



GIT

In case of fire



1. git commit



2. git push



3. leave building

C koi ?

Un logiciel de gestion de version décentralisé

Utilisable en ligne de commande ou visuellement

Difficile à maîtriser dans les cas spécifiques mais indispensable à tout projet

Développé par Linus Torvald, aka le papa de Linux



POURQUOI?

ATTENTION

GIT



VS

GITHUB



Installation

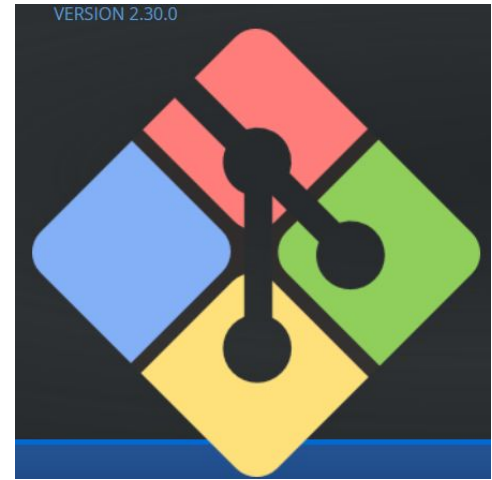
Sur Windows: installer Git Bash:

<https://gitforwindows.org/>

Sur Linux normalement il est installé par défaut sinon:

```
sudo apt install git
```

etc en fonction de votre distribution



Fonctionnement de base

On commence toujours par:

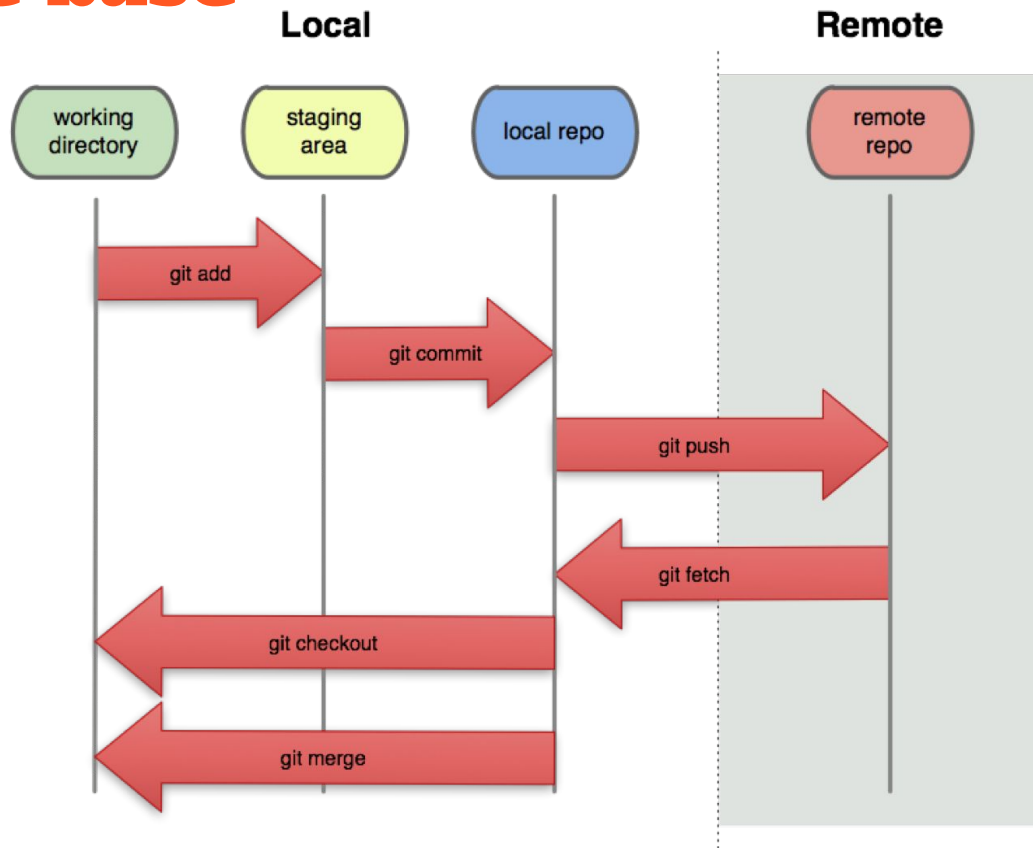
En local on crée son projet:

```
git init
```

ou

En reprenant un projet distant:

```
git clone <url_repo>
```



Fonctionnement de base

On ajoute les fichiers:

```
git add <fichier1> <fichier2>
```

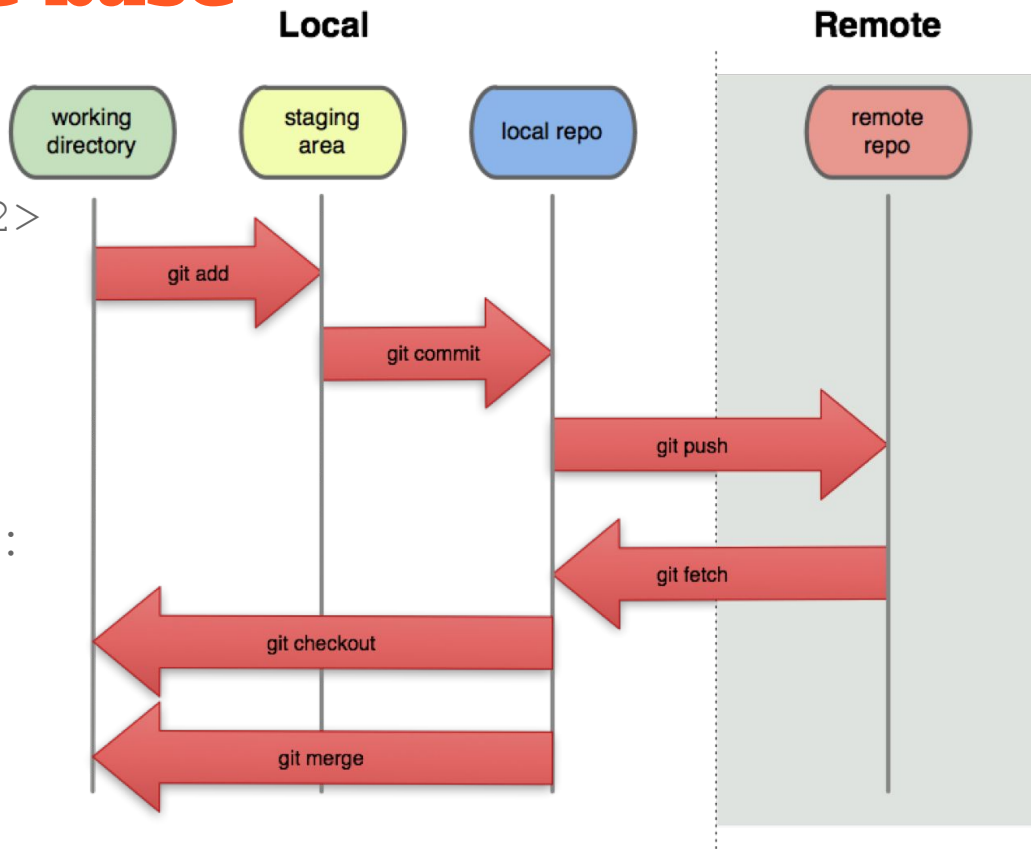
plus généralement :

```
git add *
```

puis on peut vérifier les fichiers ajoutés :

```
git status
```

au cas où : `git rm` et `git mv`



Fonctionnement de base

On peut alors enregistrer les changements :

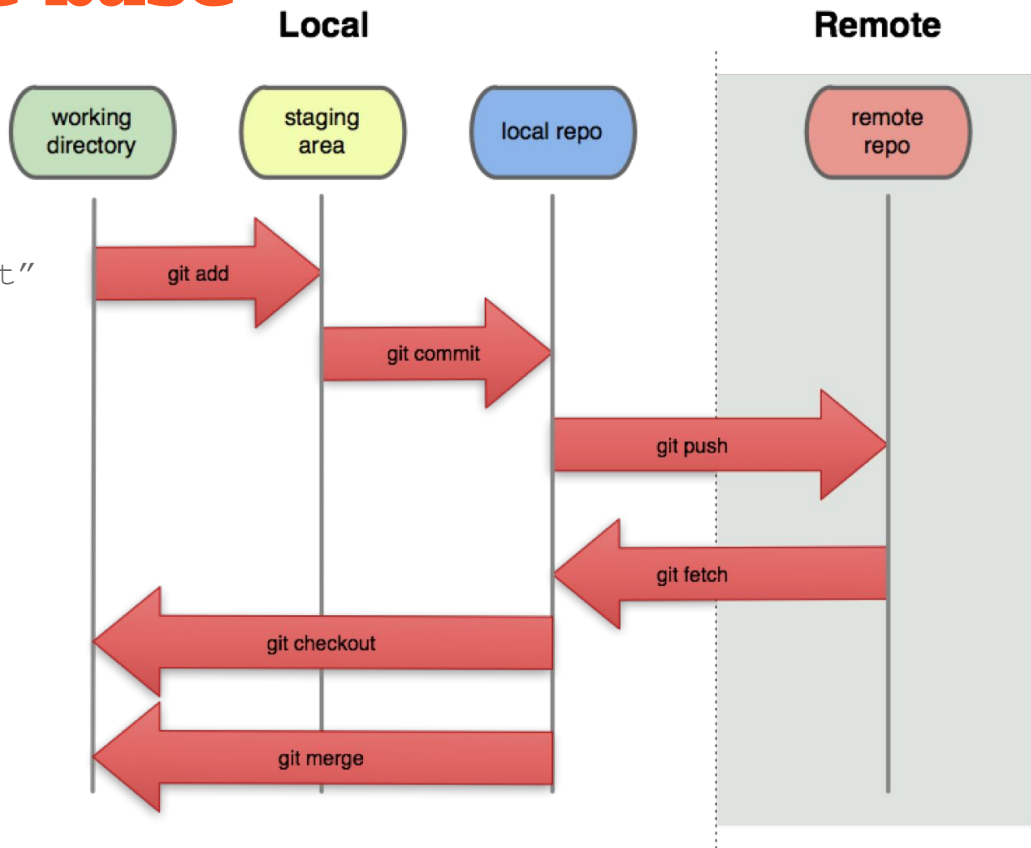
```
git commit -m "qu'est-ce que j'ai fait"
```

puis les envoyer en remote :

```
git push
```

Et aller chercher les modifications :

```
git pull
```



Les branches

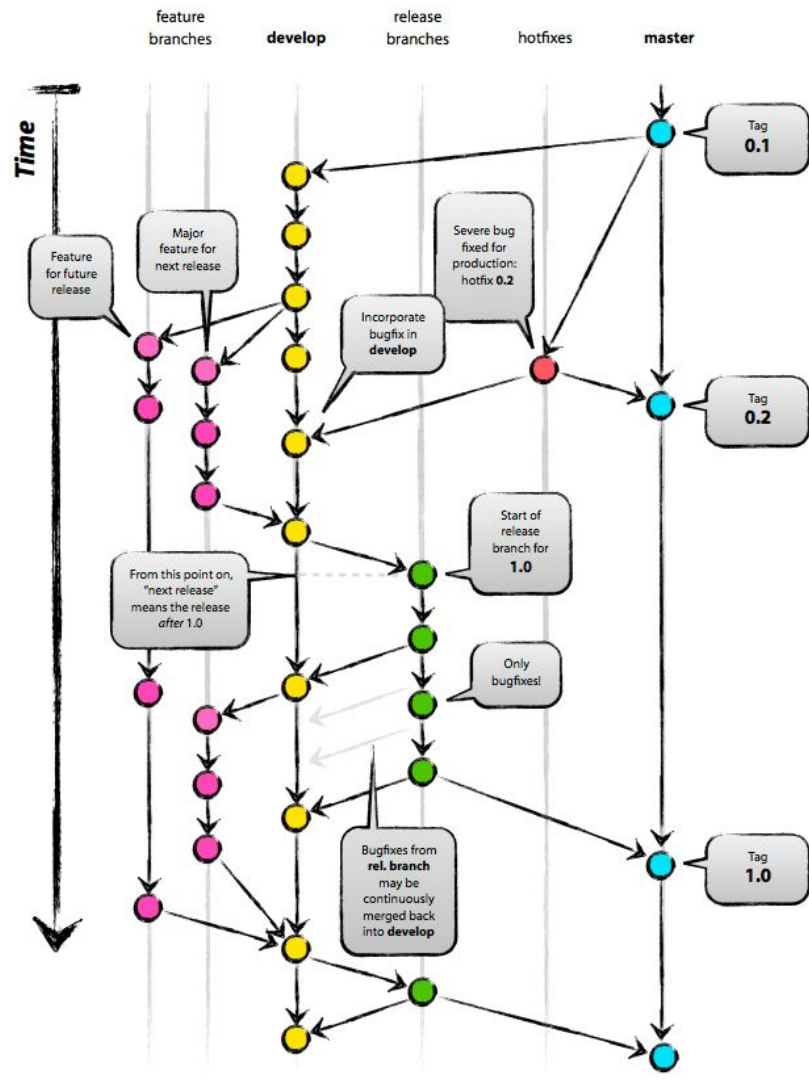
Objectif: mettre de l'ordre dans son projet

Commandes:

Créer: `git branch <nom_branche>`

ou `git checkout -b <nom_branche>`

Changer: `git checkout <nom_branche>`



Ça se complique...

<code>git add *</code> <code>git commit</code> <code>git push</code>	GitHubSetup.exe git init GitHub Desktop.app	git clone	
<code>git pull origin master</code> <code>git config</code> <code>git add foo</code>	<code>git remote add</code> <code>' .gitignore '</code> <code>'Merge made by the 'recursive' strategy.'</code>	<code>git status</code>	
<code>git reset --hard HEAD</code> <code>git branch</code> <code>rm -rf bar; git clone https://github.com/foo/bar</code> <code>git checkout</code> <code>git merge</code>	<code>'<<<<<< HEAD'</code> <code>git fetch</code> <code>git log</code> <code>git rm</code>		
<code>git diff</code> <code>git commit -am</code>	<code>git checkout -- foo</code> <code>git fetch upstream</code> <code>git merge upstream/master</code>	<code>'====='</code> <code>git remote -v</code> <code>git stash</code>	
<code>git push --force</code> <code>git push origin :foo</code> <code>git pull --rebase</code> <code>git submodule</code>	<code>git push origin HEAD:refs/for/master</code> <code>git cherry-pick</code> <code>git grep</code> <code>git blame</code> <code>git tag</code>	<code>'>>>>>> bar'</code> <code>git rebase -i</code>	
<code>git subtree</code> <code>git rev-parse --show-toplevel</code> <code>' .gitattributes '</code> <code>git merge -</code>	<code>cp pre-commit.sh .git/hooks/pre-commit</code> <code>' .git/info/exclude '</code> <code>'branch.master.mergeoptions = --no-ff'</code> <code>git branch --merged xargs git branch -d</code>	<code>git bisect</code>	
<code>git reset -p HEAD^</code> <code>git reflog</code> <code>git rerere</code> <code>git worktree</code>	<code>git update-index --assume-unchanged</code> <code>git daemon --reuseaddr --verbose --base-path= ./ .git</code> <code>git fsck</code> <code>git filter-branch</code>		
<p>'git gets easier once you get the basic idea that branches are homeomorphic endofunctors mapping submanifolds of a Hilbert space'</p>			

Naviguer entre différentes versions

```
git checkout <branche/id  
commit/HEAD...>
```

```
git diff <commit1> <commit2>
```

HEAD : dernier commit sur la branche actuelle

HEAD^ : avant-dernier commit

HEAD^^ : antépénultième commit

HEAD~2 : antépénultième commit

donc HEAD~4 = HEAD^^^^

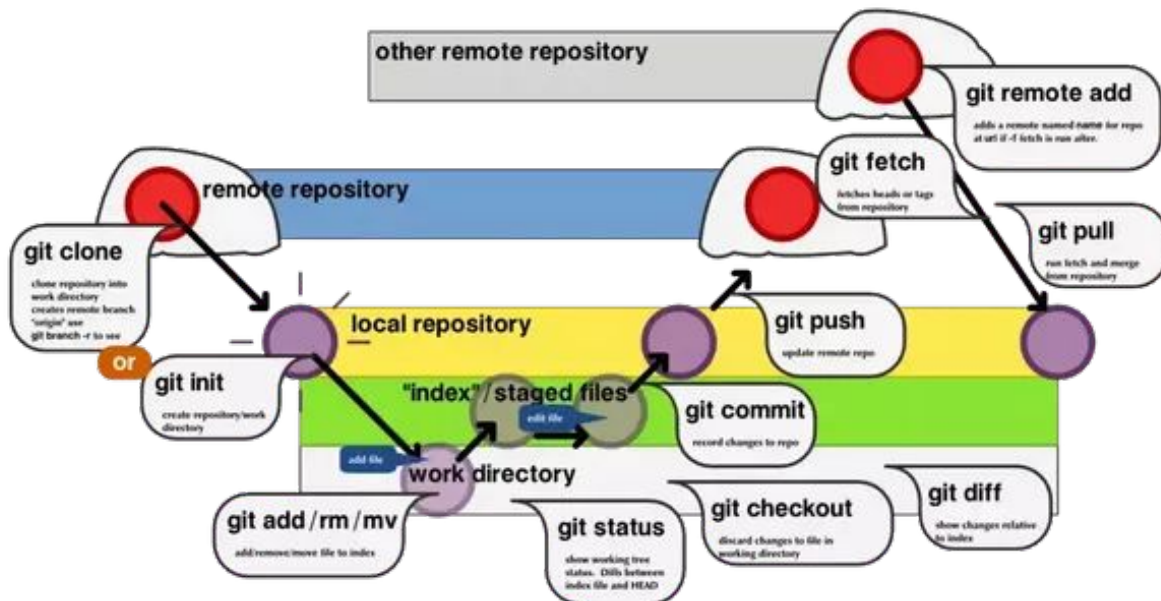
Commandes git

La base

- git init
- git clone
- git add <files>
- git rm <files>
- git commit -am "le message"
- git pull
- git push
- git branch <branch>
- git checkout <branch>
- git log
- git status

La difficulté

- git merge <branch>
- git rebase <branch>
- git stash



GIT Visual cheat sheet



git merge

Deux branches à joindre ? trop facile !

se positionner sur la branche principale puis :

```
git merge <branche_à_merge>
```

Bravo ! vous avez fait la fusion de deux branches

Si on veut mettre ses changements de côté sans commit, ou si on veut pull alors qu'on a des changements :

```
git stash
```

```
git stash apply
```



Les merge conflicts

Arrivent quand plusieurs personnes modifient différemment et en parallèle un même fichier

Plusieurs manières de les résoudre :

Manuellement

```
team_contact_info x
14 <tr>
15 <td>Harvey</td>
16 <td>Jennings</td>
17 <td>Leaving Soon</td>
18 <td>hjennings@atlassian.com</td>
19 <<<<<< HEAD
20 <td>2</td>
21 <td>B</td>
22
23 <td>2-B</td>
24 </tr>
25 <tr>
26 <td>Ryan</td>
27 <td>Lee</td>
28 <td>New Hire</td>
29 <td>rlee@atlassian.com</td>
30 <td>S-E</td>
31 >>>>>> 2a04e55405e527fb7924888b3ee0336a24849cf1
32 </tr>
33 <tr>
34 <td>Alana</td>
35 <td>Grant</td>
36 <td>Current</td>
37 <td>agrant@atlassian.com</td>
```



Depuis son IDE

```
Merge Revisions for C:\P\MergeDemoDevelopment\app\src\main\java\ca\cmpt276\mergedemo\NumberFun.java
Local Changes (Read-only) LF Result
public int getMin() {
    int min = data[0];
    for (int value : data) {
        if (value < min) {
            min = value;
        }
    }
    return min;
}

public int getAverage() {
    if (data.length == 0) {
        return 0;
    }
    int sum = 0;
    int i = 0;
    while (i < data.length) {
        sum += data[i];
        i++;
    }
    return sum / data.length;
}

public void printData() {
    // Print out the data
    for (int value : data) {
        System.out.print(value + ", ");
    }
    System.out.println();
}

// Print out the stats:
int avg = getAverage();
int min = getMin();

* NumberFun class manages some data an generates
* My Changes and Teammate's changes applied!
*/

public class NumberFun {
    private int[] data;
    public NumberFun(int[] data) {
        this.data = data;
    }

    public int getMin() {
        int min = 0;
        for (int i = 0; i < data.length; i++) {
            min = data[i];
        }
        return min;
    }

    public int getAverage() {
        int sum = 0;
        while (i < data.length) {
            sum += data[i];
            i++;
        }
        return sum / data.length;
    }

    public void printData() {
        // Print out the data
        for (int value : data) {
            System.out.print(value + ", ");
        }
        System.out.println();
    }

    // Print out the stats:
    int avg = getAverage();
    int min = getMin();
}

Changes from Server (revision 50314cc8638a07f4a488653d0752ec56f9... LF
6 changes, 2 conflicts
* NumberFun class manages some data an generates
* Teammate's changes applied!
*/

public class NumberFun {
    private int[] data;
    public NumberFun(int[] data) {
        this.data = data;
    }

    public int getMin() {
        int min = 0;
        for (int i = 0; i < data.length; i++) {
            min = data[i];
        }
        return min;
    }

    public int getAverage() {
        int sum = 0;
        while (i < data.length) {
            sum += data[i];
            i++;
        }
        return sum / data.length;
    }

    public void printData() {
        // Print out the data
        for (int value : data) {
            System.out.print(value + ", ");
        }
        System.out.println();
    }

    // Print out the stats:
    int avg = getAverage();
    int min = getMin();
}

Apply Abort
```


.gitignore

Liste les fichiers et dossiers qui seront ignorés par Git

ex: caches (inutile), fichiers confidentiels (sécurité), fichiers de configuration de l'IDE (inutile)

Syntaxe très simple :

- noms relatifs des fichiers/répertoires à ignorer
- / à la fin match uniquement des répertoires
- # permet de commenter
- ! prend la négation
- * match tout sauf \
- ? match un unique caractère sauf \
- ** spéciale :
 - <truc>/** -> tous les fichiers dans truc
 - **/<truc> -> englobe tous les répertoires menant à un truc
 - <machin>/**/<truc> -> englobe 0 ou plus répertoires entre machin et truc

<https://gitignore.io> : génère des .gitignore

```
1 # Byte-compiled / optimized / DLL files
2 __pycache__/
3 *.py[cod]
4 *$py.class
5
6 # C extensions
7 *.so
8
9 # Distribution / packaging
10 .Python
11 build/
12 develop-eggs/
13 dist/
14 downloads/
15 eggs/
16 .eggs/
17 lib/
18 lib64/
19 parts/
20 sdist/
21 var/
22 wheels/
23 share/python-wheels/
24 *.egg-info/
25 .installed.cfg
26 *.egg
27 MANIFEST
28
29 # PyInstaller
30 # Usually these files are written by a python script from a template
31 # before PyInstaller builds the exe, so as to inject date/other infos into it.
32 *.manifest
33 *.spec
34
35 # Installer logs
36 pip-log.txt
37 pip-delete-this-directory.txt
38
39 # Unit test / coverage reports
40 htmlcov/
41 .tox/
42 .nox/
43 .coverage
44 .coverage.*
45 .cache
46 nosetests.xml
47 coverage.xml
```


Concrètement

On ne git add pas tous les fichiers d'un projet ! —> on fait un **.gitignore**

On utilise les intégrations git de son IDE par exemple pour gérer les merge conflicts

On crée des branches (intelligemment)

Conventions utiles:

- **README.md**
- requirements

Apprendre en pratiquant

Les utilisations des branches visuellement: <https://learngitbranching.js.org/>

Les commandes de git en général: <https://gitexercises.fracz.com/>

Autres ressources

Un super site résumant les commandes utiles en cas de scénario terrible:

<https://ohshitgit.com/>

Le résumé des commandes essentielles:

<https://education.github.com/git-cheat-sheet-education.pdf>

Guide de survie en remote

Merge requests

Fork

Issues

CI/CD

License

L'instant sécu avec HackademINT

Comme avec n'importe quoi dans le monde de l'informatique, une mauvaise utilisation de git peut mener à des vulnérabilités !

- Vulnérabilité de l'exécutable git
- Vulnérabilité des hébergeurs (github, gitlab, gitea...)
- Mauvaises utilisations de git menant à des vulnérabilités



Ne PAS développer en production

git stocke toutes les informations nécessaires à son fonctionnement dans un dossier caché .git ...

```
[10:08|mh4ck@archlinux]:projet_de_dev$ ls -a
.  ..
[10:09|mh4ck@archlinux]:projet_de_dev$ git init -q
[10:09|mh4ck@archlinux]:projet_de_dev$ rm -rf .git/
[10:09|mh4ck@archlinux]:projet_de_dev$ git init
astuce: Utilisation de 'master' comme nom de la branche initiale. Le nom de la branche
astuce: par défaut peut changer. Pour configurer le nom de la branche initiale
astuce: pour tous les nouveaux dépôts, et supprimer cet avertissement, lancez :
astuce:
          git config --global init.defaultBranch <nom>
astuce:
astuce: Les noms les plus utilisés à la place de 'master' sont 'main', 'trunk' et
astuce: 'development'. La branche nouvellement créée peut être renommée avec :
astuce:
          git branch -m <nom>
Dépôt Git vide initialisé dans /home/mh4ck/Documents/HackademINT/projet_de_dev/.git/
[10:09|mh4ck@archlinux]:projet_de_dev$ ls -a
.  ..  .git
[10:09|mh4ck@archlinux]:projet_de_dev$ echo "Bonjour" > index.html
[10:10|mh4ck@archlinux]:projet_de_dev$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Ne PAS développer en production

... ce qui peut mener à de grosses vulnérabilités si ce dossier est rendu accessible !!



bonjour



Directory listing for /.git/

PAS BIEN : il est possible de récupérer l'intégralité du repo, dont le code source, tous les commits, toutes les branches...!

- [branches/](#)
- [config](#)
- [description](#)
- [HEAD](#)
- [hooks/](#)
- [info/](#)
- [objects/](#)
- [refs/](#)

Réfléchir au contenu des commits

- Fichiers de configuration contenant des clés ou des mots de passe rentrés à la main (config.php, config.xml, .env, .config ...)
- Fichiers de configuration de plugins ou d'extensions avec des clés générées aléatoirement et stockées dans votre projet (CMS, extensions PHP, extensions VSCode...) et aussi des secrets (tokens) GitHub, GitLab... pouvant être stockés dans votre projet et commit par erreur
- Clés personnelles (id_rsa, password.txt, MDPsDeLaBanque.kdb ...)
- Attention aux commentaires avec des infos sensibles dedans : ils sont aussi lisibles si on récupère le repo
- Je Fals De La SeCu CaR jE CoMmlt PaS dEs FiChleRs SeNsIbLeS mais je file des informations confidentielles dans les noms des commits (“astuce” utilisée par de nombreuses personnes) : NE FONCTIONNE PAS ! Avec la vulnérabilité présentée juste avant, on récupère même le nom et les messages des commits

=> De manière générale, on **réfléchit** avant de faire un git add * !!

C'était l'instant sécu avec HackademINT



Des questions ?

Bon TP à tous / toutes, et n'oubliez pas de réfléchir à ce que vous faites ;-)