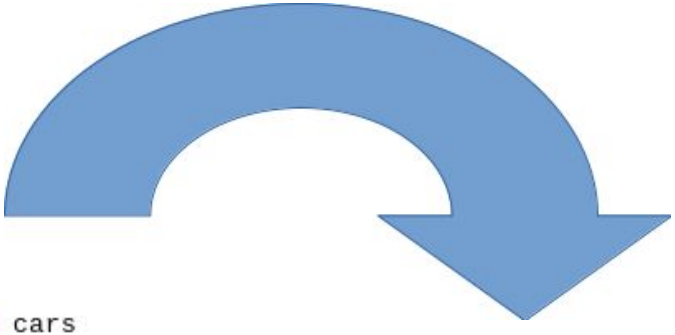
```
def add(a, b):
      return a + b

#Call to function
addition(1, 2)
#Printing
3
```

Splat List

```
def advanced_add(*param):
     return param[0] + param[1] + param[2]

#Call to function
advanced_add(10, 20, 30)
#Printing
60
```

```
def print_advanced_add(*param):
     print(param[0] + param[1] + param[2])

#Call to function
advanced_add(10, 20, 30)
#Printing
60
```

Create an object & import it



```
from controller import cars

my_car = Car()
print(m_car.wheels)
my_car.start()

#printing
4
The car is starting
```

```
#stored in controller/cars.py

class Car:
    wheels = 4
    engine = 1

    def start(self):
        print "The car is
starting"
```

Self

- Current object
- Used a lot in python Object
- example

```
my_car = Car() #Car object
my_car.my_car_function() #Will edit the « my_car » object
```

__init__ & __init__.py

1) __init__ function

```
class Voiture:
    wheels = 4
    engine = 1
    def __init__(self):
        self.nom = "to be defined"

    def start(self):
        print "the car is
starting"
```

2) __init__.py file

Example in model folder :

__init__.py

```
from sqlalchemy.ext.declarative import declarative_base

""" base class from which all mapped classes should inherit """
Base = declarative_base()
```

User.py

```
#getting the global variable « Base » from the __init__.py of the model
folder
from model import Base


class User(Base):
...
...
```

Heritage & overload

```
class Car:
      wheels = 4
      engine = 1
      def __init__(self):
            self.name = "To be defined"

      def start(self):
            print "the car is starting"


#Voiture is the mother of VoitureSport
class SportCar(Car):

      def __init__(self):
            self.name = "Ferrari"

      #Overload, or redefinition, of the «start» function
      def start(self):
            print "the car is starting"


my_car = SportCar()
my_car.wheels()
my_car.start()

#Printing
4
the car is starting
```

Exceptions and errors : Try, except and raise

Try to do an operation

Create an interruption in case of failing and raise it

```
def get(self, id):
        try:
            return self.get_user(id)
        except NoResultFound:
            #Raise the exception error (function defined by you)
            Raise Error("User not found")
```

Application au MVC

Model/users_model.py

```
class User:
  def __init__(self, database_session):
          self._database_session = database_session
    ...

  def get(self, id):
    try:
      # looking into table user and return only one user, with asked id
          return self._database_session.query(User).filter_by(id=id).one()
    except NoResultFound:
      raise ResourceNotFound()
```

Controller/users_controller.py

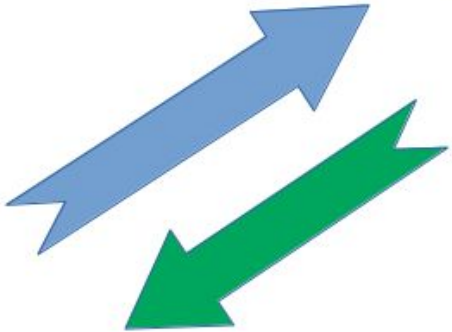
```
from model import users_model

class UserController:
  def __init__(self, database_session):
    ...

  def get(self, id):
          searched_user= User().get(id)

          return searched_user
```

view/users.py

```
from controller import users_controller

class User:
  def __init__(self, database_session):
    ...

  def get(self, id):
          wanted_user = UserController()
          #modify the string to be in    uppercase
          wanted_user.get().upper()
          return wanted_user
```

main.py

```
from view import users

first_user= User()
print(first_user.get(1))


if __name__ == "__main__":
  main()
```



SQL Alchemy

Simple library to create to relate Python classes with database tables.

It permit to create a small local database, with lot of functionalities already available.

Use a Session system for using the database.

Start a database

- Connecting
  - `>>> from sqlalchemy import create_engine`
  - `>>> engine = create_engine('sqlite:///:memory:', echo=True)`
- Declare a mapping
  - `>>> from sqlalchemy.ext.declarative import declarative_base`
  - `>>> Base = declarative_base()`
- And Now, you can use your base inside of your class :
  - `>>> from sqlalchemy import Column, Integer, String`
  - `>>> class User(Base):`
    - `...  __tablename__ = 'users'`
    - `...  id = Column(Integer, primary_key=True)`
    - `...  name = Column(String)`
    - `...  fullname = Column(String)`
    - `...  nickname = Column(String)`
  - `...`
    - `...  def __repr__(self):`
    - `...  return "<User(name='%s', fullname='%s', nickname='%s')>" % (`
    - `...  self.name, self.fullname, self.nickname)`

Session

Create

- `>>> from sqlalchemy.orm import sessionmaker`
- `>>> Session = sessionmaker(bind=engine)`
- `>>> Session.configure(bind=engine)  # once engine is available`

Adding/updating

- `>>> ed_user = User(name='ed', fullname='Ed Jones', nickname='edsnickname')`
- `>>> session.add(ed_user)`

Think to commit to save your modification !

- `>>> session.commit()`

Quering

- **for** name, fullname **in** session.query(User.name, User.fullname):
- ... **print**(name, fullname)

some useful common filter :

- query.filter(User.name == 'ed')
- query.filter(User.name.like('%ed%'))
- *# works with query objects too:*
- query.filter(User.name.in_(
- session.query(User.name).filter(User.name.like('%ed%'))
-))

and many more...

https://docs.sqlalchemy.org/en/13/orm/tutorial.html

Resources

- https://python.doctor/
- https://docs.python.org/3/
- https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
- https://www.qt.io/qt-for-python or https://docs.python.org/fr/3/library/tkinter.html
- https://courspython.com/bases-python.html
- Le base code utilise la librairie SQL alchemy pour communiquer avec la base de donnée. La création de la base est vous sera fourni pour les TD. https://docs.sqlalchemy.org/en/13/orm/tutorial.html

Questions ?

Hands on, you shall practice

creation of software for a clothing store

Specs

Customer

- Subscribe
- Browse article
- Fill his shopping cart with items
- Make a command
- Follow status of their commands

Seller

- Add and remove article
- Update article price, picture and description
- Valid and update status of customer commands
- Remove members

- Lister des idées de fonctionnement et d'évolution de ce projet.
- Réaliser le schéma d'architecture et de structure de code d'un tel projet en utilisant les notions de MVC

TD 1

- Initialiser le projet sur git
- Réaliser un petit logiciel de magasin (comme souhaité précédemment)
- En ligne de commandes :
  - Ajouter un membre,
  - Modifier un membre,
  - Lister les membres,
  - Supprimer un membre

Vous créerez une branche pour chacunes de ses fonctionnalités

Un utilisateur se compose : d’un nom, prénom, email, numéro de téléphone, mot de passe et aura le type “client”.

TD 2

- Ajouter la gestion des articles
- Ajouter la gestion des commandes

TD 3

- Mettre en place les test automatiques de vos fonctionnalités

TD 4

- Créer une interface pour votre programme : Au choix
  - Interface graphique (tkinter ou Qt)
  - Ou API REST