





Masters 1 & 2 - Coordination Front & Back

Session 3 - Automatisation avec Git

Par Julien SOSTHENE & Olivier HUSSON - 2021



Sommaire

1. Travailler avec des dépôts interdépendants : Git Submodules 
2. Committer proprement avec les Git Hooks 
3. Git Flow 
4. Pipelines CI/CD 

1. Travailler avec des dépôts interdépendants : Git Submodules



1.1 Kézako ?

Parfois, deux (ou **+**) dépôts Git sont interdépendants :

- Votre application dépend d'une bibliothèque interne utilisée dans plusieurs projets
- Votre back-end construit et sert votre front-end
- ...

🤔 Solution 1

Copier coller le code de la bibliothèque et gérer ses versions directement dans le dépôt dépendant

- ✗ L'état de la bibliothèque est fixe et devra être mis à jour à la main dans chaque projet à chaque patch
- ✗ On perd l'historique des commits de la bibliothèque, ses branches, etc

🤔 Solution 2

Passer par un package manager tiers : NPM, Composer, Gems, etc...

- ❌ Les deux projets doivent utiliser le même package manager (ex : front avec NPM et back avec Composer)
- ❌ Le sous-projet est souvent dans le `.gitignore` (submodules) et ne permet pas de travailler directement dans la même hiérarchie de fichiers

- ✗ Workflow pour mettre à jour la bibliothèque :
- Passer dans le projet de la bibliothèque
 - Faire la modification
 - Commiter
 - Publier une nouvelle version sur le package manager
 - Repasser sur le projet dépendant
 - Mettre à jour la version de la bibliothèque
 - Lancer l'update

🤔 Solution 3

Git propose un outil de gestion de dépôts interdépendants : les *submodules*.

En bref, un submodule peut être considéré comme **un lien symbolique vers un commit particulier d'un autre dépôt** dans un sous-dossier d'un dépôt donné.

- ✅ Ils coexistent dans la même hiérarchie
- ✅ Ils peuvent être mis à jour indépendamment

1.2 Fonctionnement

On crée un lien de dépendance avec

```
git submodule add git@example.com/unautredepot.git dossiercible
```

💡 Par défaut, ce sous-dépôt sera placé sur le bout de la branche principale.

A partir de là, les modifications faites dans un dépôt ou dans l'autre sont prises en compte pour le dépôt courant, pas pour l'autre !

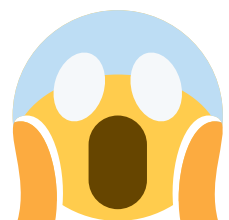


Fichier `.gitmodules`

Allons faire un tour dans le nouveau fichier qui a été créé, `.gitmodules`

Clonage d'un dépôt contenant un submodule

Clonons notre dépôt comme si nous étions un autre développeur 🧑💻



le dossier est vide !

Pour récupérer le contenu du dossier, il faut dire à Git d'utiliser les submodules pour ce dépôt :

```
git submodule init
```

Puis mettre à jour leur contenu

```
git submodule update
```

💡 Par défaut, le dépôt est en **Detached HEAD**, c'est à dire positionné sur un commit particulier et pas sur une branche.

👉😊 Vous pouvez en revanche passer sur une branche et travailler sur ce sous-dépôt normalement!

 Faisons un commit dans le dépôt "enfant".

La modification est perçue dans le dépôt parent! On peut alors commiter cette modification.

👉😊 En réalité, cela ne change que le commit vers lequel pointe notre submodule...

2. Committer proprement avec les Git Hooks



2.1 Kézako ?

Les *Git hooks* sont des scripts custom qui s'exécutent lors du déclenchement de certaines actions sur le dépôt :

- Après le checkout d'un fichier ou d'un ensemble de fichiers
- Avant un push ou un pull
- Avant de commiter
- Après avoir commité
- ...

On installe les hooks dans le dossier `.git/hooks` du dépôt.

Vous y trouverez une liste de hooks d'exemple en `.sample`

[Documentation officielle](#)



⚠ Attention, ces fichiers ne sont pas commités/copiés de client en client

👉😊 Il est de votre responsabilité de les distribuer à toutes les personnes qui doivent travailler sur le projet, où de **les implémenter directement sur le serveur.**

🤔 Gitlab/Github etc utilisent ces hooks pour leurs outils de CI/CD, directement côté serveur.

2.2 Hooks disponibles

Sur le client

- `pre-commit` - s'exécute avant le commit. Si le script retourne autre chose que zéro, le commit est annulé (ex: linting du code)
- `prepare-commit-msg` s'exécute avant de lancer l'éditeur de message de commit
- `commit-msg` s'exécute après la sauvegarde du message de commit (et annule le commit s'il renvoie autre chose que 0) (ex : validation des messages de commit)
- `post-commit` s'exécute quand le commit est enregistré

- **pre-rebase** s'exécute avant un rebase
- **post-checkout** s'exécute après un check-out (ex: remplacer certains fichiers, lancer une vérification d'intégrité...)
- **pre-push** s'exécute avant un push et annule le push s'il finit avec autre chose que zéro (ex : linting, exécution de tests automatiques)

Sur le serveur

- **pre-receive** s'exécute quand le client a fait un push, avant d'intégrer les changements (un résultat différent de zéro fait échouer le push)
- **update** s'exécute juste après, mais une fois par branche pushée s'il y en a plusieurs
- **post-receive** s'exécute après l'intégration des changements (ex : déploiement)

2.3 Démonstration

Créons un projet avec du Javascript et mettons en place des hooks client pour *linter* et formater le code avec eslint.

- ⚠ Si le lint n'est pas parfait, il sera impossible de pusher.

 Créons un script NPM pour installer notre *hook* automatiquement!

2.4 Des hooks au moment de faire `git add` ?

Il n'existe pas de hooks pour `git add`, en revanche, il existe des `filters` qui sont gérés par le fichier `.gitattributes`

- `smudge` est lancé pour transformer un fichier lorsqu'on l'ajoute à l'index
- `clean` est utilisé pour restaurer le fichier lorsqu'on le sort de l'index

💡 C'est ce qu'utilise Git LFS, maintenant installé par défaut avec git!

Documentation

3. Git Flow

3.1 Pourquoi Git Flow 🏊 ?

Git Flow est un ensemble de pratiques Git ayant pour but de systématiser l'organisation d'un dépôt auquel contribue une équipe entière de développeurs.

- Par le nommage des branches
- Par l'implémentation d'un workflow évitant la désorganisation du projet (A.K.A "Le bordel")
- Par la protection de branches sur lesquelles le projet doit être stable.

🤔 Exemple d'une organisation "à l'arrache":

- Chaque membre de l'équipe a sa propre branche nominative
- Lorsqu'un membre de l'équipe a besoin de l'avancée d'un autre, il fusionne la branche en question
- L'autre membre fait la fusion inverse pour récupérer les changements du premier, alors qu'ils ne sont pas vraiment stables et travaille sur une base instable
- `main` est mis à jour de manière sporadique par n'importe qui, entraînant des instabilités et des conflits

? Questions

- ? Quelle branche dois-je récupérer pour obtenir la feature X ?
- ? Peut-on livrer `main` sans risque d'introduire des bugs ?
- ? Comment éviter les fusions bi-directionnelles, souvent instables ?

En utilisant Git flow , on peut s'assurer

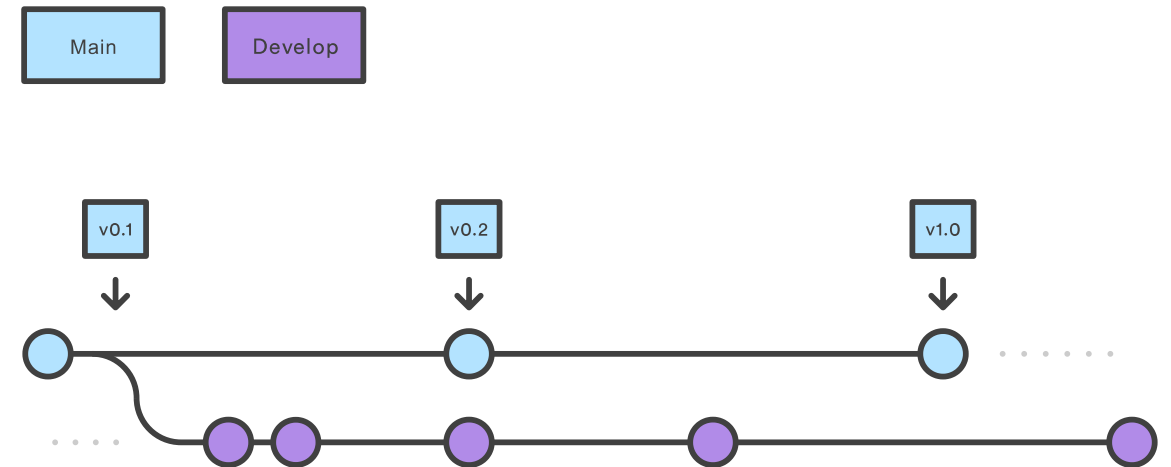
- De savoir où est une fonctionnalité à tout moment (nommage)
- De ne pas ajouter par erreur de commits erronés sur `main` ou de fonctionnalités incomplètes
- De produire des livraisons stables dans un contexte de livraisons régulières (ex : agile)
- D'aplanir la gestion des fusions et du partage des fonctionnalités entre les branches.

3.2 Fonctionnement

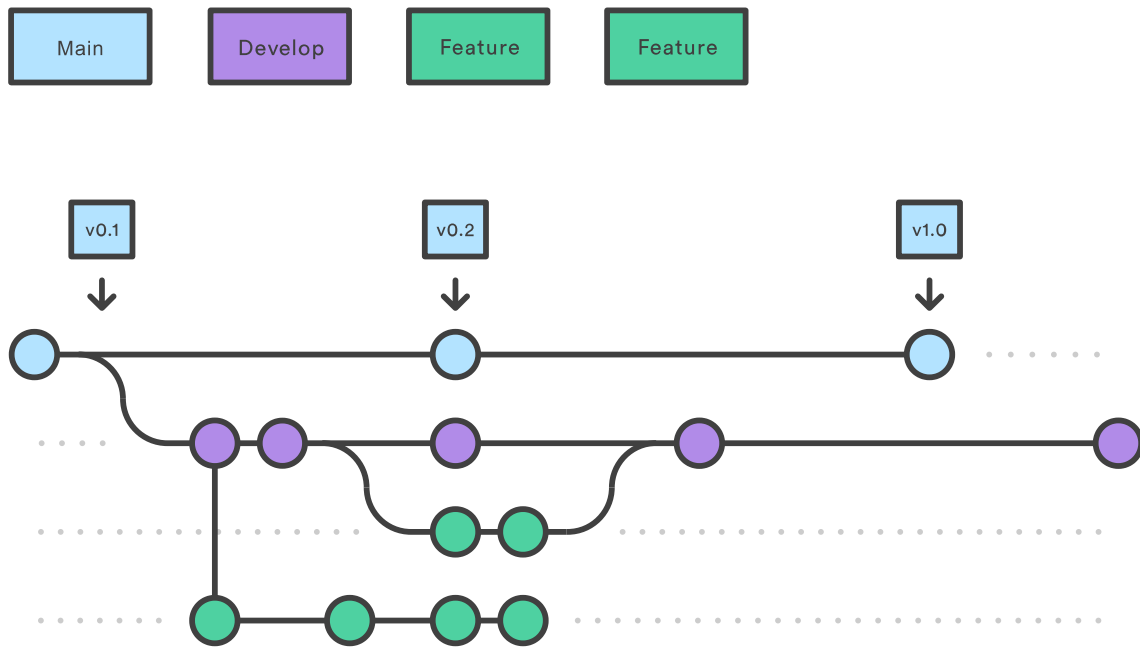
Git flow s'appuie sur deux branches principales :

- `main`
- `develop`

Crédit images



⚠ On ne pousse jamais directement sur `main` ou `develop`. A la place, on procède par *pull request* (ou *merge request*) via des branches de fonctionnalités



A chaque fois qu'un développeur travaille sur une fonctionnalité, il crée **une branche de fonctionnalité** (*feature branch*, ex : `feature/cool-feature`).

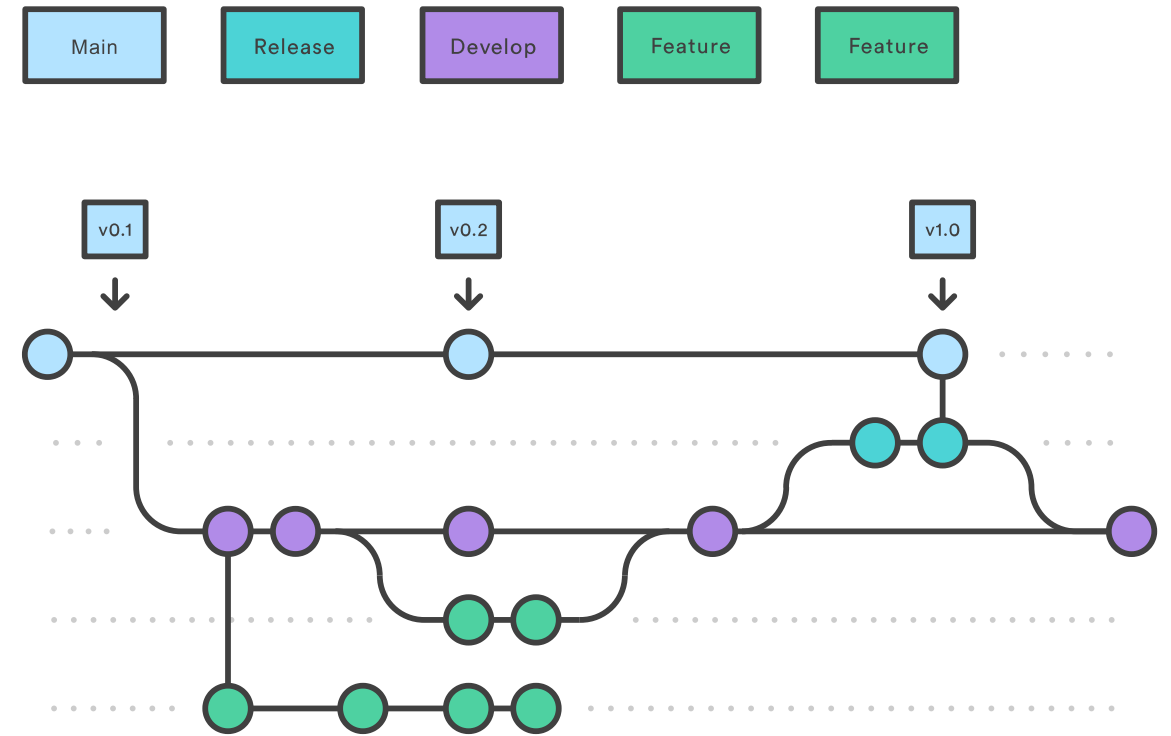
👉😊 Cette branche sera fusionnée dans `develop` lorsque la fonctionnalité sera considérée stable, via une *pull request*

La pull request est vérifiée par un *maintainer* avant d'être fusionnée dans *develop*.

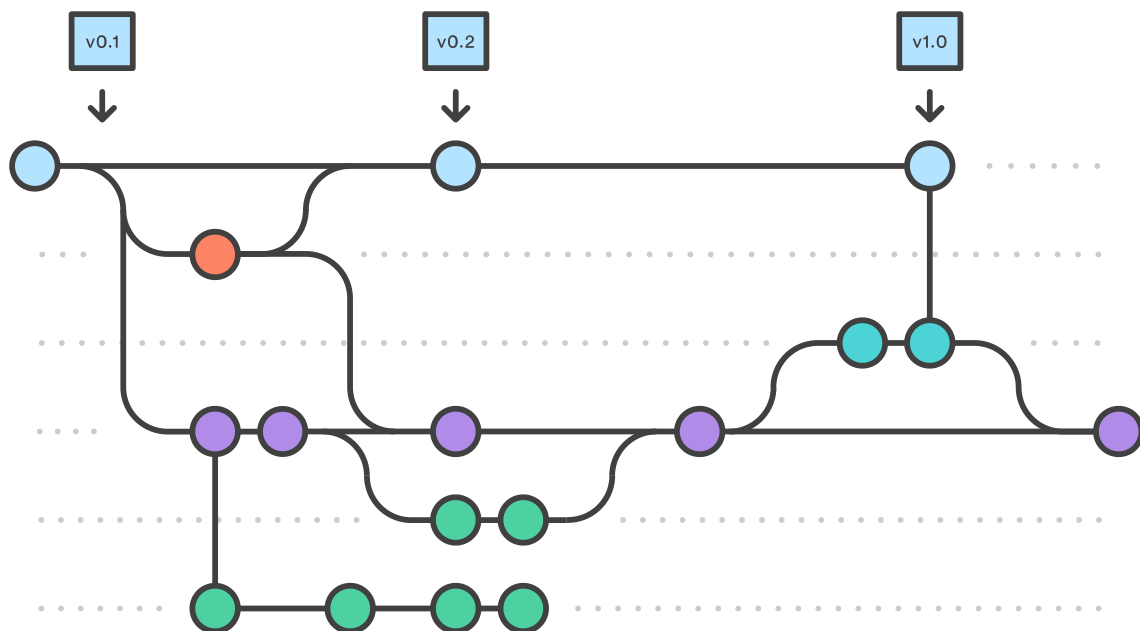
💡 Une fois fusionnée, on supprime la branche.

Lorsqu'on a assez de fonctionnalités pour créer une livraison, on crée une branche *release* (ex : `release/v1-1`) dans laquelle on fera les derniers ajustements pour la livraison.

💡 ceci permet de continuer à fusionner des choses sur `develop` en parallèle sans perturber la livraison.



👉😊 Une fois la branche `release` stable, testée et corrigée, on peut la fusionner dans `main` et de nouveau dans `develop` pour lui apporter les changements effectués durant la stabilisation.



En cas de correctifs à apporter à chaud sans attendre une livraison (ex : grave faille de sécurité) on crée à partir de **main** une branche **hotfix** qu'on intègre directement à **main** par une *pull request*.

3.3. Eviter les impairs : branches protégées

Protéger une branche signifie **empêcher** à cette branche d'être *pushée* sur le serveur.

💡 Ceci est une feature de Gitlab, Github, etc et pas de Git à proprement parler

La seule manière d'y apporter des modifications est de procéder par *pull request*, lesquelles doivent être approuvées par des utilisateurs avec un rôle suffisant.

👉😊 Dans Git Flow, on va traditionnellement protéger `main` et `develop`

Et en cas de conflit sur une pull request de branche protégée ? 🤔

En cas de conflit, créez une branche locale partant de la branche cible, et effectuez la fusion en local et résolvez les conflits avant de soumettre une nouvelle pull request !

3.3 Outil en ligne de commande

Git flow peut être imposé par l'utilisation d'une extension de git

Instructions d'installation

👉😊 Notez que l'extension a tendance à être très rigide, donc à utiliser ... ou non!

3.4 *Trunk based development*

Dans l'hypothèse de l'intégration continue, de plus en plus populaire, Git Flow devient moins populaire du fait de ses longues chaînes de commits sur les branches `features`.

Le *Trunk based development* propose un modèle

- avec une seule branche principale `main` ou `trunk`
- et des branches de "fonctionnalités" similaires à git flow, mais très courtes (comprendre : une branche = une user story)


💡 Ce sont l'exhaustivité des tests automatiques et les reviews manuelles des pull requests, lancés avant chaque fusion, qui assurent la stabilité sur la branche `main`

Si on veut éviter la publication de fonctionnalités avant l'heure, on utilisera des *features flags* permettant d'activer ou de désactiver une fonctionnalité dans un *build*.



4. Pipelines CI/CD

4.1 Pipelines ?

Les *pipelines CI/CD* (Github Actions, Gitlab Pipelines, Bitbucket Pipelines...) sont des actions automatiques en chaîne  se déclenchant lors d'actions sur le dépôt distant :

- Fusion sur une branche
- Pull request
- push sur n'importe quelle branche
- ...

! Ces outils ne sont pas standardisés !

Ce qui sous-entend que chaque hébergeur de dépôt a son propre système

Ces outils permettent l'**intégration continue**, la **livraison continue** ou le **déploiement continu**



Intégration continue

- Les changements de chaque développeurs sont intégrés à la branche principale environ une fois par jour/tous les deux jours.
- Les tests automatiques à l'intégration assurent l'intégrité du code
- Méthode privilégiée pour le *trunk based development*



Livraison continue

- Le code est testé automatiquement à l'intégration d'une nouvelle fonctionnalité
- Le code est compilé/*buildé* automatiquement si les tests ont réussi pour permettre une livraison à tout moment

Déploiement continu

- Le code est testé, compilé et **déployé sur un serveur de test** à chaque intégration de fonctionnalité.
- La version sur le serveur de test est **testée et approuvée**
- Le déploiement sur le serveur de production est rendu disponible et déclenché par une action d'un responsable.

Chaque projet étant différent :

- Un workflow différent (git flow, trunk based development...)
- Des technologies de développement différentes et des méthodes de *build* différentes (Node, Go, Ruby, TypeScript, etc.)
- Des environnements de tests et d'utilisation différents (Machine linux, windows, serveur, ordinateur de bureau, micro-services sur des machines multiples, présence de bibliothèques/modules hardware...)

🤔 Il est impossible de fournir des outils automatiques spécifiques adaptés à tous.

A la place, ces services proposent le lancement d'images Docker 🐳 éphémères et configurables pour exécuter des chaînes d'opérations en étapes (tests, compilation, déploiement):



Les Pipelines



4.2 Mais pas que !

D'autres outils, à installer séparément de votre hébergement de dépôt, peuvent servir à l'intégration continue :

- Jenkins, CircleCI, Bamboo

4.3 Le cas de Github et Gitlab

Gitlab et Github ont chacun leur propre format pour décrire les pipelines, mais les deux suivent le même principe ; c'est à dire **un document YAML** décrivant :

- Une image Docker préconfigurée
- Une configuration complémentaire
- Une suite d'étapes de tests, de compilation et de déploiement définies par l'utilisateur

Gitlab CI/CD vs Github Actions

Gitlab s'est longtemps différencié par ses outils de CI/CD extrêmement complets.

Github a fini par proposer Github Actions pour lui faire concurrence.

🤔 Les deux ont donc les mêmes capacités avec des syntaxes différentes

4.4 Pipeline de CI/CD Gitlab

Les pipelines Gitlab sont constituées :

- D'une image Docker de base
- De conditions de déclenchement de workflow
- D'une liste d'étapes (*stages*)
- D'une liste de *jobs* pour chaque *stage* (qui s'exécutent simultanément)

Pour cet exemple, nous allons **déployer** notre projet sur Heroku.

 Créons une *app* sur Heroku!

Il faudra également préciser les versions de Node et de NPM à utiliser dans le `package.json`

```
"engines": {  
  "node": "17.2.0",  
  "npm": "8.3.0"  
},
```

Nous allons avoir besoin de la clé d'API pour déployer le projet.
En attendant, allons configurer Gitlab!



La configuration de Gitlab CI/CD peut être pré-générée à partir d'un *template* pour nous donner une base :

CI/CD → Pipelines

👉😌 Ici, nous utiliserons l'image pour Node!

Commentons le template :

```
image: node:latest
```

Image Docker à utiliser comme base

services:

- mysql:latest
- redis:latest
- postgres:latest

On peut choisir un certain nombre de services complémentaires pré-configurés. Ici, nous n'aurons besoin d'aucun d'entre eux!

```
cache:  
  paths:  
    - node_modules/
```

Dossiers à mettre en cache pour les récupérer au déploiement suivant

💡 Rappelez-vous que chaque image est relancée à chaque lancement de la pipeline!

```
test_async:  
  script:  
    - npm install  
    - node ./specs/start.js ./specs/async.spec.js
```

Ces lignes servent à lancer des tests. Par défaut, les deux `test_async` et `test_db` s'exécuteront en parallèle!

Commençons par essayer de remplacer les commandes de test par des `echo` et vérifier l'exécution du tout.

💡 Nous reviendrons sur les tests automatiques un peu plus tard! 😊

Configurons notre pipeline

Pour déployer sur Heroku, nous aurons besoin du package de déploiement DPL (ruby)

On peut inclure cette installation dans notre configuration avec l'entrée `before_script`:

```
before_script:  
  - apt-get update -qy  
  - apt-get install -y ruby-dev  
  - gem install dpl
```

💡 N.B. Heroku gère déjà les submodules mais nécessite de rendre les mots de passe visibles dans le `.submodule` si le dépôt est privé

Nous aurons également besoin de renseigner nos clés d'API.

🤔 La clé est privée, mais si nous l'ajoutons dans le fichier `.gitlab-ci.yml`, il sera visible dans le dépôt... 🤔

On peut pour cela utiliser des variables de CI/CD dans les paramètres de Gitlab : **Settings** -> **CI/CD** -> **Variables**

 Créons une variable **\$HEROKU_API_KEY**

On peut ensuite ajouter un *job* pour le déploiement :

```
heroku :  
  type : deploy  
  script: dpl --provider=heroku --strategy=git \  
    --app=ynov-coordfb-test-app-glasses --api-key=$HEROKU_API_KEY
```


Pour que ce job ne s'exécute que pour les commits sur la branche `main`, on peut rajouter :

```
only:  
- main
```

4.5 Gérer 2 environnements de déploiement

Pour assurer la stabilité de nos applications, une bonne pratique consiste à avoir 2 environnements de déploiement :

- `prod` qui sera mis à jour sur la branche `main`
- `staging` qui sera mis à jour sur la branche `develop`

 Créons un autre job déployant sur une deuxième app, réservée au développement, suivant la branche `develop` !

4.6 Github Actions

Github actions est pensé de manière plus modulaire :

Vous pouvez utiliser des actions pré-configurées dans vos propres actions !

💡 Nous ne ferons pas un exemple complet mais nous pouvons aller regarder un exemple!

Next time on Coordination Front/Back

- Communication efficace en Agile
 - Avec le client
 - Avec l'équipe

