Gestion de projets avec Git

Vincent Dubois

Introduction

Objectifs de ce cours

Pour un projet reposant sur un ensemble de fichiers (par exemple, du code, des images, des documents de travail etc …), il est inévitable de se retrouver avec un certain nombres de copies (partielles ou totales, modifiées à des degrés divers) d'un même ensemble de fichiers. Une des difficultés dans la gestion du projet est de manipuler ces copies avec suffisamment de rigueur pour ne pas perdre d'informations. Cela peut être réalisé manuellement, en créant des archives facilement identifiables (idéalement, en précisant au moins pour chaque version la date et l'auteur.).

La gestion manuelle d'un tel ensemble d'archives est très lourd, même pour des opérations relativement simples, comme le partage de la dernière version, l'intégration du travail des membres d'un groupe, la recherche d'un fichier d'une des versions précédentes etc..

L'objectif de ce cours est d'apprendre à utiliser un outil dédié à la gestion des fichiers d'un projet, et qui permet de simplifier toutes ces tâches.

Présentation générale de Git

Git est un système distribué de gestion de versions. Un système de gestion de versions est un outil pour gérer les versions d'un projet, c'est à dire les archives des différents fichiers. On dit que le système est distribué (par opposition à un système centralisé) quand chaque dépôt local (c'est à dire chaque utilisateur) a l'intégralité des archives disponibles (localement, donc). Un tel système permet aussi de synchroniser les différents dépôts, c'est à dire de partager les différentes versions, et de combiner les modifications de plusieurs utilisateurs (ce qui peut être fait automatiquement, dans une certaine mesure).

La bonne utilisation de Git vous permettra entre autre :

- D'assurer la sauvegarde des données de votre projet (sous forme d'archives, que l'on peut explorer selon de nombreux critères comme la date ou les fichiers modifiés).
- D'éviter la perte ou l'écrasement de fichiers : chaque utilisateur a une copie de toutes les archives, ce qui rend très improbable la perte totale des données (il faudrait que chaque membre du groupe perde ses données …). En pratique, seules les données non partagées sont vulnérables.
- De partager les modifications apportées par chaque membre au projet. Les méthodes de synchronisation sont nombreuses (principalement réseau : serveur spécialisé, http et ssh

entre autre, mais aussi par clef usb ou échange de fichiers sur une même machine).
- D'intégrer, souvent de manière automatique, les différentes modifications. Autrement dit, à partir de plusieurs versions d'un même projet, il est possible de produire une version combinant les différentes modifications.
- De faciliter le déploiement du projet, en mettant facilement à jour les fichiers sur la machine de test (dans le cas d'un site web, par exemple).
- De consulter facilement les archives du projet, par exemple pour localiser un problème, ou pour trouver l'auteur d'une modification.

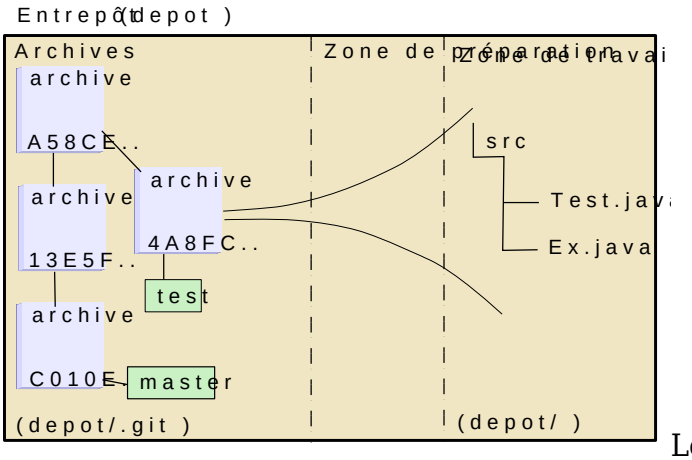
En revanche, l'utilisation de Git ne résout pas en soi les problèmes liés à l'organisation d'un groupe autour d'un même projet ; comme tout outil de gestion de projet, il rend juste plus facile les tâches nécessaires au bon avancement du projet. Git doit donc être intégré dans vos processus de travail, il ne peut en aucun cas se substituer à une bonne organisation du projet (distribution des rôles et des tâches par exemple).

Les archives

L'élément de base géré par git est l'archive, qui représente l'ensemble des fichiers du projet à un moment donné, sur une machine donnée. On parle aussi d'instantanné (au sens photographique du terme), ou en anglais de snapshot, ou encore simplement de version du projet. Le terme commit (du nom de la commande qui sert à créer les archives) est aussi utilisé. Ce sont les archives qui sont sauvegardées dans les dépôts, échangées, fusionnées etc.. Chaque archive se comporte comme une version compressée (même si la réalité est plus complexe – et plus efficace) du répertoire principal du projet, celui contenant tous les fichiers et répertoires. Chaque archive comporte l'intégralité des fichiers du projet, ainsi que les informations caractéristiques de la version (texte descriptif, version(s) précédente(s), date , auteur ..), ainsi qu'un identifiant unique, sous forme d'un code en hexadécimal calculé à partir de l'ensemble des informations de l'archive. Ce code permet à la fois de désigner l'archive sans ambiguïté et de garantir son intégrité (protection contre la corruption de données, et contre les modifications frauduleuses : toute modification du contenu de l'archive modifie le code associé, celui étant calculé automatiquement à partir des données.)

Une fois créée, une archive ne peut plus être altérée. Ajouter la version actuelle des fichiers en tant que nouvelle archive ne modifie donc en rien les archives existantes. Il n'y a aucun risque de perte de données en archivant (faites-le souvent !).

Les entrepôts



Les archives sont regroupées dans des entrepôts (repository en anglais). Chaque membre du projet travaille sur (au moins) un de ces dépôts, local à sa machine. La plupart des tâches peuvent être effectuées sur ce dépôt local, sans nécessiter aucun accès aux autres dépôts (le dépôt local contient tout l'historique du projet). Internet n'est donc pas indispensable pour travailler avec Git. Même pour partager les informations, il est possible de s'en passer (clefs usb, connexion directe d'ordinateurs à ordinateur etc…)

Chaque entrepôt occupe un répertoire (par exemple depot), qui lui-même comporte plusieurs zones. Le répertoire depot/.git contient les archives, la configuration et les autres informations liées au fonctionnement de Git. Le répertoire depot/ abrite la zone de travail, c'est à dire les répertoires et fichiers du projets sur lesquels le travail se porte à un moment donné. Ces fichiers sont disponibles, utilisables et modifiables normalement et sans restriction ni outil particulier. De plus, il est possible d'amener tout ou partie de n'importe quelle archive dans ce répertoire.

Les branches

Pour chaque nouvelle archive ajoutée au dépôt, git note l'archive dont elle est issue (autrement dit, la version précédente). La dernière version en date d'une archive ainsi que les versions précédentes constituent ce que l'on appelle une branche. Par défaut, la branche contenant la version actuelle d'un projet s'appelle la branche maître (master). Il est possible (et recommandé) de définir de nouvelles branches pour le travail de chacun, chaque branche correspondant normalement à une des tache à réaliser. Une fois le travail réalisé et le résultat vérifié, il est possible d'intégrer les améliorations d'une branche à la branche maître. Si plusieurs branches ont été développées en parallèle, il faut alors prendre en compte les modifications apportées sur chacune. On appelle cette opération la fusion (merge) des branches.

Le partage d'archives entre entrepôts.

Git est conçu et optimisé pour le partage des archives entre entrepôts. De nombreuses options sont disponibles, en utilisant les protocoles réseaux standards (ssh, scp, ftp, http, mail …), un serveur spécifique, ou simplement le partage direct des informations sur une même machine (y compris en utilisant une clef usb). Les opérations de partage de versions permettent d'envoyer ou de recevoir les archives qui n'existent que dans un des deux cotés de l'échange. Ces échanges ne peuvent qu'ajouter à un entrepôt des archives qui ne sont pas déjà présentes. Il n'y a aucun risque d'altérer les archives d'une dépôt (on est donc assuré de ne pas perdre de données lors de la

synchronisation des dépôt !).

il est parfaitement possible de définir un dépôt central, mais cela n'est en rien obligatoire : il est possible de travailler chacun sur son dépôt, mais cela impose de procéder à des mises à jour pour chaque dépôt. En pratique, un dépôt centralisé est recommandé. Cela peut être le dépôt d'une personne du projet jouant le rôle d'intégrateur, ou alors un dépôt extérieur facilement accessible (pour les projets open-source, il existe des services de dépôt git gratuits (par exemple GitHub). Si vous disposer d'une machine accessible sur internet, elle peut assez facilement servir de dépôt, si elle dispose d'un des protocoles supportés par git).

Les commandes

Les commandes de git permettent principalement :

- d'enregistrer la zone de travail dans une archive, en passant par une étape de préparation (staging), ou au contraire d'ouvrir une archive particulière dans la zone de travail
- de gérer les archives elles-mêmes, à l'intérieur d'un même dépôt (par exemple en les étiquetant, ou en combinant les modifications réparties dans plusieurs archives)
- de manipuler les dépôts, par exemple en les copiant, en échangeant des archives etc..

Vous trouverez le détails des principales commandes nécessaires à toutes ces tâches dans l'aide-mémoire. Mais l'utilisation d'un système de gestion de version ne se résume pas à une maîtrise technique de l'outil, c'est l'intégration de l'outil dans la méthode de travail du groupe qui permet de bénéficier pleinement des avantages de celui-ci. Les différentes étapes nécessaires à un usage efficace de Git dans le cadre de votre projet sont détaillées dans les prochains chapitres.

Méthodes de travail

Introduction

Le développement en groupe d'un projet commun s'appuie sur une bonne organisation et de bonnes méthodes de travail. Ces éléments sont nécessaires pour être efficacite, c'est à dire pour obtenir les meilleurs résultats possibles pour une quantité de travail donnée. Avec une mauvaise organisation ou coordination dans un groupe, le résultat produit peut ne pas être meilleur que le travail d'un seul individu, et même dans certains cas être pire (par exemple en cas de pertes de données). Même un travail individuel peut bénéficier d'une bonne méthode, en particulier en ce qui concerne la gestion des versions des différents fichiers d'un gros projet.

Le fonctionnement recommandé est le suivant :

- réunions de travail, permettant de répartir les tâches, de discuter de l'avancement des travaux, de faire les choix à sur l'orientation du projet, de distribuer les nouvelles versions.
- travail individuel, sur les aspects déterminés en réunion.

Pour les gros groupes, il peut être nécessaire de désigner un membre chargé de réaliser l'intégration des travaux individuels, afin de réduire le temps nécessaire pour les réunions de travail.

Ce fonctionnement est basé entre autre sur le fait que travailler à plusieurs sur une tâche qui pourrait être réalisée par une seule personne revient à diviser l'efficacité du groupe par le nombre de personnes mobilisées sur la tâche. Ainsi, en travaillant à 3 sur une même machine pendant 2 heures, on produit en gros 2h de travail au lieu de 2x3h = 6h, ce qui entraîne un manque à gagner de 4h de travail (en 2h !). Il faut donc réserver les réunions à ce qui doit être fait en groupe, comme la répartition du travail, le choix des objectifs et l'échange d'idées et de remarques. C'est l'utilisation de Git dans le cadre d'une organisation de ce type qui est expliquée dans ce document.

Travail sur des taches différentes

Une méthode de travail très courante pour les projets est de répartir le travail entre les membres du groupe. Pour optimiser l'efficacité dans ce cas, il est important que les tâches soient le plus indépendantes possibles. Par exemple :

- Un membre travaille sur le style d'un site, l'autre sur le contenu.
- Chacun travaille sur une partie différente du contenu.

Cela nécessite d'affecter le travail au préalable (par exemple en réunion, ou sur décision du chef de projet, ou par décision collégiale), en vérifiant que les domaines de responsabilité se chevauchent le moins possible. Ensuite, chacun peut travailler indépendamment, en intégrant régulièrement son travail (il y a peu de risques de conflits). Il est recommandé de faire des réunions régulièrement pour que le groupe puisse vérifier l'avancement et la cohérence du travail réalisé. Il est aussi possible de réaffecter les missions et responsabilités en fonction de l'avancement (attention tout de même, ajouter quelqu'un en plus sur une tache ne garanti pas un avancement plus rapide, au contraire…)

Travail exploratoire

Quand le travail à réaliser demande une certaine créativité, il peut être intéressant de produire plusieurs prototypes. Par exemple :

- Construction d’une charte graphique ou du style d’une page.
- Choix de la structure d’un site.

Dans cette situation, l’utilisation des branches est particulièrement bien adaptée. Chaque branche est développée en parallèle par une partie du groupe. Quand les prototypes sont prêts, il convient de déterminer celui (ou ceux) qui seront intégrés (fusionnés) à la version officielle. De nouveaux développements peuvent ensuite être réalisés, en partant de la nouvelle version.

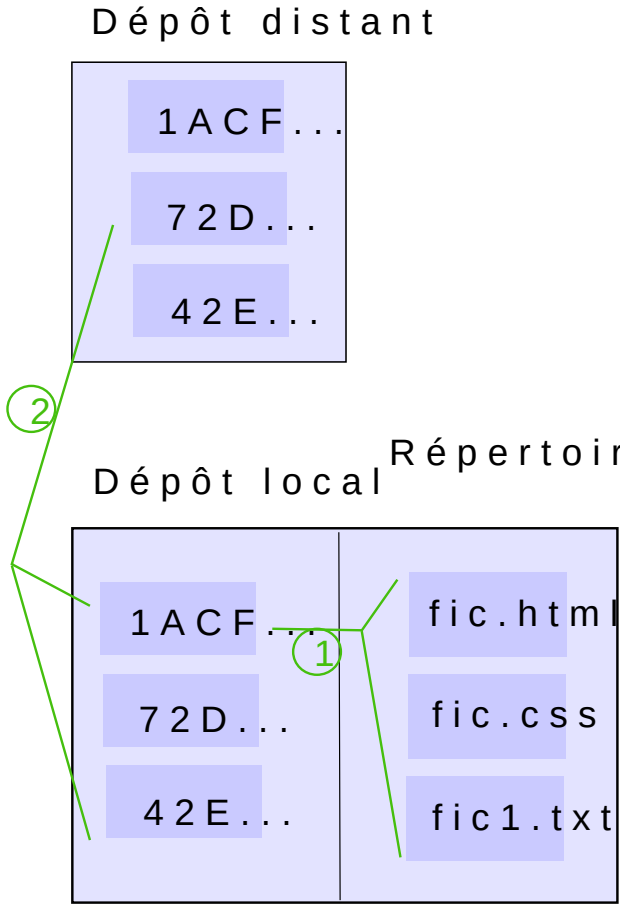
Session de travail type

Phase du travail	**Etapes**	**Commandes git**	**Disponible hors ligne ?**
Préparation du travail	Obtenir un entrepôt à jour		
	S’il n’y en a pas, le créer	init	oui
	S’il y en a un, mais pas localement	clone	non
	S’il y en un local à mettre à jour	fetch,merge / pull	non
	(Créer une branche pour le travail à réaliser)	branch / checkout	oui
Travail sur le projet (à répéter autant de fois que nécessaire)	Créer et/ou éditer les fichier		oui
	Corriger les problèmes jusqu’à ce que le projet tourne.		oui
	Sauvegarder la version qui tourne dans le dépôt	add / commit	oui
Synchronisation du travail avec celui de l’équipe	Mettre à jour l’entrepôt local (si en ligne)	fetch, merge / pull	non
	Fusionner la branche créée avec la branche principale (master)	checkout / merge	oui (sans intérêt sans le fetch)
	Régler les conflits de version	add/commit	Id.
	Vérifier que le projet fusionné tourne. Sinon, retour à « travail sur le projet » pour corriger les problèmes.		Id.
	Exporter les changements.	push / bundle / format-patch	oui (sauf le push)

Les Dépôts

Il existe beaucoup d'organisations possibles du travail avec Git, avec des configurations des liens entre dépôts différentes à chaque fois: les dépôts peuvent être centralisés, hiérarchisés, et même communiquer de manière transversale. L'organisation devrait idéalement être déterminée en fonction des contraintes techniques, du travail à réaliser et des préférences de chacun.

Principes généraux



Chacun dispose au minimum d'un dépôt local, associé à son répertoire de travail, et d'un dépôt distant, accessible par le réseau à tous les membres du projet. La sauvegarde sur dépôt distant se fait en plusieurs étapes :

1. Sauvegarde du répertoire de travail dans une archive du dépôt local :

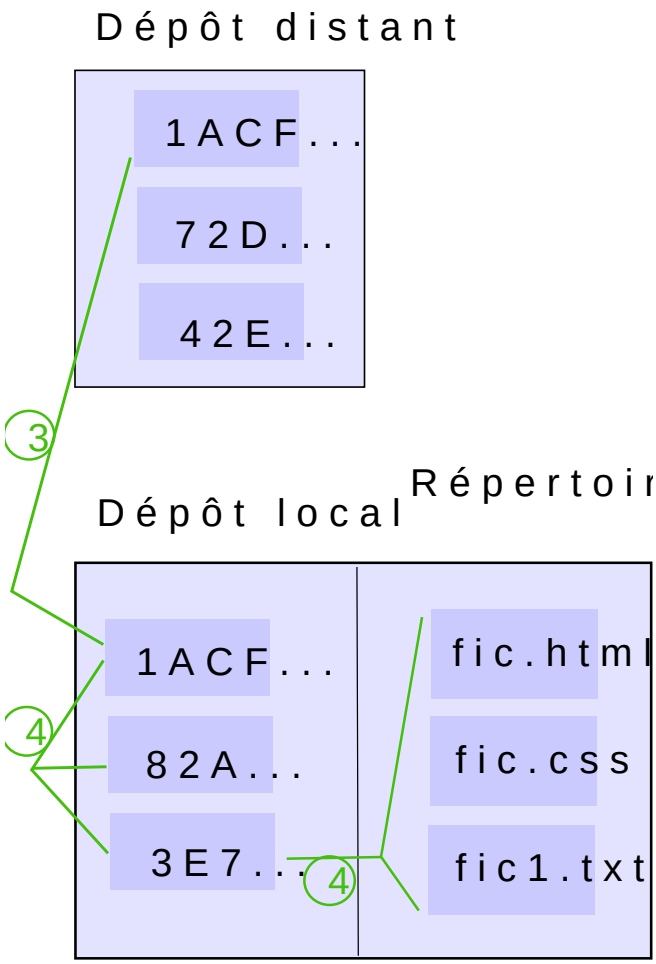
```
git add nouveaux_fichiers

git commit -a -m 'Détail des dernières modifications'
```

1. Copie des nouvelles archives locales vers le dépôt distant :

```
git push –all
```

L'étape 1 doit être effectuée très souvent. Il est plus facile de documenter les modifications au fur et a mesure, plutôt que de devoir se rappeler après coup de tout ce que l'on a fait. Il est donc préférable de sauvegarder dans le dépôt local à chaque fois que l'on achève une modification donnée. L'étape 2 doit être réalisée à chaque fois que l'on termine une session de travail sur une machine, pour rendre les données accessible.

Inversement, on peut récupérer et intégrer les modifications du dépôt distant dans le dépôt local :



1. Copie des archives du dépôts distant vers le dépôt local :

```
git fetch origin master
```

1. Fusion de la dernière version distante et de la version locale :

```
git merge origin/master
```

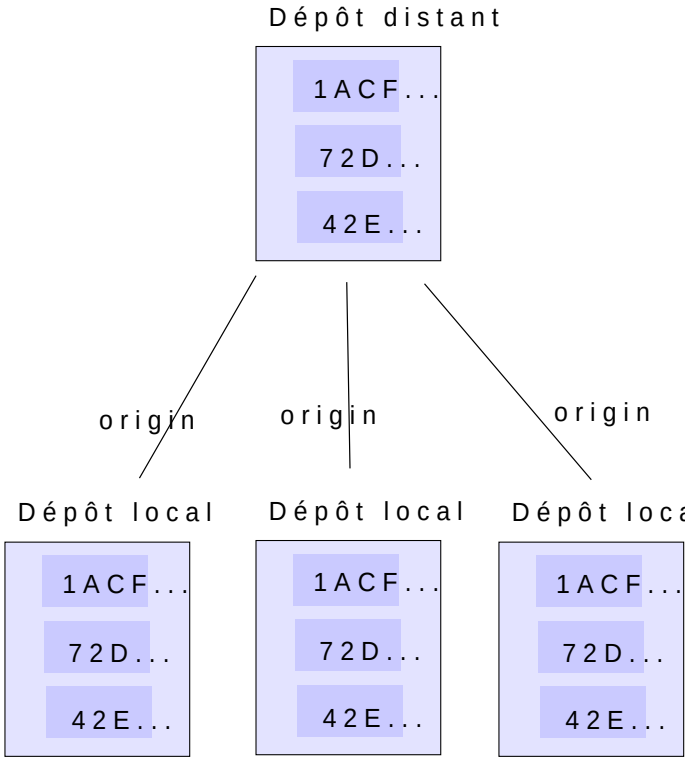
Remarques :

- Les commandes 3 et 4 peuvent être faites en une seule ligne avec la commande pull :

```
git pull origin master
```

- Les fichiers du répertoire de travail doivent avoir été sauvegardés (commande 1) avant de tenter la commande 3.
- Si vous êtes seul à utiliser le dépôt distant, la fusion sera toujours triviale (il suffit de prendre la version la plus récente). Dans la contraire, il peut se produire des conflits entre les versions
- Il est possible de disposer de plusieurs postes de travail pour une même personne, avec un dépôt local et un répertoire de travail à chaque fois. Pourvu que le travail soit bien sauvegardé et bien chargé (en utilisant les commandes décrites précédemment), le travail sera disponible dans sa dernière version sur tous les postes.

Dépôt distant unique



Pour un petit groupe (ou une personne seule sur un projet), on peut s'organiser avec un seul dépôt distant pour tout le monde. Ce dépôt contient la version de référence du projet.

Le dépôt central est créé sans répertoire de travail (`git init –bare`), les autres dépôt sont obtenus en clonant ce dépôt (`git clone url`). Le fait de cloner le dépôt permet d'obtenir une copie de celui-ci, et configure automatiquement le dépôt d'origine comme dépôt par défaut (ce dépôt distant est alors nommé `origin`).

Dans ce mode de fonctionnement, chaque utilisateur doit commencer sa session de travail en récupérant et intégrant la dernière version de référence disponible (étapes 3 et 4 ci-dessus).

Quand le travail est terminé, le résultat doit être mis à disposition sur le dépôt distant (étapes 1 et 2 ci-dessus). Cette architecture est celle retenue pour le TP1.

Remarque : A moins de souscrire un compte payant, le site d'hébergement de dépôt Github ne permet pas de travailler à plusieurs sur un même dépôt.

Lorsqu'une version est envoyée vers le dépôt distant (étape 2) par A, il se peut que le dépôt ait évolué depuis la dernière fois où il a été consulté, B ayant par exemple pu ajouter ses modifications. Afin d'éviter de perdre les modifications de B en les remplaçant par celles de A, Git refuse dans ce cas de modifier le dépôt distant, et invite A à prendre en compte les modifications de B avec la commande `git pull` (voir étapes 3 et 4 ci-dessus) avant de re-soumettre sa version. Si les modifications concernent des fichiers différents, Git conserve simplement les dernières versions de chacun. Sinon, ont dit qu'il y a un conflit.

Gestion des conflits

Un conflit résulte donc de la modification simultanée d'un même fichier par deux utilisateurs A et B. Il faut alors décider quoi faire des deux versions. Sans système de gestion de version, il est nécessaire comparer à la main les fichiers, décider lequel garder, ou éditer un des deux fichiers pour intégrer les modifications de A et B.

Avec Git, c'est le logiciel qui analyse ligne par ligne les différences entre les fichiers. Cette analyse permet de produire un fichier contenant les lignes communes aux deux versions, et les deux versions (avec leur origine) pour les parties conflictuelles.

Pour chacun des fichiers comportant des conflits, il est alors possible de choisir comment les régler :

1. éditer le fichier pour régler manuellement les conflits. Ces derniers sont signalés par des chevrons (<<<<< ===== >>>>>), qui encadrent les différentes versions.
2. Revenir à la version avant fusion avec la commande

`git checkout —-ours` **`nom_fichier`**.

1. Accepter la version distante à la place de la version locale :

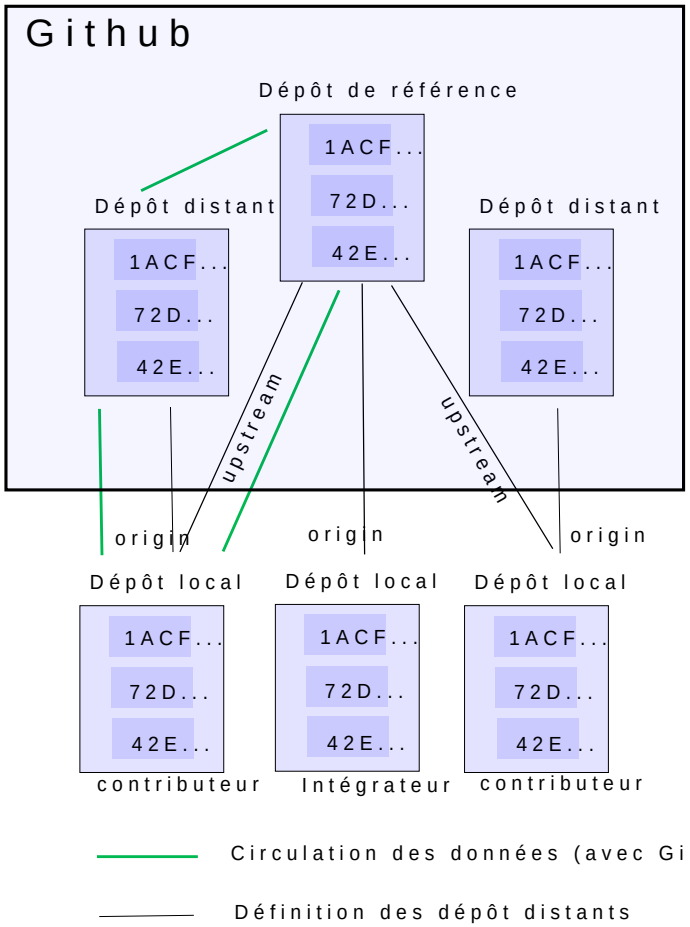
`git checkout —-theirs` **`nom_fichier`**.

En cas d'erreur, on peut revenir à la version avec marqueurs avec

`git checkout -m` **`nom_fichier`**.

Une fois le conflit réglé, il faut utiliser `git add` sur le fichier. Quand tous les conflits sont résolus et validés par `git add`, `git commit` permet de sauver la version fusionnée.

Dépôts distants individuels



Pour les projets d'une certaine taille, il n'est pas envisageable de travailler sur un seul dépôt distant. Cela occasionnerait de nombreux conflits de versions, et impliquerait de donner l'accès à ce dépôt à tous les contributeurs (avec le risque d'erreurs ou d'actions malveillantes que cela entraîne).

L'alternative généralement retenue, en particulier dans le domaine du logiciel libre, c'est de disposer pour chaque membre d'un dépôt distant. Ce dépôt distant est accessible publiquement en lecture (au moins à l'échelle du projet), mais ne peut être modifié que par son propriétaire. Un des dépôts sert de dépôt de référence (pour la dernière version du projet), et son propriétaire est appelé intégrateur. Quand une modification doit être ajoutée au projet, le contributeur prévient l'intégrateur. L'intégrateur récupère alors la nouvelle version directement sur dépôt du contributeur (spécifié par son url), avec la commande

`git pull` **`url`**.

Comme dans le cas du dépôt distant unique, la commande `git pull` ne fonctionnera que si la nouvelle version est bien basée sur la dernière version (autrement dit, si il n'y a pas eu de modifications du dépôt de référence depuis la dernière fois où le contributeur a fait sa mise à jour).

A part l'intégrateur, chacun utilise normalement deux dépôts distants :

1. son dépôt individuel public, nommé par défaut origin
2. le dépôt public de l'intégrateur, nommé upstream

Le lien vers origin est configuré par défaut lorsque le dépôt individuel est cloné, en revanche il est nécessaire d'ajouter le lien vers upstream manuellement avec la commande suivante :

`git remote add upstream` **`url`**

Pour garder la version locale à jour, il est important d'intégrer régulièrement la version de référence. Cela se fait avec la commande suivante :

```
git pull upstream master
```

(cette commande suppose que répertoire de travail soit propre, c'est à dire qu'il n'ait pas été modifié depuis le dernier `git commit`)

Ce mode de fonctionnement est le celui retenu dans le TP2, et s'intègre naturellement avec l'hébergement gratuit de dépôt sur Github : seul le propriétaire peut modifier un dépôt Github, mais tout le monde peut le consulter.

TD 1 (Découverte)

L'objectif de ce TP est d'utiliser les fonctions de base de Git pour la création et la gestion d'une page web individuelle.

Création du dépôt local.

Un dépôt commun a déjà été créé pour votre groupe dans un répertoire partagé, mais allez chacun devoir créer un clone de ce dépôt pour travailler. Votre travail se fera donc en local, dans votre copie du dépôt, et vos modifications seront plus tard (au prochain TP) partagées via le dépôt commun. Le partage se fera par les commandes correspondantes de git, vous ne toucherez donc jamais directement aux fichiers du dépôt commun. Pour travailler localement, il faudra créer un répertoire git, se placer dedans, et cloner (le terme technique pour copeir un dépôt git) le dépôt commun.

Cela se fait avec les commandes suivantes :

```
mkdir git
cd git
git clone chemin
```

Il faut bien sur remplacer **chemin** par le chemin de votre dépôt, c'est à dire `/export/commun/mmi/git/2018/groupe.git` (**groupe** correspond à votre groupe TP, par exemple mmi1a1).
Cette commande crée une copie locale du dépôt. C'est sur cette copie que nous allons travailler (n'essayez pas de travailler directement dans le répertoire partagé, ce n'est pas une bonne idée… A votre avis pourquoi ?).

1. Quels sont les fichiers et/ou répertoires créés par `git clone` ? (`ls -al` permet de lister aussi les fichiers cachés).
2. Quels sont les principaux protocoles supportés par `git clone` ? (le manuel est accessible par la commante `git help clone`)
3. Utilisez un navigateur pour ouvrir les fichiers html.

Configuration générale

Adaptez les commandes suivantes pour configurer votre installation de git (en modifiant évidemment les parties en gras):

```
git config --global user.name "Votre nom"
git config --global user.mail login@machine.net
git config --global core.editor nano
```

L'effet de ces commandes est permanent, vous n'aurez plus besoin de les taper (à moins de travailler sur un compte différent — par exemple chez vous).

Travail local.

1. Modifiez maintenant le fichier mon_nom.html (sans le renommer pour l'instant) en mettant le contenu à votre nom (il y a exactement deux lignes à modifier).
   Qu'indique maintenant la commande `git status` ?Pour voir le détail des dernières modifications depuis le dernier commit (c'est à dire la dernière sauvegarde dans votre dépôt), utilisez la commande
   `git diff`.Pour archiver les modifications dans votre entrepôt (c'est ce que l'on appelle un commit), utilisez les commandes suivantes :

```
git add *.html
```

```
git commit -m « Message expliquant ce qui a été modifié.«
```

La commande `git add` indique les fichiers à prendre en compte pour la prochaine sauvegarde. La commande `git commit` réalise l'archivage des fichiers sauvegardés dans répertoire de travail vers le dépôt local. L'option -m permet de préciser le message explicatif dans la ligne de commande, au lieu d'ouvrir un éditeur de texte (comportement par défaut). Remarque : un commit sans message n'est jamais valide. A votre avis, pourquoi ? (pensez à ce qui peut arriver sur un gros projet, comme le noyau Linux…)

Il est recommandé d'utiliser ces deux commandes (git add / commit) très souvent : seules les versions ayant fait l'objet d'un commit sont conservées. On doit toujours faire un commit dans les cas suivants :

- Après avoir fini et/ou corrigé quelque-chose (une nouvelle page, un paragraphe de texte, une feuille de style, une image de fond etc…)
- Avant de faire quelque-chose de « risqué » (tout ce qui pourrait « casser » le site
- Avant de fermer une session ou d'arrêter de travailler sur quelque chose.

Attention, on ne peut pas faire deux commits d'affilée sans avoir rien modifier entre deux (quelle commande permet de vérifier si c'est le cas ou non ?).
Si vous avez beaucoup de choses à mettre dans votre message de commit, il aurait probablement fallu en faire un plus tôt.

1. Renommez maintenant votre fichier mon_nom.html en le mettant à votre nom (pensez aussi à mettre à jour le lien dans le fichier index.html).Pour vérifier comment git voit vos dernières modifications (les modifications depuis le dernier commit), utilisez la commande :

```
git status
```

Pour indiquer que le nouveau fichier doit être inclus dans le commit, utilisez `git add`. Pour faire le commit, utilisez l'option -a :

```
git add fichier
```

```
git commit -a
```

A quoi correspond l'option -a ? (pensez à utiliser le manuel)

1. Vérifiez maintenant que la page est toujours correcte dans le navigateur, et ajoutez quelques informations sur vous et/ou une présentation dans votre fichier html. N'oubliez pas de faire régulièrement des add/commit ! (à chaque fois que cela est adapté)

2. Que produit la commande `git log` ?Cette commande accepte aussi l'option `—-oneline`, essayez la. Il est possible d'afficher ces informations (et beaucoup d'autres !) avec des outils externes, comme `gitk.`

Utilisation du fichier .gitignore

Certains fichiers n'ont pas leur place dans le dépôt, comme par exemple les fichiers de sauvegarde des éditeurs de texte. Pour éviter que ces fichiers soient considérés comme des fichiers à ajouter par git, on utilise un fichier d'exclusion : .gitignore

Consultez ce fichier et ajoutez si nécessaire des fichiers à ignorer.

Récupérer une version précédente d'un fichier

Tous les fichiers enregistrés à un moment donné dans un dépôt git peuvent être récupérés. La principale commande pour cela est `git checkout`.
Attention : avant d'utiliser une commande suivantes, vérifiez que vous avez bien fait un commit sur votre dernière version.

Pour récupérer une version précédente d'un fichier (et donc remplacer la version actuelle), utilisez une des commandes suivantes :

```
git checkout monfichier # si monfichier n'a pas encore été ajouté au prochain
 commit par git add
git checkout HEAD monfichier # sinon
git checkout 1234ABCDE monfichier # pour une version en particulier
git checkout master # pour revenir à la dernière version (pour tous les fichi
ers)
```

Essayer par exemple de récupérer la version originale de index.html, puis restaurez la dernière version (en vérfiant l'état du fichier à chaque fois, ce qui se fait simplement en rafraîchissant la page sur le navigateur). N'oubliez pas de terminer par « git checkout master » avant de continuer.

Ajoutez une image

Quelles sont les commandes à utiliser pour que le nouveau fichier soit inséré dans le dépôt ?

Essayez et vérifiez l'état du répertoire avec git status

Sauvegarde sur une clef

Il est possible de sauvegarder tout ou partie d'un dépôt dans un seul fichier compressé avec la commande git bundle :

`git bundle create` **`nom_du_fichier`** `master`

1. Quelle est la taille de ce fichier ? Comparez avec la taille des fichiers dans le répertoire de travail et le nombre d'archives (commits).

Le fichier ainsi créé peut être transmis (par exemple sur une clef, par mail, par un ftp, par un service type dropbox etc..). A partir de ce fichier, on peut cloner le dépôt d'origine :

```
git clone nom_du_fichier_archive -b master
```

Si l'on dispose déjà d'une version du dépôt, on peut faire un git pull sur ce fichier :

```
git pull nom_du_fichier_archive master
```

1. Faites une modification sur un de vos fichier, puis le commit, puis partagez votre dépôt avec un camarade sans passer par le dépôt distant.

TD 2 (Echanges)

L'objectif de ce TD est d'utiliser git pour mettre en commun les pages perso crées lors du premier TD.

Vérification des pages

Complétez votre page, en utilisant régulièrement les commandes vues précédemment pour faire des sauvegardes. Vérifiez que vous avez bien changé votre nom et ajouté une photo. Complétez en incluant une courte présentation de votre parcours, ainsi que vos centres d'intérêt et votre projet professionnel.

Vérification du dépôt commun

Le partage se fera en envoyant toutes les mises à jour sur le même dépôt. La commande

```
git remote -v
```

permet de voir les dépôts distants configurés. Quels sont-ils ?

A votre avis, quand et comment ces informations ont-elles été ajoutées ?

Mise à jour du dépôt distant.

Votre dépôt local contient les différentes versions de votre travail. L'étape suivante est le partage de ces versions (commits) avec le reste du groupe.

1. Vérifiez que votre site s'affiche correctement dans le navigateur, et que vos liens fonctionnent
2. Vérifiez avec `git status` que votre dépôt est propre, c'est à dire que le répertoire de travail contient bien la même chose que le dernier commit. **Attention**, le fichier mon_nom.html ne doit plus être présent dans votre répertoire (il a normalement été renommé au cours du TD 1). Corrigez si nécessaire la situation.
3. La commande `git diff` **`ref`** permet de comparer la version actuelle d'un fichier et la version dans n'importe quelle archive.
   Vérifiez que la version actuelle de votre fichier index.html ne diffère de l'original que par le lien vers votre page. Corrigez si nécessaire la situation.
4. La commande `git push` permet de 'pousser' vos commits vers le dépôt d'origine. git push --all
5. Que s'est-il passé ?
6. Quel est le nom de la branche dans laquelle s'effectue le travail ?

Mise à jour à partir du dépôt distant.

Pour récupérer la dernière version à jour, il faut réaliser deux actions : la récupération des commits distants (qui sont désignés par origin/branche), et la fusion :

```
git fetch
```

```
git merge origin/master
```

On peut aussi utiliser la commande suivante, qui réalise la même chose :

```
git pull
```

Comme vous l'avez probablement constaté, il est impossible d'envoyer sur le dépôt distant un commit qui n'est pas basé sur la version la plus à jour de ce dépôt. Au lieu de risquer d'égarer des données, Git prévient que les données ont été modifiées depuis votre dernière mise à jour.

1. Réalisez un diagramme indiquant les commits impliqués dans ce problème, en indiquant par des flèches les versions précédentes.

Gestion des conflits

La commande `git merge` (et donc la commande `pull`) peut aboutir à un conflit, si un même fichier a été modifié de manière différente dans les deux branches. Vous devriez normalement vous retrouver dans cette situation (sauf si vous êtes le premier à avoir exécuté `git push`).

1. Comment sont indiquées les zones de conflit ? (aller voir dans le fichier incriminé) Comment voit-on sa version et la version du dépôt distant ?
2. Corrigez le fichier (supprimez les indicateurs de conflit, et corrigez ce qui doit l'être)
3. Validez le résultat par `git add/commit`

Une fois les conflits résolus et validés, vous pouvez à nouveau tenter le `git push`.

Transfert par clef et autres moyens

Comme vous pouvez le constater, l'utilisation du dépôt centralisée peut-être problématique quand tout le monde veut mettre à jour simultanément. C'est pourquoi cette solution est rarement utilisée sur de grands projets.

Heureusement, il est possible d'échanger les mises à jour sans passer par un dépôt central.
On peut sauvegarder tout ou partie d'un dépôt dans un seul fichier compressé avec la commande git bundle :

```
git bundle create nom_du_fichier master
```

1. Quelle est la taille de ce fichier ? Comparez avec la taille des fichiers dans le répertoire de travail et le nombre d'archives (commits).

Le fichier ainsi créé peut être transmis (par exemple sur une clef, par mail, par un ftp, par un service type dropbox etc..). A partir de ce fichier, on peut cloner le dépôt d'origine :

```
git clone nom_du_fichier -b master
```

Si l'on dispose déjà d'une version du dépôt, on peut faire un git pull sur ce fichier :

```
git pull nom_du_fichier master
```

Pour gagner du temps, utilisez cette possibilité pour regrouper plusieurs dépôts sur un même compte grâce à des bundles, avant de synchroniser le tout avec le dépôt central. N'oubliez pas de résoudre localement les confits avant de partager.

Récupérez régulièrement les dernières versions (git pull).

Corrections diverses

1. Une fois les différentes pages mises à jour sur le dépôt distant, récupérez la dernière version et affichez le journal des modifications :

```
git pull
```

```
git log
```

1. Passez en revue (par exemple avec le navigateur) les différentes pages, et corrigez les fautes d'orthographes et/ou les erreurs dans le code. Pensez à faire un `commit` et un `push` régulièrement (à chaque fichier corrigé par exemple).

Bilan

Au vu du TP :

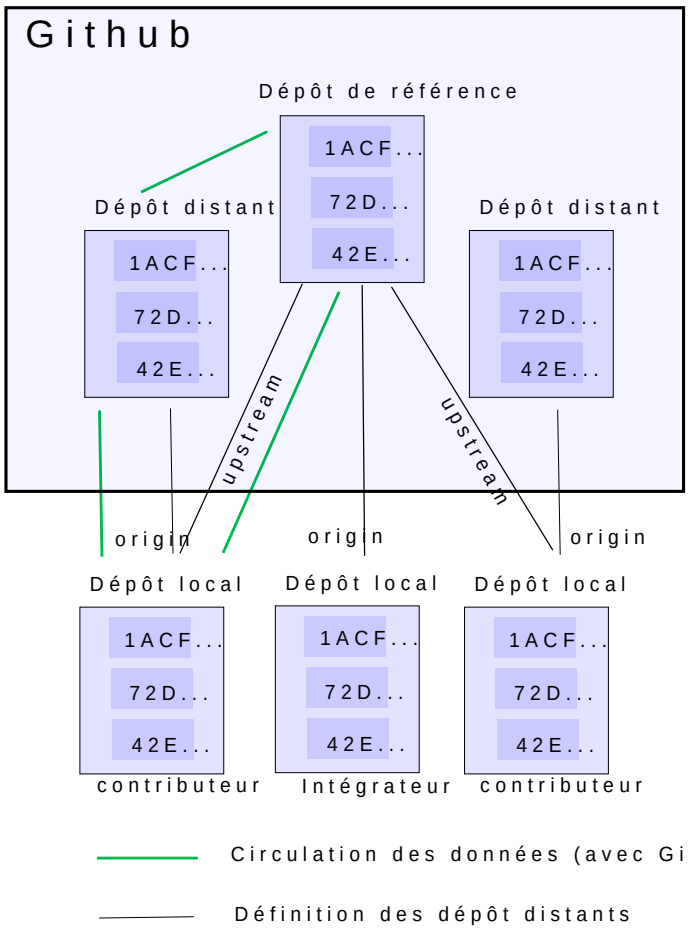
1. Quelles sont les étapes du TP qui ne nécessitent pas un accès au dépôt distant ?
2. Que se passe-t-il si le dépôt distant est perdu ? Et si un dépôt local est perdu ?
3. Est ce qu'un outil comme git permet de se dispenser de l'organisation du travail entre membres du groupe ? (justifiez)
4. A votre avis, pourquoi le message à chaque commit est-il obligatoire ? Même question pour l'auteur ?
5. Pour les projets très actifs et impliquant beaucoup de personnes, on utilise un système de patchs (fichiers décrivant les modifications entre deux versions) envoyés à un membre de l'équipe appelé intégrateur. C'est l'intégrateur (et lui seul) qui ajoute les nouvelles versions au dépôt distant. A votre avis, pourquoi ce mode de fonctionnement ? (il y a plusieurs raisons valables).

TD 3 (GitHub et branches)

L'objectif de ce TP est de réaliser une mini revue de presse, de manière collaborative.

Dépôt sur github

Pour ce TP, nous allons créer un dépôt sur github, un service d'hébergement de code basé sur Git. L'organisation retenue est celle décrite dans la page Les Dépôts , chapitre Dépôts distants individuels ; n'hésitez pas à lire ou relire ces pages.



1. Créez d'un compte sur https://github.com. Vous choisirez si possible un identifiant de la forme **NomPrenom**. Pensez à noter quelque part votre identifiant et votre mot de passe, afin de ne pas perdre de temps à les retrouver lors des prochaines utilisations.
2. Paramètrage local de Git

Si ce n'est pas déjà fait, configurez Git sur votre compte (nom, mail etc.. voir TP 1).
L'iut étant derrière un proxy, il est nécessaire de le préciser (il me semble que ce n'est plus nécessaire, je laisse la commande au cas où) :

```
git config --global http.proxy cache-etu.univ-artois.fr:3128
```

1. Obtention du dépôt

Chaque étudiant va disposer de son dépôt local (comme au TP 1) ainsi que d'un dépôt public

hébergé sur Github. Un des dépôts publics sera considéré comme la version de référence du groupe, son propriétaire sera l'intégrateur du projet. L'intégrateur est responsable de la validation (principalement technique) des modifications proposées, ainsi que de la fusion. En tant que garant de cohérence du projet, l'intégrateur est habituellement choisi pour sa compétence technique, son sérieux et sa fiabilité (ce n'est pas nécessairement le chef de projet, il est même préférable qu'il ne le soit pas).

Pour l'intégrateur :

L'intégrateur obtient un embranchement (fork) du dépôt indiqué par l'enseignant :
https://github.com/VincentDubois/TP2

Pour les autres :

Les autres étudiants obtiennent ensuite un embranchement du dépôt GitHub de l'intégrateur.

Pour tous :

Un embranchement s'obtient en cliquant sur 'fork' à partir de la page GitHub d'un projet. Une copie du projet est alors réalisée du projet à partir duquel vous avez cliqué sur votre compte GitHub.

La copie locale (sur votre machine) est ensuite obtenue par la commande clone, en utilisant l'url donnée sur votre page du projet GitHub en cliquant sur (https) :

```
git clone url répertoire_destination
```

L'utilisation d'un embranchement est détaillé dans l'aide de github :

https://help.github.com/articles/fork-a-repo/

En continuant à suivre ces instructions (vous avez déjà fait l'équivalent de la première moitié de ce tutoriel, vous êtes à l'étape 3 : ajout des « remotes »), ajouter un lien vers 'upstream' (qui désignera l'url http du dépôt officiel, c'est à dire celui de l'intégrateur).

remarque : *l'intégrateur n'a pas à configurer un 'upstream', puisqu'il est lui-même la référence du projet pour le groupe. Pour l'intégrateur 'upstream' et 'origin' désigneraient le même dépôt.*

Vérifications

Normalement, votre environnement de travail est maintenant prêt. Avant de commencer à travailler, il est recommander de vérifier toute l'installation.

Sur GitHub

Vérifiez que vous disposez bien sur votre compte d'une copie du dépôt du projet. Pour cela, il suffit de regarder la liste de vos dépôt.
Ensuite, vérifiez en allant sur ce dépôt que le dépôt d'origine est bien celui de votre intégrateur.

En local

Vous devez normalement disposer d'une version locale du dépôt git. Placez-vous dedans et vérifiez qu'il s'agit bien d'un dépôt et pas d'une simple copie des fichiers (par exemple avec git status),

Normalement, les liens vers les dépôts distants doivent être les suivants :

- origin : lien vers votre dépôt GitHub
- upstream : lien vers le dépôt GitHub de votre intégrateur

Création d'un style

L'objectif va être ici de proposer chacun une feuille de style en modifiant par exemple les couleurs ou la disposition des éléments. Pour cela, vous utiliserez une des fonctionnalités les plus puissantes de Git, qui est la possibilité de créer facilement de nouvelles branches. Une branche désigne une version différente de la version principale (dont le nom par défaut est 'master'), qui évolue de manière indépendante. Les branches peuvent ensuite être fusionnées, combinant ainsi le travail réalisé sur chacune.

Pour créer votre feuille de style :

1. Créez votre branche et placez-vous dedans :

```
git checkout -b nom_de_branche
```

(*En fait, il s'agit d'un raccourci pour*

```
git branch nom_de_branche
git checkout nom_de_branche
```

(La commande branch crée la branche, et la commande checkout permet de placer la branche dans votre espace de travail)

pour afficher toutes les branches, et la branche dans laquelle on se trouve :

```
git branch -a
```

(La commande `git status` affiche aussi la branche dans laquelle vous êtes.)

1. Modifiez, testez et sauvegarder la feuille de style. N'oubliez pas les add/commit réguliers !
2. Pour mettre votre style en ligne (sur votre dépôt public), utilisez la commande :

```
git push --all
```

Cette commande envoie toutes vos archives (y compris les nouvelles branches) sur le dépôt origin (le dépôt distant par défaut, normalement configuré sur votre dépôt Github).

A partir du moment où une version du projet est sur Github, il est possible de la télécharger sous forme d'archive zip à partir de votre page sur le projet (attention à bien sélectionner la bonne branche), ce qui peut être utile pour consulter les versions de chacun sans avoir à faire un clone ou un pull.

1. Décidez (ensemble) quel style est retenu pour le groupe.
2. Partagez le nouveau style

Avant tout, vérifiez que vous avez bien fait votre commit depuis les dernières modifications

Pour le style retenu

Demandez à l'intégrateur d'insérer vos modifications, cliquez sur le bouton 'Pull request' :

https://help.github.com/articles/using-pull-requests

Notez que GitHub permet de commenter ces demandes avant d'accepter l'ajout.

Pour tout le monde (y compris le style retenu)

```
git checkout master
git pull upstream master
git push
```

TD 4 (Travail sur un même document)

Ce TD continue et achève le TD précédent. L'objectif est de compléter la page du projet pour obtenir une revue de presse, en utilisant le style défini précédemment.

Préparation

Avant tout, assurez-vous des points suivants :

- Votre dépôt est propre
- Vous êtes dans la branche master
- Votre style est bien celui retenu lors du TD 3
- Vous avez configuré origin et upstream vers les bon dépôts (respectivement votre dépôt github et celui de votre intégrateur)

Si un de ces points n'est pas vérifié, corrigez le problème (éventuellement en retournant voir dans le TD précédent les commandes à utiliser)

Pour limiter les conflits, il est recommandé de préparer dans le fichier html un emplacement par article, et d'indiquer clairement (par exemple par des commentaires dans le html) qui sera responsable de quelle partie. Pour cela, il faut donc :

1. Se répartir les rôles (Edito, articles etc…)
2. Editer le fichier html en conséquence (une seule personne édite et soumet la nouvelle version du fichier).
3. Chacun met à jour sa version du fichier html sur le dépôt officiel.

En fonction de la taille du groupe, il peut être judicieux de créer des pages par rubriques ou par thématique, chacune de ces pages ayant son propre responsable (exemple : une rubrique sport, une rubrique international etc…).

Création du contenu

Pour remplir votre partie, vous pouvez utiliser par exemple l'afp (http://www.afp.com/afpcom/fr/content/news)
ou tout autre site d'information, thématique ou non (ou même votre imagination).

Le contenu lui-même a peu d'importance pour ce TP, l'important est que chacun ait l'occasion de réaliser une contribution en passant par GitHub *(restez tout de même dans le domaine du raisonnable, et tenez des propos corrects: il vous faudra assumer la responsabilité de ce que vous écrivez, y compris sur le plan légal…)*. Pensez à faire des commits réguliers.

Une fois chaque article terminé, il faut le soumettre (pull request) à votre responsable de rubrique ou à l'intégrateur.

Ensuite, vous pouvez ajoutez une miniature (pensez à définir des dimensions standards…) à votre article.

TD 5

Un canevas de site vous est proposé sur le dépôt git suivant :

https:github.com/VincentDubois/TP3

A partir de ce canevas, et en utilisant votre compte gitHub (que vous préciserez sur la feuille d'appel), vous devrez créer un site répondant à la demande de l'enseignant. Vous êtes libres de choisir le mode d'organisation que vous souhaitez. Une fois l'organisation décidée (en 15 minutes maximum), tous les échanges se font par les outils informatiques de GitHub (pas de discussion/mail/messagerie autres que ce qui est proposé sur ce site).

Tous les documents sont autorisés, y compris ceux en ligne. Si vous êtes bloqués, vous pouvez demander de l'aide à l'enseignant (comme pour un TP normal).

Les éléments évalués sont :

Pour le groupe :

- Organisation du groupe
- Contribution de chacun au projet (que ce soir sur le fond, la forme, ou même la vérification/correction. Comme dans un projet réel, il n'est pas nécessaire que chacun contribue à tous les aspects, il faut juste que chacun contribue à au moins un aspect).
- Modification au minimum du contenu et du style. Il est recommandé aussi (en fonction de la taille du groupe) de changer la navigation (quelques pages en plus, per exemple par catégorie, ou ajout d'une structure et de lien dans la page elle-même.
- (dans une moindre mesure) Qualité du résultat final. Recopier un contenu en ligne, en précisant la source sous forme de lien est acceptable pour cet exercice. Néanmoins, produire au final un contenu original est préférable (résumé ou synthèse, création personnelle)

Pour chaque étudiant :

- Création d'une copie sur GitHub/locale du dépôt
- Configuration de votre espace de travail (sur le pc)
- Modifications locales/sur GitHub/sur le projet
- Contributions (au moins trois, avec un ajout de fichier si cela a un sens dans votre contribution)
- Utilisation correcte et adéquate des fonctionnalités de l'outil de gestion de projet.

Remarques :

- Avant de demander de l'aide, consultez la faq. Si la réponse à votre problème était dedans, vous pouvez perdre des points…
- Intégrez rapidement le travail de chacun, même sous forme d'ébauche. Cela permet de détecter les problèmes plus tôt — et donc vous laisse une chance de les corrigez.
- Toujours vérifier l'apparence du site avec vos modifications avant de les partager ! L'intégrateur vérifie les modifications apportées aux fichiers, il n'a pas d'accès facile au résultat final, vous si !
- Contrairement à un projet réel, n'aidez pas vos collègue sur l'utilisation des commandes git (puisque le bon usage de celle-ci est évalué pour chaque étudiant)

FAQ

Q : *Comment obtenir l'aide sur git ?*

R : `git help nom_de_la_commande`

Q : *Je suis perdu !*

R : Essayez la commande `git status`

Q : *Je n'arrive pas à faire* `git clone` **`url`***!*

R : Vérifiez :

1. la configuration du proxy *(Attention aux copier/coller, les – ne passent pas !)*
2. l'url *(à l'iut, seules les URL en https:// et les fichiers locaux sont utilisables avec git)*

Q : *Aucune commande ne fonctionne, pas même* `git status` *!*

R : Etes-vous dans un dépôt git ? Autrement dit, y a-t-il un répertoire .git là où vous êtes ?

Q: *La commande* `git push/pull/merge` *refuse de s'exécuter* !

R: Vérifiez si votre répertoire de travail est propre (i.e. correspond au dernier commit). Si ce n'est pas le cas, faites un commit avant de recommencer (attention, si vous avez changé de branche, pensez à revenir à la bonne branche avec `git checkout` branche avant de faire le commit!)

Q : *Quand je fais* `git push`, *Git répond que tout est à jour. J'ai pourtant modifié des fichiers …*

R : Vérifiez avec `git status`. Vous avez probablement oublié de faire `git commit`…

Q : *Quand je fais* `git push`, *Git me demande mon identifiant et mon mot de passe, et rejette ma connection. Pourtant, je suis sûr d'avoir d'avoir tapé les bons …*

R : Vérifiez le dépôt sur lequel vous envoyez vos données (par exemple avec `git remote -v`.) Si le dépôt n'est pas à vous, vous n'avez généralement pas les droits pour le modifier.

Q : *Je viens de faire un clone/pull sur mon dépôt distant, mais il me manque des fichiers !*

R : Les fichiers n'ont probablement pas été ajoutés à la dernière archive. Il faut faire

`git add` **`nom_fichiers`**

avant chaque sauvegarde(commit). Alternativement, on peut utiliser `git commit -a`

(Mais les nouveaux fichiers ne seront pas ajoutés automatiquement).

Q : *Quand je tape* `git pull`, *Git me dit qu'il ne sait pas quoi faire* !

R : Essayez `git pull` **`remote branch`** (remote étant le dépôt distant, et branche le nom de la branche à récupérer)

Q : *J'ai /égaré/modifié par erreur/supprimé par accident/ un fichier, ou raté ma fusion… Comment retrouver une version précédente d'un fichier ?*

R : Utilisez `git checkout` **`version fichier`** (version correspond à un nom de version : master, ma branche, 1ABCF…, et fichier est le nom du fichier à récupérer). Pour annuler la fusion, voir question suivante.

Q : *J'ai fait n'importe quoi, je souhaite revenir au dernier commit !*

R : `git reset --hard HEAD` *(attention, les modifications apportées aux fichiers depuis le dernier commit sont définitivement perdues!)*

Aide-mémoire

Création de dépôt git	
`git init`	Crée un dépôt dans le répertoire courant
`git init --bare`	Le dépôt est créé sans répertoire de travail (utile pour un dépôt partagé)
`git clone` **`url`**	Duplique le dépôt donné par **url**
Aide et statut	
`git help` **`commande`**	Manuel de la **commande** git.
`git status`	Donne l'état du répertoire de travail.
`git branch -a`	Donne la liste des branches.
`git remote -v`	Indique tous les dépôts distants définis.
Configuration générale	
`git config –global user.name` **`'Votre nom'`**	
`git config --global user.mail` login@provider.net	
`git config --global core.editor nano`	
`git config --global http.proxy cache-etu.univ-artois.fr:3128`	
`git config --global github.user` **`identifiant_Github`**	
Sauvegarde du répertoire de travail	
`git add` **`fichiers`**	Sélection des fichiers à intégrer dans l'archive
`git rm` **`fichiers`**	Supprime les fichiers (faire un commit après)
`git commit`	Crée une archive avec les **fichiers** ajoutés.
`git commit -a -m` **`'message'`**	Idem. Les fichiers de la dernière archive sont ajoutés automatiquement.
Branches	
`git branch` **`branche`**	Crée une branche .
`git checkout` **`branche`**	Se place dans la branche.
`git checkout -b` **`branche`**	Équivalent aux deux commandes précédentes.
`git merge` **`branche`**	Fusionne la branche actuelle avec **branche**.
Dépôts distants	
`git remote add` **`dépôt url`**	Ajout d'un lien vers un **dépôt** distant.
`git remote rm` **`dépôt`**	Suppression du lien vers **dépôt**.
`git fetch` **`dépôt`**	Récupère les archives d'un **dépôt** (origin par défaut)
`git pull` **`dépôt branche`**	Récupère et fusionne la **branche**.
`git push --all` **`dépôt`**	Envoie toutes vos archives vers le dépôt.