Capsule 2 – Commandes de base et bonnes pratiques de collaboration

Hubert Cadieux et Arnaud Beaulé

Introduction (00:00 - 00:30)

Bonjour tout le monde !

Dans la première capsule, nous avons vu ce qu'étaient Git et GitHub, pourquoi ces outils sont utiles en sciences sociales, et nous avons appris le vocabulaire de base. Maintenant, il est temps de passer à la pratique !

Dans cette deuxième capsule, nous allons voir concrètement comment collaborer avec Git et GitHub. Pour illustrer tout ça, nous allons suivre l'exemple d'Hubert et d'Arnaud, deux étudiants en sciences sociales qui travaillent ensemble sur un projet d'analyse sur le développement économique et social des pays.

À travers leur collaboration, vous allez apprendre les commandes essentielles, comprendre le flux de travail collaboratif, et découvrir les bonnes pratiques qui vous feront gagner du temps et éviter les problèmes.

> **Note**
>
> Pour cette capsule, on pourrait partager l'écran en changeant la couleur du header selon l'étudiant : une couleur pour Hubert, une autre pour Arnaud, avec une icône différente pour chacun. Ça rendrait clair quand on passe d'un étudiant à l'autre.

Démarrer un projet collaboratif avec Git et GitHub (00:30 - 02:45)

Alors, commençons par la première étape. Dans tout projet collaboratif avec Git et GitHub, quelqu'un doit d'abord créer le dépôt principal sur GitHub. C'est ce qu'on appelle "initialiser le projet".

(Note affichée au bas de l'écran : Rappel du vocabulaire vu dans la capsule 1)

ÉCRAN D'HUBERT

Hubert prend l'initiative de créer le projet. Il se rend sur github.com et navigue vers l'organisation où il veut héberger leur projet collaboratif *(Est-ce qu'on crée une organisation FSS? Est-ce qu'on montre comment créer sa propre organisation? Est-ce qu'on fait l'exemple dans un repo personnel?).*

Voici les étapes qu'il suit :

1. Il clique sur l'onglet **Repositories** dans l'organisation
2. Puis sur le bouton vert **New repository**

GitHub lui présente alors un formulaire à remplir. Les éléments essentiels à considérer :

- **Nom du dépôt** : Hubert choisit un nom descriptif, en minuscules, sans accents ni espaces - par exemple "analyse-dev-pays-2025"
- **Visibilité** : Il choisit entre "Public" (visible par tous) ou "Private" (accès restreint aux collaborateurs). Dans son cas, il choisit "Public" puisqu'il compte partager ses résultats avec la communauté scientifique.
- **README** : Il coche la case pour créer automatiquement un fichier README - c'est la page d'accueil de son projet qui expliquera de quoi il s'agit, comment d'autres personnes peuvent collaborer sur ce projet, comment le reproduire, etc.

Les autres options du formulaire (comme .gitignore, licence, etc.) ne sont pas nécessaires pour commencer. Ce sont des options plus avancées que vous pourrez explorer une fois que vous serez familiers avec Git et GitHub.

Il clique maintenant sur `Create repository`, et c'est fait: le dépôt est maintenant créé. Il ne contient rien encore, mais il est prêt à être utilisé par Hubert et Arnaud pour commencer leur travail d'analyse.

Le cycle Git essentiel : commandes de base (02:45 - 07:15)

`git clone`

Maintenant, Hubert veut pouvoir ajouter et modifier des fichiers dans ce dépôt en travaillant de son poste local, de son ordinateur. Pour ce faire, il va utiliser la commande `git clone`.

La commande `git clone` permet de télécharger une copie complète d'un dépôt distant sur votre ordinateur. Contrairement à un simple téléchargement de fichiers, `git clone` copie tout l'historique des versions et configure automatiquement la connexion avec le dépôt GitHub. C'est comme avoir une machine à voyager dans le temps pour votre projet, directement sur votre ordinateur. Pour chaque projet, nous utilisons `git clone` qu'une seule fois: une fois la

copie du dépôt créée sur son ordinateur, nous allons utiliser d'autres commandes, qu'on verra plus tard dans la capsule, pour travailler sur le projet.

ÉCRAN D'HUBERT

Voici comment ça marche :

1. **Sur GitHub** : Dans son dépôt, Hubert clique sur le bouton vert **Code**, puis copie le lien HTTPS
2. **Dans son terminal** :
    - Sur Mac : Il cherche "Terminal" dans Spotlight
    - Sur Windows : Il ouvre "Command Prompt" ou "PowerShell" via le menu Démarrer
3. **Navigation** : Il se déplace dans le dossier où il veut travailler avec la commande `cd` (Change Directory)
4. **Clonage** : Il tape `git clone` suivi du lien HTTPS copié depuis GitHub

Et voilà ! Le dépôt apparaît maintenant sur son ordinateur, dans le dossier actuel du terminal, entièrement configuré pour se synchroniser avec le dépôt hébergé sur GitHub. Hubert peut maintenant travailler localement avec ses outils préférés comme RStudio.

`git add`, `git commit` et `git push`

ÉCRAN D'HUBERT

Hubert peut maintenant ouvrir RStudio dans le dossier du dépôt qu'il vient de cloner. Il commence à faire une analyse simple en créant un script pour explorer les données Gapminder.

[Accéléré du script qui s'écrit dans RStudio]

Une fois ce script `exploration.R` enregistré, il n'existe encore que sur son ordinateur local. Le dépôt distant sur GitHub ne contient pas encore ce nouveau fichier - nous pouvons le vérifier en consultant le dépôt en ligne.

[Démonstration sur GitHub montrant l'absence du fichier]

Pour synchroniser son travail avec GitHub, Hubert doit utiliser trois commandes essentielles qui forment le cœur du workflow Git :

1. `git add` - Préparer les changements

Pensez à `git add` comme mettre des documents dans une enveloppe avant de les poster. Cette commande indique à Git : "Je veux inclure ce fichier dans ma prochaine sauvegarde."

Dans son terminal, Hubert tape :

```
git add exploration.R
```

2. `git commit` - Créer un point de sauvegarde

`git commit` crée une "photo" de l'état actuel des fichiers préparés, avec un message descriptif. C'est comme signer et dater l'enveloppe.

```
git commit -m "Ajout du script d'exploration des données gapminder"
```

Ce message est important, alors qu'il permet aux gens qui collaborent avec vous de comprendre les changements apportés au projet. Il doit être court, précis et concret.

3. `git push` - Envoyer vers GitHub

`git push` envoie tous les commits locaux vers le dépôt distant sur GitHub, rendant les modifications accessibles aux collaborateurs.

```
git push
```

[Schéma de Git qui montre les différences entre les commandes]

[Retour sur GitHub]

En consultant le dépôt sur GitHub, Hubert peut maintenant constater que :

- Le script `exploration.R` apparaît dans les fichiers
- Le commit "Ajout du script d'exploration des données gapminder" est visible dans l'historique
- En cliquant sur ce message de commit, il peut voir exactement quelles lignes de code ont été ajoutées

Le travail d'Hubert est maintenant prêt à être partagé avec Arnaud !

Arnaud rejoint le projet : clone et première contribution (07:15 - 08:05)

ÉCRAN D'ARNAUD

Arnaud veut maintenant rejoindre le projet. Puisque Hubert a déjà créé le dépôt et ajouté son premier script, Arnaud n'a qu'à cloner le dépôt existant pour commencer à travailler.

`git clone` du projet existant

Arnaud se rend sur GitHub, trouve le dépôt `analyse-dev-pays-2025`, copie l'URL HTTPS, puis dans son terminal :

```
git clone https://github.com/organisation/analyse-dev-pays-2025.git cd analyse-dev-pays-2025
```

Parfait ! Arnaud a maintenant une copie locale du projet avec le script d'Hubert inclus.

Contribution d'Arnaud : analyse du PIB

Arnaud ouvre RStudio et décide de compléter l'analyse en créant un script sur l'évolution du PIB par habitant :

[Accéléré d'Arnaud créant son script `analyse_pib.R`]

Synchronisation rapide

Arnaud applique maintenant le workflow qu'il a vu avec Hubert :

```
git add analyse_pib.R git commit -m "Ajout analyse évolution PIB par continent"
git push
```

[Retour sur GitHub]

Le dépôt contient maintenant les deux scripts : celui d'Hubert sur l'espérance de vie et celui d'Arnaud sur le PIB. La collaboration a commencé !

Synchronisation : Hubert récupère les modifications d'Arnaud (08:05 - 09:45)

ÉCRAN D'HUBERT

Hubert reçoit une notification d'Arnaud ou voit directement sur GitHub qu'Arnaud a ajouté son script d'analyse du PIB. Pour continuer à travailler sur la version la plus récente du projet, Hubert doit maintenant récupérer les modifications d'Arnaud sur son ordinateur local.

[Montrer dans le working directory d'Hubert qu'on ne voit pas le script d'Arnaud]

`git pull` - Récupérer les dernières modifications

La commande `git pull` télécharge et intègre automatiquement les dernières modifications du dépôt distant dans votre copie locale. C'est l'inverse de `git push` : au lieu d'envoyer vos modifications vers GitHub, vous récupérez les modifications des autres.

Dans son terminal, Hubert tape simplement :

```
git pull
```

[Démonstration dans le terminal montrant le téléchargement du fichier]

Immédiatement, Hubert voit apparaître le fichier `analyse_pib.R` d'Arnaud dans son dossier local. Il peut maintenant ouvrir RStudio et voir le script d'Arnaud, l'exécuter, et continuer à travailler sur la version la plus récente du projet.

Bonne pratique : toujours faire `git pull` avant de commencer

Il est recommandé de toujours faire `git pull` au début de chaque session de travail pour s'assurer d'avoir la version la plus récente du projet, donc avant de faire `git add`, `git commit` et `git push`. Cela évite les conflits et garantit que vous travaillez sur la base la plus à jour.

Maintenant, Hubert et Arnaud peuvent travailler en parallèle, chacun récupérant régulièrement le travail de l'autre !

Régler un conflit de fusion (09:45 - 11:15)

ÉCRAN D'HUBERT

[Hubert modifie le fichier README.md pour ajouter une description de son analyse de l'espérance de vie. Il commit et push ses changements.]

ÉCRAN D'ARNAUD

[Pendant ce temps, Arnaud modifie aussi le README.md pour décrire son analyse du PIB, mais dans la même section que Hubert. Il commit ses changements localement, puis essaie de faire git push.]

```
git push
```

[Git refuse le push et affiche un message d'erreur indiquant qu'il y a des modifications distantes.]

Arnaud doit d'abord récupérer les changements d'Hubert :

```
git pull
```

[Git détecte un conflit et affiche un message indiquant qu'il y a un conflit de fusion dans README.md]

[Arnaud ouvre le fichier README.md dans RStudio. Le fichier contient maintenant des marqueurs de conflit]

Résolution du conflit :

[Arnaud supprime les marqueurs de conflit (`<<<<<<<`, `=======`, `>>>>>>>`) et combine les deux descriptions :]

[Arnaud sauvegarde le fichier, puis finalise la résolution :]

```
git add README.md       git commit -m "Résolution conflit: fusion descriptions analyses" git push
```

Le conflit est résolu ! Les deux contributions sont maintenant intégrées dans le projet.