



Projet dynamique

Git - Github

Le versioning



Introduction
Fonctionnement
Commandes de base

Principes de base

Application de gestion des versions

VCS en anglais pour Version Control System

Un gestionnaire de version est un système qui enregistre l'évolution d'un fichier ou d'un ensemble de fichiers au cours du temps de manière à ce qu'on puisse rappeler une version antérieure d'un fichier à tout moment.

Il crée des instantanés de l'évolution de votre projet que vous pouvez récupérer si vous constatez des dysfonctionnements ou des erreurs majeures.

Ne confondez pas **Git** et **GitHub**:

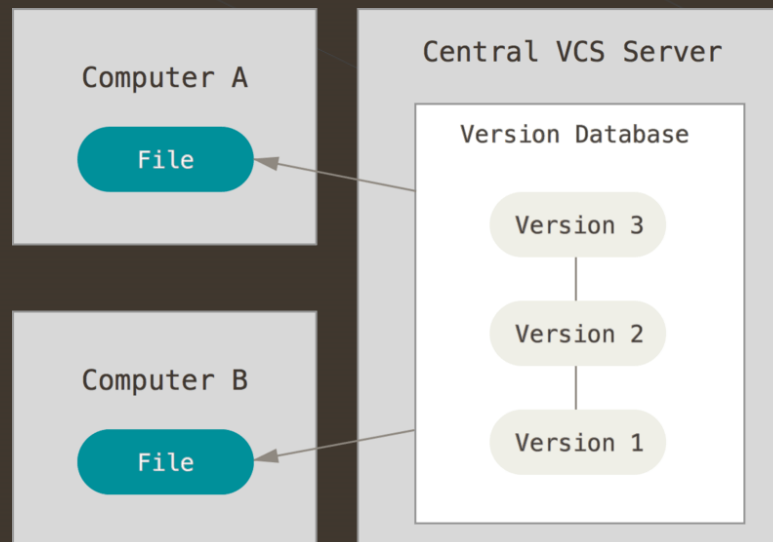
Git est l'application de versioning et GitHub qui un des services de dépôt en ligne.

Avantages

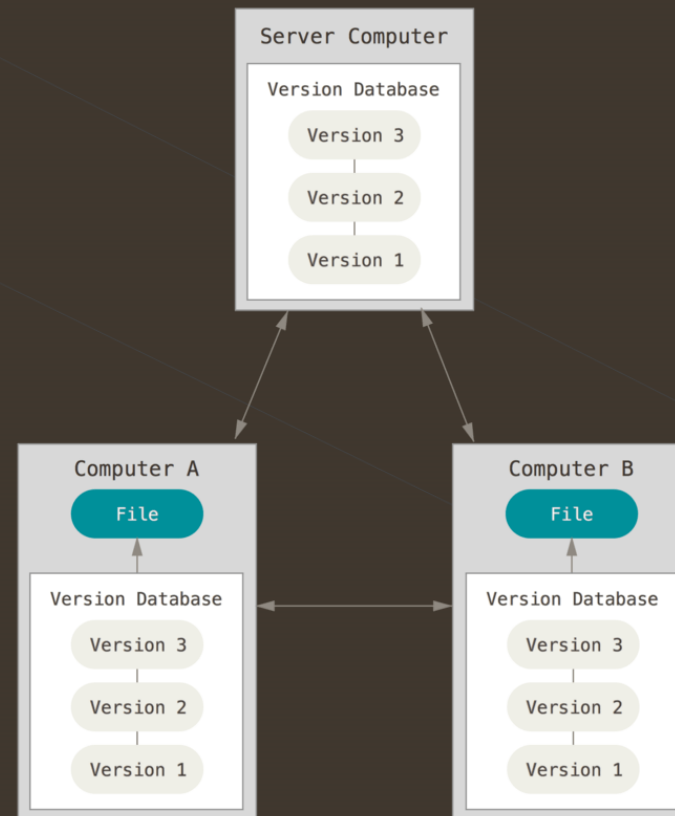
- De ce type d'application, on peut retirer trois grandes fonctionnalités
 - travailler à plusieurs sans risquer de supprimer les modifications des autres collaborateurs;
 - revenir en arrière en cas de problème;
 - suivre l'évolution étape par étape d'un code source pour retenir les modifications effectuées sur chaque fichier.

Logiciel centralisé ou distribué ?

- Il existe deux types principaux de logiciels de gestion de versions.
- Les logiciels centralisés : un serveur conserve les anciennes versions des fichiers et les développeurs s'y connectent pour prendre connaissance des fichiers qui ont été modifiés par d'autres personnes et pour y envoyer leurs modifications.

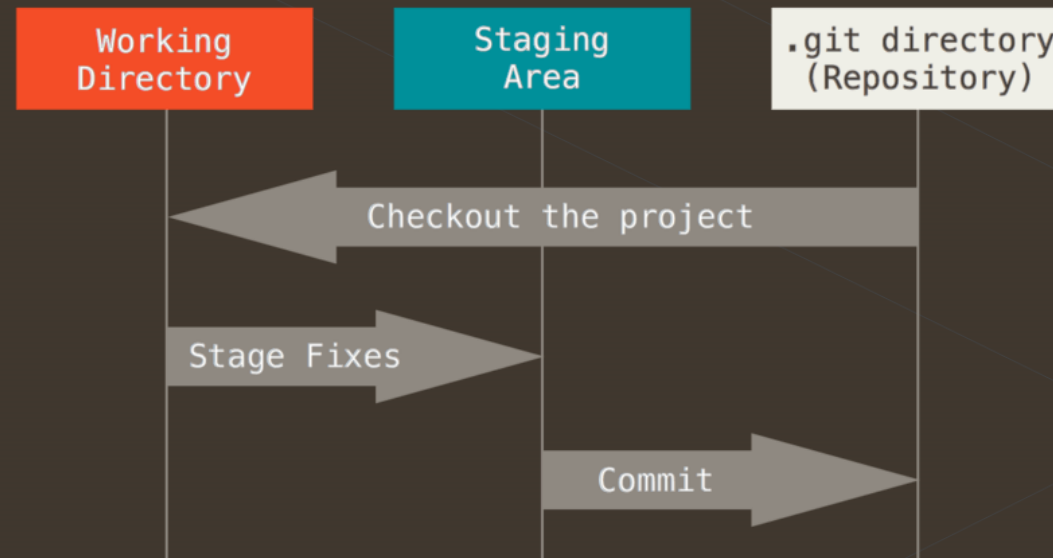


- Les systèmes de gestion de versions distribués
- les clients n'extraient plus seulement la dernière version d'un fichier, mais ils dupliquent complètement le dépôt. Ainsi, si le serveur disparaît et si les systèmes collaboraient via ce serveur, n'importe quel dépôt d'un des clients peut être copié sur le serveur pour le restaurer. Chaque extraction devient une sauvegarde complète de toutes les données.

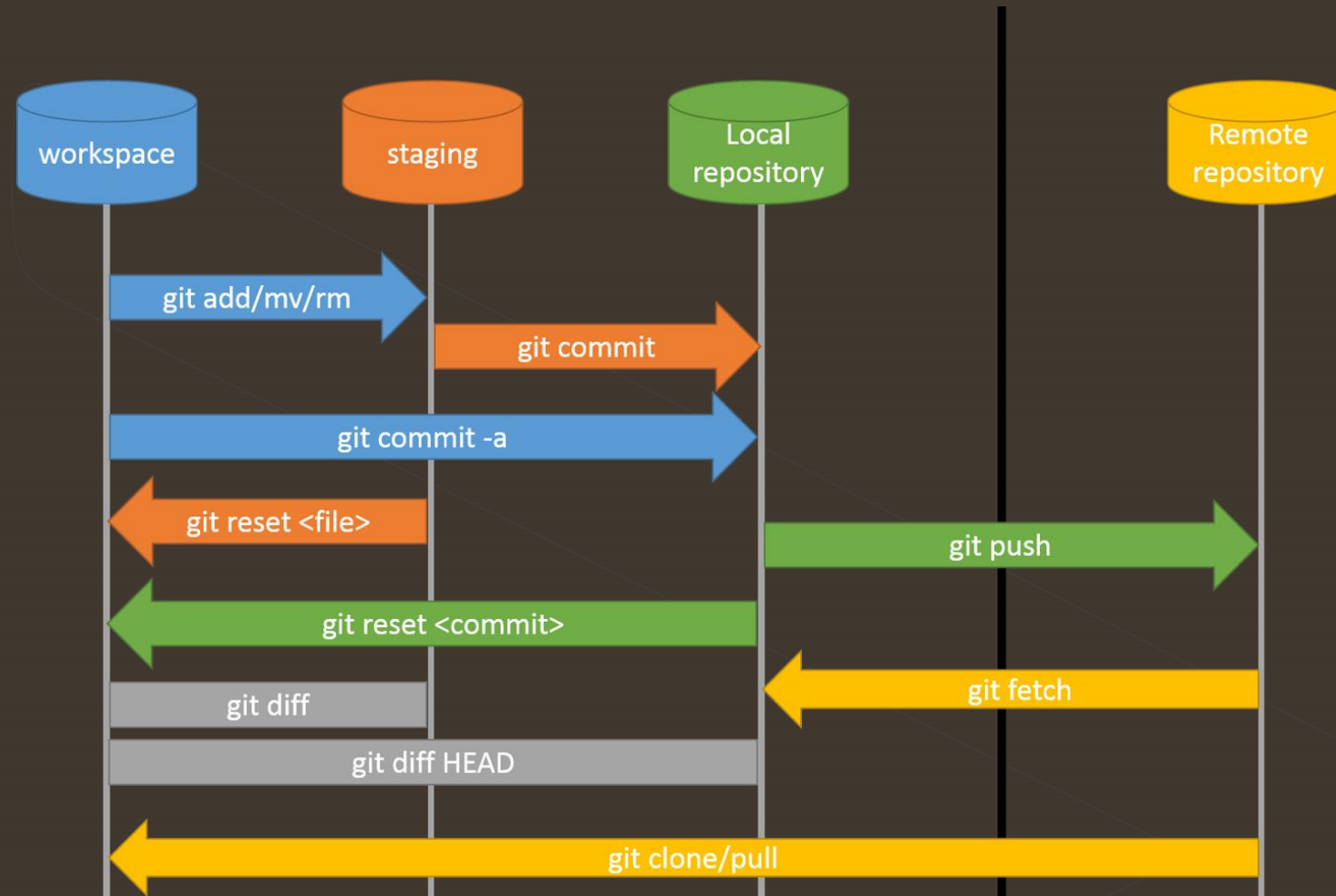


Fonctionnement général

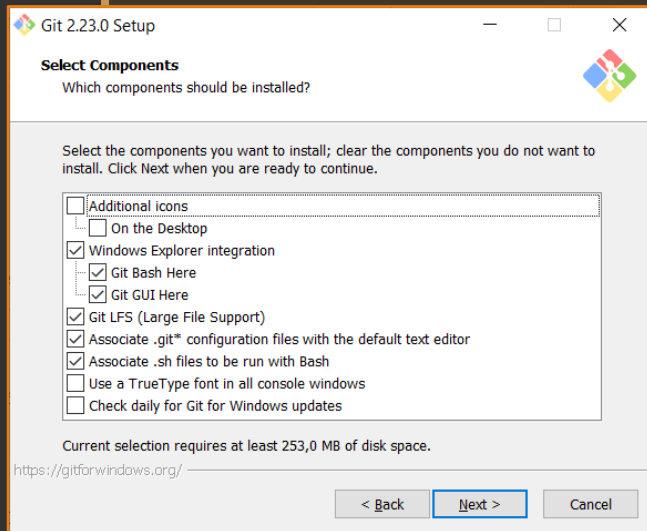
- Presque toutes les opérations sont locales et comme vous disposez de l'historique complet du projet sur votre disque dur, la plupart des opérations semblent instantanées.
- L'utilisation standard de Git se passe comme suit :
 - vous modifiez des fichiers dans votre répertoire de travail;
 - vous indexez les fichiers modifiés, ce qui ajoute des instantanés de ces fichiers dans la zone d'index;
 - vous validez, ce qui a pour effet de basculer les instantanés des fichiers de l'index dans la base de données du répertoire Git.



Fonctionnement via les commandes



Installation sous Windows



- Rendez-vous sur le site officielle pour télécharger la dernière version: <https://git-scm.com/download/win>
- Sélectionnez « Git Bash » et « Git Gui »
- Laissez les autres options par défaut.
- Testez le bon fonctionnement dans **Git Bash** avec la commande: **git --help**

Paramétrage à la première utilisation

- Une fois Git installé sur votre système, vous devez personnaliser votre environnement Git. Vous ne devriez avoir à réaliser ces réglages qu'une seule fois ; ils persisteront lors des mises à jour. Vous pouvez aussi les changer à tout instant en relançant les mêmes commandes.
- Git contient un outil appelé **git config** pour vous permettre de voir et modifier les variables de configuration qui contrôlent tous les aspects de l'apparence et du comportement de Git.

Git config

- La première chose à faire après l'installation de Git est de renseigner votre nom et votre adresse email dans deux variables. Si vous disposez d'un dépôt distant inscrivez les données que vous avez fournies et non de fausses informations.
- `git config --global user.name "John Doe"`
- `git config --global user.email john@doe@example.com`
- Vérifiez les données avec: `git config --list`

Démarrer un dépôt Git

- Vous pouvez principalement démarrer un dépôt Git de deux manières. La première consiste à prendre un projet ou un répertoire existant et à l'importer dans Git. La seconde consiste à cloner un dépôt Git existant sur un autre serveur.

Initialisation d'un dépôt Git dans un répertoire existant

- Si vous commencez à suivre un projet existant dans Git, vous n'avez qu'à vous positionner dans le répertoire du projet et saisir :
- `git init`
- Cela crée un nouveau sous-répertoire nommé `.git` qui contient tous les fichiers nécessaires au dépôt — un squelette de dépôt Git. Pour l'instant, aucun fichier n'est encore versionné.

Démarrer un dépôt Git

Cloner un dépôt existant

- Si vous souhaitez obtenir une copie d'un dépôt Git existant la commande dont vous avez besoin s'appelle `git clone [url]`.
- Ceci crée un répertoire portant le même nom que le dépôt, initialise un répertoire `.git` à l'intérieur, récupère toutes les données de ce dépôt, et extrait une copie de travail de la dernière version.

Vérifier l'état des fichiers

- L'outil principal pour déterminer quels fichiers sont dans quel état est la commande **git status**. Si vous lancez cette commande juste après un git init, vous devriez voir ce qui suit :

On branch master

No commits yet

Untracked files:

(use "git add <file>..." to include in what will be committed)

.idea/

nothing added to commit but untracked files present (use "git add" to track)

Placer de nouveaux fichiers sous suivi de version

- Commençons par créer un nouveau fichier **readme.md** dans la racine de notre projet. Si PhpStorm demande d'indexer le fichier refusez, nous procéderons manuellement pour une meilleure compréhension.
- Ecrivez pour l'exemple un simple texte et exécutez la commande `git status`
 - On branch master
 - No commits yet
 - Untracked files:
 - **.idea/**
 - **readme.md**

Readme.md apparaît dans la liste des fichiers non suivis

- Pour commencer à suivre un nouveau fichier, vous utiliserez la commande `git add readme.md`
- Avec `git status`, vous constaterez que le fichier n'est plus dans la zone non suivie "Untracked files" mais bien dans la zone "Modifications qui seront validées"

Changes to be committed:
(use "git rm --cached <file>..." to unstage)
new file: readme.md

- La commande `git add -all` permet d'indexer l'ensemble des fichiers.

Désindexer un fichier

- Si vous souhaitez retirer un fichier ou un dossier de la « staging area » utilisez la commande `git reset <fichier>` ou `git reset --all`

Modifier un fichier indexé

- Si vous modifiez le contenu d'un fichier déjà indexé et que vous saisissez `git status`, vous constaterez que des modifications ont été effectuées sur le fichier:

Changes not staged for commit:

(use "`git add <file>...`" to update what will be committed)

(use "`git restore <file>...`" to discard changes in working directory)

modified: `readme.md`

- Vous devez alors le réindexer avec la commande `git add`
- Si vous souhaitez annuler la dernière modification depuis son indexation utilisez la commande `git restore <file>`

Ignorer des fichiers

- Il arrive souvent que certains fichiers présent dans la copie de travail ne doivent pas être ajoutés automatiquement ou même ne doivent pas apparaître comme fichiers potentiels pour le suivi de version. Ce sont par exemple des fichiers générés automatiquement tels que les fichiers de logs ou de sauvegardes produits par l'outil que vous utilisez. Dans ce cas, on peut indiquer leurs noms ou définir un modèle (patrons) dans le fichier `.gitignore`.

Exemples de .gitignore

- Créez un fichier **.gitignore** dans votre projet
- Ajoutez y les fichiers à ignorer sous la forme de fichiers ou de patrons.
- Exemples:
 - *.txt
 - readme.md
 - tmp/ (ignore tous les fichiers dans le répertoire)
 - doc/**/*.*txt (ignorer tous les fichiers .txt sous le répertoire doc/)
 - .idea
 - .gitignore

Testez le .gitignore

- Avant la création du .gitignore, si vous lancez un git status vous obtiendrez dans la liste des fichiers non suivis le dossier système de PhpStorm: `.idea/`
- Vous devez absolument l'ajouter au `.gitignore` ça ne concerne pas votre projet.
- Après la création du gitignore ils n'apparaîtront plus même dans la zone des fichiers non suivis.

Commit

Valider vos modifications

Les commits

- Nous allons ajouter quelques fichiers dans notre projet:
 - `css/style.css`
 - `index.php`
- Ajoutez les fichiers pour indexation: `git add -all`
- Maintenant que votre zone d'index est dans l'état désiré, vous pouvez valider vos modifications. Souvenez-vous que tout ce qui est encore non indexé — tous les fichiers qui ont été créés ou modifiés mais n'ont pas subi de `git add` depuis que vous les avez modifiés — ne feront pas partie de la prochaine validation. Ils resteront en tant que fichiers modifiés sur votre disque.
- Saisissez `git commit -m « Votre message »`
- Si vous vérifiez avec `git status`, vous verrez le message suivant:

`On branch master`

`nothing to commit, working tree clean`

Commit rapide pour fichiers modifiés

- Si vous ne faites que modifier des fichiers se trouvant déjà la « Staging area » vous pouvez saisir la commande:
- `git commit -a -m « Message du commit »`
- Ainsi vous évitez de ressaisir la commande `git commit add -a`

Visualiser l'historique des validations

- Après avoir créé plusieurs commits ou si vous avez cloné un dépôt ayant un historique de commits, vous souhaitez probablement revoir le fil des événements. Pour ce faire, la commande `git log` est l'outil le plus basique et le plus puissant.
- Par défaut, `git log` invoqué sans argument énumère en ordre chronologique inversé les commits réalisés. Cela signifie que les commits les plus récents apparaissent en premier. Comme vous le remarquerez, cette commande indique chaque commit les contrôles SHA-1, le nom et l'e-mail de l'auteur, la date et le message du commit.

git log

- commit 2984efa20c155bf567551c62714d8f9c6df01722 (HEAD -> master)
- Author: John Doe <johndoe@example.com>
- Date: Wed Oct 23 21:35:32 2019 +0200
- premier commit

git log

- Un commit correspond à une étape du travail. Evitez de faire de commits à tout bout de champs sinon l'historique devient ingérable.
- Souvenez-vous que la validation enregistre l'instantané que vous avez préparé dans la zone d'index. Tout ce que vous n'avez pas indexé est toujours en état modifié. vous pouvez réaliser une nouvelle validation pour l'ajouter à l'historique.
- Vous pourrez revenir facilement sur un instantané précédent si vous avez des problèmes.

Git log et les options

- `git log` dispose d'un très grand nombre d'options permettant de paramétrer exactement ce que l'on cherche à voir.
- `git log -p` (montre les différences introduites entre chaque validation)
- `git log -2` (limite la sortie de la commande aux deux entrées les plus récentes)
- `git log -stat` (affiche un résumé des modifications fichier par fichier)
- `git log --pretty=oneline` (affiche chaque commit sur une seule ligne)
- `git log --oneline` (raccourci)

Git diff

- **Git diff** (affiche le détail des modifications apportées sur les fichiers et le dernier commit)



Revenir en arrière

Modifier le dernier message de commit ou ajouter des fichiers

- Une des annulations les plus communes apparaît lorsqu'on valide une modification trop tôt en oubliant d'ajouter certains fichiers, ou si on se trompe dans le message de validation. Si vous souhaitez rectifier cette erreur, vous utiliserez l'option --amend
- `git commit --amend` ou `git commit --amend -m « Nouveau message »`
- Cette commande prend en compte la zone d'index et l'utilise pour le commit. Si aucune modification n'a été réalisée depuis la dernière, alors l'instantané sera identique et la seule modification à introduire sera le message de validation.
- Vous n'aurez au final qu'un unique commit, la seconde validation remplace le résultat de la première.
- Pour sortir de l'éditeur exécutez « :WQ » ou « CRL + X »

La commande checkout

- Elle permet de visualiser l'état d'un fichier ou d'un commit ancien dans votre zone de travail. Il s'agit seulement d'observer en tant que spectateur des anciens fichiers mais pas de revenir véritablement en arrière.
- `git checkout <commit>` (id du commit)
- `git checkout master` (revenir au dernier instantané)
- `git checkout <commit> <file>` (récupérer un fichier précis d'un commit et le place dans l'index). Vous pouvez ainsi refaire un commit du fichier avant les dernières modifications. Pour annuler l'opération, saisissez à nouveau la même commande.

Inverser un commit.

- `git revert <commit>`
- Cette commande va défaire ce qui avait été fait au moment du `<commit>` en créant un nouveau commit. Cela n'altère pas l'historique mais va ajouter un nouveau commit d'inversion concernant uniquement l'id du commit indiqué.
- Utilisé généralement pour annuler le dernier commit sans modifier l'historique.

Revenir en arrière

- **Supprimer un commit en laissant la zone de travail intacte**
- `git reset <commit>` (id du commit)
- Permet de revenir en arrière jusqu'au <commit>, réinitialise la zone de staging tout en laissant votre dossier de travail en l'état. L'historique sera perdu (les commits suivant <commit> seront perdus, mais pas vos modifications). Cette commande vous permet surtout de nettoyer l'historique en re-soumettant un commit unique à la place de commit multiples et sans fondement.

Supprimer un commit définitivement

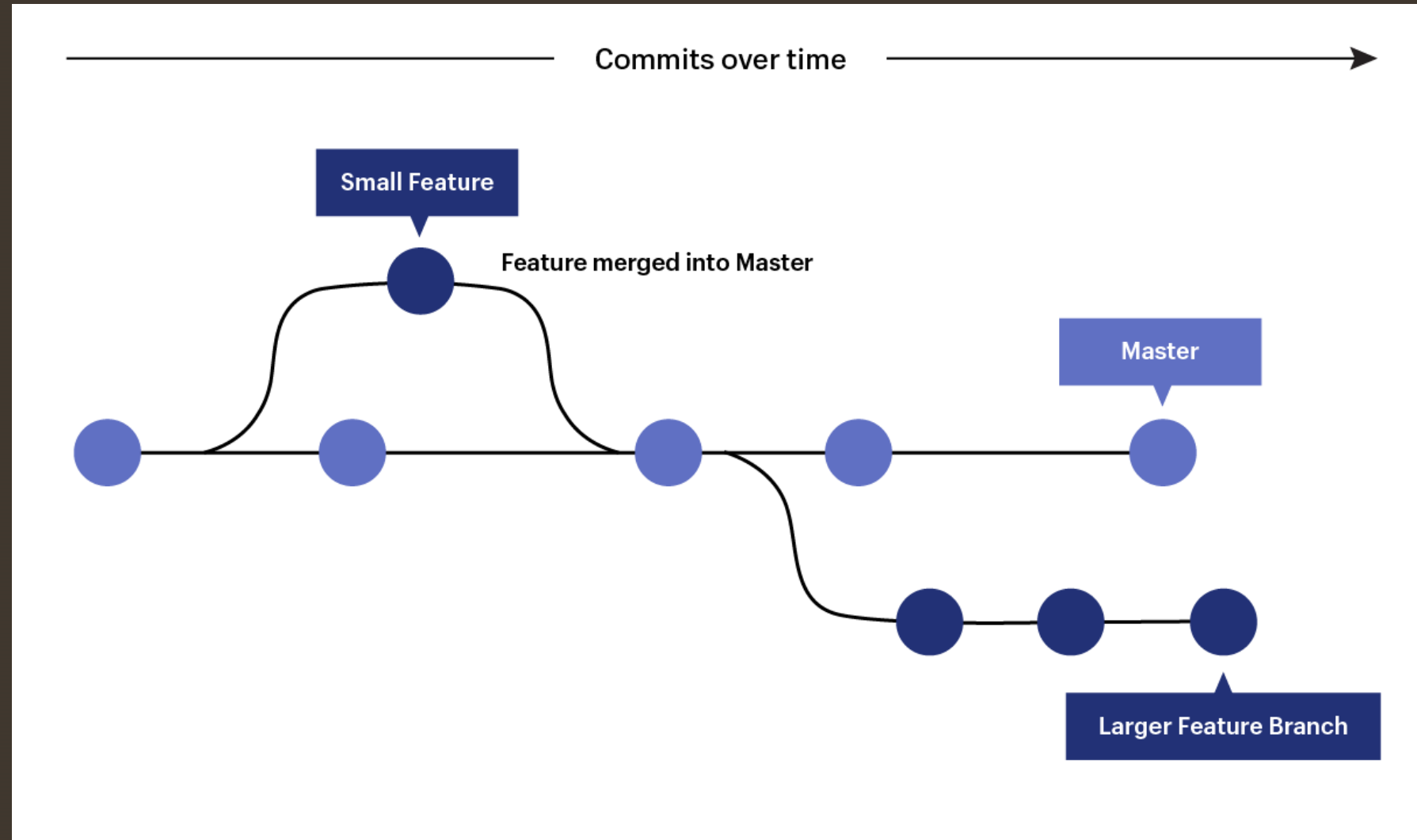
- `git reset <commit> --hard`
- Permet de revenir au <commit> et réinitialise la zone de staging et le dossier de travail pour correspondre. Toutes les modifications, ainsi que tous les commits fait après le <commit> seront supprimés. A utiliser avec une extrême précaution !



Les branches

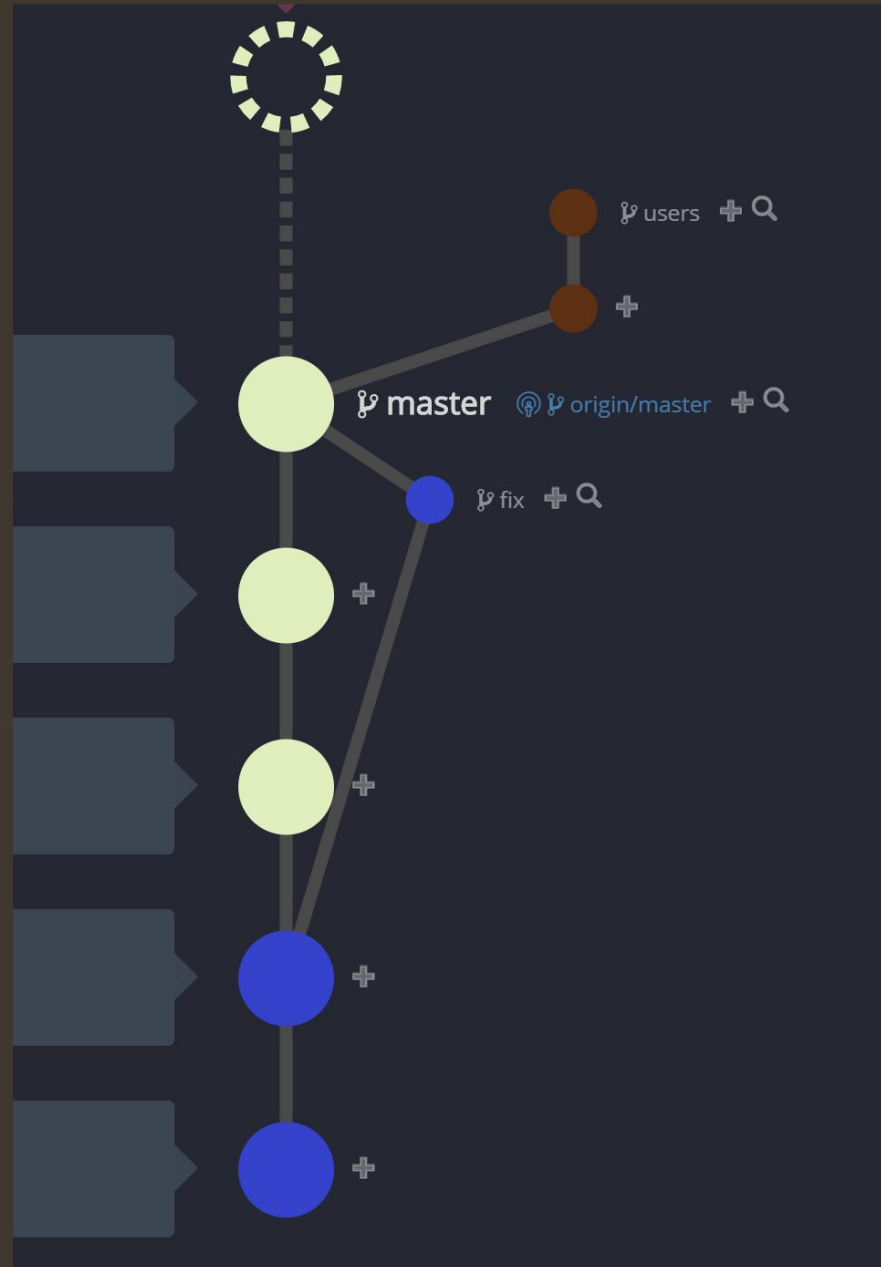
Principe

- Dans Git, toutes les modifications que vous faites au fil du temps sont par défaut considérées comme appartenant à la branche principale appelée « **master** »
- Les branches sont un moyen de travailler en parallèle sur d'autres fonctionnalités ou de tester du code sans être certain que ça va fonctionner et sans perturber la branche principale.



Utilitaire

- Quand on travaille sur des branches, il est intéressant de visualiser graphiquement l'historique de tout le projet.
- L'application open source Ungit est simple et peut-être exécuté directement dans votre terminal:
<https://github.com/FredrikNoren/ungit>
- L'installation se fait avec NPM: `npm install -g ungit`
- Ensuite saisissez `ungit` dans le terminal !!!



Création

- `git branch <branche>` (Permet de créer une nouvelle branche <branche>)
- `git branch` (Permet de lister les branches)
- `git branch -m <branche>` (Renomme la branche courante en <branche>)
- `git branch -d <branche>` (Permet de supprimer une branche)

Changer de branches

- Pour se rendre sur une branche existante, utilisez la commande checkout
- `git checkout <branche>`
- `git checkout master` (Pour retourner sur la branche principale)

Fusionner les branches

- La commande **merge** permet de ramener une branche sur une autre et ainsi de la fusionner. La fusion de 2 branches se fait toujours à partir de la branche principale.
- **git merge <branche>**
- La branche principale (master) sera affectée en récupérant l'historique de la branche ou un commit de fusion.



Exercise



Consignes

- Créez un nouveau projet gitex
- Ajoutez une structure de dossiers: css – js – img
- Ignorer le dossier système de PhpStorm
- Indexez votre structure et créez votre commit « structure project »
- Créez un fichier readme.md avec un titre et indexez-le
- Créez deux fichiers: main.css et bootstrap.css
- Retirez readme de l'index et ignorez-le
- Créez le commit « Structure CSS »
- ...



Les dépôts distants

Introduction

- C'est avec la commande push que vous allez envoyer vos commits sur votre dépôt.
- Un push est irréversible. Une fois que vos commits sont publiés, il deviendra impossible de les supprimer ou de modifier le message de commit ! Réfléchissez donc à deux fois avant de faire un push. C'est pour cela que je recommande de ne faire un push qu'une fois par jour, afin de ne pas prendre l'habitude d'envoyer définitivement chacun de vos commits trop vite.

Dépôt distant - GitLab


- Après la création du compte vous devez créer un dépôt distant avec la commande: `git remote add`
- Dans mon cas:
- `git remote add origin https://gitlab.com/Teacher01/learning-template.git`
- Origin représente par défaut le nom du dépôt distant. Vous pouvez le remplacer.
- Ensuite, vous devez transférer votre dossier de travail et votre historique avec: `git push -u origin master`

Cloner un dépôt existant

- Cloner un dépôt existant consiste à récupérer tout l'historique et tous les codes source d'un projet avec Git.
- Vous devez posséder l'url du dépôt et les accès s'il s'agit d'un dépôt privé.
- Pour cloner le projet: `git clone`
<https://gitlab.com/Teacher01/learning-template.git>
- Un dossier « learning-template » est créé et tous les fichiers source du projet ainsi que l'historique de chacune de leurs modifications sont ajoutés.

Les dépôts Symfony

- En raison du nombre de fichiers et du poids trop élevé d'un projet Symfony, le dossier « vendor » ne fait jamais partie d'un dépôt.
- Pour réinstaller le projet complet vous devez lancer la commande: `composer install`
- Dans votre dépôt se trouve un fichier « `composer.json` » qui contient la liste de tous les services et dépendances du projet. Composer se charge de tout réinstaller.



Advanced Symfony



Security

La gestion des rôles

- Jusqu'à présent, nous utilisons un seul rôle par défaut « ROLE_USER » qui était renvoyé par défaut via la méthode `getRoles()` de l'entité `User`.
- Si nous souhaitons gérer d'autres rôles, nous devons ajouter un attribut « role » de type json dans l'entité et le typé comme un array.

```
/**  
 * @ORM\Column(type="json")  
 */  
private $role = [];
```

La hiérarchie des rôles

- Pour faciliter les autorisations, vous devez définir un système hiérarchique des rôles. Par exemple le rôle admin héritant du rôle user.
- Vous devez modifier le fichier `security.yaml`

```
role_hierarchy:  
  ROLE_ADMIN:      ROLE_USER  
  ROLE_SUPER_ADMIN: [ROLE_ADMIN]
```

- Un tableau permet de spécifier plusieurs rôles

La méthode getUsername()

- Si vous ne disposez pas d'un attribut userName dans votre entité assurez vous d'avoir la méthode getUsername() qui renvoi l'attribut unique servant pour l'authentification.

```
public function getUsername(): string
{
    return (string) $this->email;
}
```


La classe authenticationUtils

- Il s'agit d'une classe d'utilitaires permettant la gestion des erreurs dans l'authentification. Elle sera implémentée dans la méthode login() du controller.
- Deux méthodes vont être utiles:
 - `getLastAuthenticationError()`: retourne la dernière erreur survenue (bad credentials)
 - `getLastUsername()`: permet de récupérer le nom de l'utilisateur (_username)
- `use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;`

Méthode login()

```
public function login(AuthenticationUtils $utils)
{
    // get the login error if there is one
    $error = $utils->getLastAuthenticationError();

    // last username entered by the user
    $lastUsername = $utils->getLastUsername();

    return $this->render('account/login.html.twig', [
        'last_username' => $lastUsername,
        'error'         => $error !== null, // si $erreur différent de null
    ]);
}
```

La vue: login.html.twig

```
<div class="container" style="min-height: 70vh;">
  <h2 class="my-4">Identification</h2>
  {% if error %}
    <div class="alert alert-danger">Nom d'utilisateur ou mot de passe erronés</div>
  {% endif %}
  <form action="{{ path('login') }}" method="post">
    <div class="form-group">
      <label for="mail">Your EMail</label>
      <input type="email" class="form-control" required name="_username"
placeholder="Mail" id="mail" value="{{ last_username }}">
    </div>
    <div class="form-group">
      <label for="pass">Your Password</label>
      <input type="password" class="form-control" required name="_password"
placeholder="Password" id="pass">
    </div>
    <div class="form-group">
      <button type="submit" class="btn-primary">Connexion</button>
    </div>
  </form>
</div>
```

Protection CSRF

- Le formulaire de login que vous avez créé manuellement n'est pas protégé contre la faille CSRF (cross-site request forgery) contrairement aux autres formulaires de Symfony.
- Pour se protéger, vous devez activer le `security.csrf.token_manager` dans `security.yaml`. Celui-ci permettra de générer un token (jeton) dans le formulaire qui sera ensuite comparé par le composant Security de Symfony

```
form_login:  
    login_path: login  
    check_path: login  
    csrf_token_generator: security.csrf.token_manager
```

Le formulaire de login

- Ajoutez un champ caché dont le name doit obligatoirement être `_csrf_token` avec la valeur `csrf_token('authenticate')`

```
<div class="form-group">
  <input type="hidden" name="_csrf_token"
value="{{ csrf_token('authenticate') }}">
  <button type="submit" class="btn-
primary">Connexion</button>
</div>
```

- Vérifiez la présence du token dans le formulaire avec un view source. Si vous le modifiez, la connexion est refusée !

Modifier le Password

- Nous allons proposer à l'utilisateur de modifier son password via un formulaire supplémentaire.
- **Etapas:**
 - Création d'une entité (pas de migration) pour bénéficier des règles de validation.
 - Création du FormType (**PasswordUpdateType**)
 - Ajout d'une méthode (**accountPassword**) dans le Controller AccountController
 - Création d'un formulaire en twig (**account_password.html.twig**)

L'entité PasswordUpdate

- Créez une nouvelle classe manuellement dans le dossier entity. Cette classe ne devra pas être migrée vers la DB !!!
- Les attributs:
 - `private $oldPassword;`
 - `private $newPassword;`
 - `private $confirmPassword;`
- Les getters et les setters
- Ajoutez les contraintes (Length et EquatTo)

La classe PasswordUpdateType

- Créez le formType mais sans l'associer à une entité. La classe que vous venez de créer n'est pas une Entité !!! Mais une simple classe PHP.
- Ajoutez les trois champs de type PasswordType

La méthode accountPassword du contrôleur

```
public function accountPassword()
{
    $passwordUpdate = new PasswordUpdate();
    $form = $this->createForm(PasswordUpdateType::class, $passwordUpdate);
    return $this->render('account/account_password.html.twig',
        [
            'form' => $form->createView()
        ]
    );
}
```

▀ Sauvegarder le nouveau mot de passe



Symfony bundles

Sources

- De nombreux bundles peuvent être installés sous Symfony. Faites cependant attention à leur compatibilité et leur provenance. Certains sont des librairies officielles et d'autres sont des contributions.
- Sources:
- <https://packagist.org/>
- <https://flex.symfony.com/>



Générer des slugs

Slugify



Slugify

- Dépôt officiel: <https://packagist.org/packages/cocur/slugify>
- Installation: `composer require cocur/slugify`
- Utilisez un générateur de slug pour obtenir des url plus cohérentes notamment pour les articles. Actuellement pour accéder à un article particulier, l'url se termine par article-num. Pour obtenir plus de sens vous souhaiteriez avoir le titre de l'article. Pour ce faire, ce titre doit être nettoyé (sanitize) pour correspondre aux normes des url (pas d'espace, maj, accents et caractères spéciaux).
- Il faut disposer dans la table des articles d'un champ appelé slug (string – taille du titre – not null).

Slugify pour les fixtures

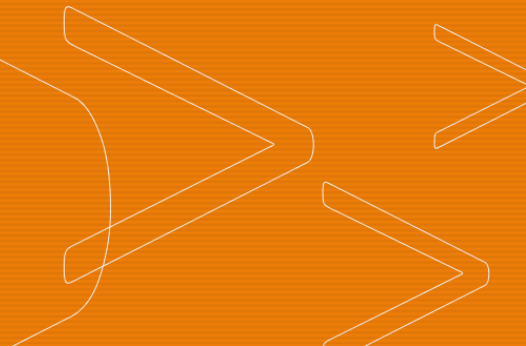
```
use Cocur\Slugify\Slugify;
```

```
$slugify = new Slugify();
```

```
$course->setSlug($slugify->slugify($course->getName()));
```

VichUploader

Gestion des images



Installation

- Rechercher sur: <https://flex.symfony.com> (symfony Recipes Server)
- Package details et repository pour obtenir toute la documentation nécessaire.
- Commande: `composer require vich/uploader-bundle`. Répondez **Yes** à la question « Do you want to execute this recipe? »
- Configurations automatiques:
 - Lors de l'installation Symfony Flex s'est chargé de configurer le fichier `config/bundles.php` (Enable the bundle).
 - Le driver ORM pour la base de données a également été configuré dans le fichier: `config/packages/vich_uploader.yaml`

Configuration

- Configurations manuelles:
- Le mapping (les données de l'image: nom et positionnement). Dans le fichier `vich_uploader.yaml` ajoutez les propriétés suivantes. Un exemple est fourni dans les commentaires.

```
mappings:  
  user_image:  
    uri_prefix: /img/avatar  
    upload_destination:  
      '%kernel.project_dir%/public/img/avatar'
```

- Les modifications dans l'entité concernée (adverts).
 - Importation du namespace:

```
use Vich\UploaderBundle\Mapping\Annotation as Vich;
```

Configuration

- Ajout du use et de l'annotation dans l'entité:

```
use Vich\UploaderBundle\Mapping\Annotation as Vich;  
  
@Vich\Uploadable
```

- Création d'une nouvelle propriété (\$imageFile) qui ne sera pas persistée en base de données:

```
/**  
 * @Vich\UploadableField(mapping="user_image", fileNameProperty="image", size="imageSize")  
 *  
 * @var File|null  
 */  
private $imageFile;
```

- Importez la classe pour le type File: `use Symfony\Component\HttpFoundation\File\File;`
- Création des mutateurs en fluent (getter et setter): add Getter and Setter avec PhpStorm

Modifier la méthode setImageFile()

- Lors de l'update, pour des raisons de persistance de l'objet, vous devez modifier la méthode setImageFile().
- Assure-vous d'avoir dans votre entité une propriété updatedAt qui permettra à Vich de l'actualiser pour obtenir une persistance.

```
public function setImageFile(?File $imageFile): void
{
    $this->imageFile = $imageFile;
    if ($this->imageFile instanceof UploadedFile) {
        $this->updatedAt = new \DateTime('now');
    }
}
```

Modifier le formType

- Le champ d'upload doit correspondre à l'attribut imageFile de l'entité et non pas à l'attribut image qui enregistre le nom de l'image. Ici l'upload n'est pas obligatoire.

```
->add('imageFile', FileType::class, [  
    'label' => 'Image:',  
    'required' => false,  
    'attr' => ['placeholder' => 'Photo']  
])
```

Modifier le Controller

- Comme l'upload n'est pas obligatoire mais que l'attribut image ne peut être null, vous devez proposer un fichier image par défaut.

```
if(empty($user->getImageFile())) $user->setImage('default.jpg');
```