

Qualité de Code

Quelques outils et conventions

Support de cours IUT de Calais

Kevin GUERRIER

Lien vers le support de cours

<https://bit.ly/2024-iut-qualitecode-outils-conventions-cm-kguerrier>

Ce support est fortement inspiré du support de Valentin FOURNY, intervenant également à l'IUT de Calais et qui a mis à disposition sur

<https://formation-vfourny.vercel.app/ci-cd>

Préambule

Vous commencez à avoir déjà un certain nombre d'heures de code dans vos bagages et vous avez donc déjà pu vous rendre compte que cette totale liberté d'écriture peut très vite tourner à la catastrophe lorsqu'elle n'est pas cadrée. Dans ce cadre, de nombreux outils, accompagnés par de bonnes pratiques et le respect des conventions existantes, sont de précieux atouts dans votre quotidien.

Nous allons donc utiliser ces quelques heures pour voir quelques notions importantes dans ce domaine.

Linter et Prettier

Dans le cas où ces termes ne vous disent pas grand chose, ne vous inquiétez pas, vous avez certainement déjà rencontré ces 2 outils. Ils font généralement partie des extensions que vous installez dans votre environnement de développement et qui vous permettent de vous faciliter la lecture / l'écriture de vos lignes de codes dans un langage spécifique.

Un Linter va analyser votre code et vérifier qu'il est conforme à tout un ensemble de règles propres au langage que vous utilisez, et c'est notamment lui qui vous signale vos erreurs de syntaxes. Vous pouvez également l'utiliser pour lui demander de vérifier la conformité de votre code avec vos propres conventions.

Le Prettier, ou Formateur de code, a un rôle différent et vous permet d'automatiser la manière dont vos lignes vont être formatées. C'est généralement lui qui s'occupe de la coloration syntaxique pour vous faciliter la lecture, qui va gérer votre indentation, le positionnement de vos accolades, le fait de tronquer les longues lignes, ...

Si on prends un exemple de projet [Node.js](#), voici ce que cela pourrait donner :

ESLint

ESLint est l'outil de linting le plus populaire pour JavaScript et TypeScript. Il permet de détecter et de corriger les problèmes dans votre code. Vous retrouverez la documentation officielle sur <https://eslint.org/docs/latest>

Installation d'ESLint

Commencez par installer ESLint et ses dépendances :

```
$ npm init @eslint/config@latest
```

Vous remarquerez que nous ne passons pas directement par un npm install mais plutôt par un npm init. Voyez cela comme une pré-installation via assistant.

Lors de cette initialisation, il vous posera des questions sur le contexte de votre projet. Vous pourrez être amené à préciser le langage utilisé, si vous voulez uniquement vérifier les syntaxes ou si vous souhaitez rechercher des problèmes également, et bien sûr la manière dont utilisez vos imports (CommonJS ou JavaScript modules), si vous utilisez un framework spécifique, du typescript, ...

Une fois cette initialisation réalisée, il vous proposera donc d'installer les paquets nécessaires en fonction des réponses que vous lui avez apportées.

Configuration d'ESLint

Après l'initialisation, vous aurez un fichier `eslint.config.mjs` (ou `.json` ou `.yaml`) dans votre projet.

```
import js from "@eslint/js";
import globals from "globals";
import tseslint from "typescript-eslint";
import { defineConfig } from "eslint/config";

export default defineConfig([
  { files: ["**/*.js,mjs,cjs,ts"], plugins: { js }, extends: ["js/recommended"] },
  { files: ["**/*.js,mjs,cjs,ts"], languageOptions: { globals: globals.browser } },
  tseslint.configs.recommended,
]);
```

Vous pouvez personnaliser davantage les règles selon vos besoins. Pour plus d'informations, je vous invite à consulter la documentation officielle sur <https://eslint.org/docs/latest/use/configure/>

Ignorer des fichiers

Tout comme vous pouvez le faire pour vos dépôts Git, vous pouvez éditer votre fichier [eslint.config.js](#) pour y préciser la liste de certains fichiers ou répertoires à exclure du linting.

Par exemple :

```
export default defineConfig([
  [...]
  {
    // Note: there should be no other properties in this object
    ignores: ["node_modules", "dist", "build", "coverage", "**/*.disable.*"],
  },
]);
```

Prettier

Prettier est un formateur de code qui impose un style cohérent à travers votre base de code. Vous retrouverez la documentation officielle sur <https://prettier.io/docs/>

Installation de Prettier

```
$ npm install -D prettier eslint-config-prettier eslint-plugin-prettier
```

- prettier : Le formateur lui-même
- eslint-config-prettier : Désactive les règles ESLint qui pourraient entrer en conflit avec Prettier
- eslint-plugin-prettier : Exécute Prettier comme une règle ESLint

Configuration de Prettier

Vous devriez commencer à être habitué : la configuration passe par un fichier ! Créez donc un fichier `.prettierrc` à la racine de votre projet :

```
{
  "singleQuote": true, // https://prettier.io/docs/options#quotes
  "trailingComma": "all", // https://prettier.io/docs/options#trailing-commas
  "semi": false, // https://prettier.io/docs/options#semicolons
  "printWidth": 80, // https://prettier.io/docs/options#print-width
  "tabWidth": 2, // https://prettier.io/docs/options#tab-width
  "bracketSpacing": true, // https://prettier.io/docs/options#bracket-spacing
  "bracketSameLine": false, // https://prettier.io/docs/options#bracket-line
  "arrowParens": "always" // https://prettier.io/docs/options#arrow-function-parentheses
}
```

Ignorer des fichiers avec Prettier

Même principe : Créez un fichier `.prettiignore` pour exclure certains fichiers du formatage :



Scripts npm

Maintenant que vos configurations sont faites, il vous est possible d'ajouter des scripts pratiques à votre `package.json` pour lancer facilement ESLint et Prettier :

```
[...]
"scripts": {
  [...]
  "lint": "eslint .",
  "lint:fix": "eslint --fix .",
  "format": "prettier --write ."
},
```

ce qui vous donne accès aux commandes suivantes :

- `$ npm run lint` Vérifie le code avec ESLint
- `$ npm run lint:fix` Corrige automatiquement les problèmes ESLint
- `$ npm run format` Formate le code avec Prettier

Intégration avec VSCode

Pour une expérience de développement optimale, vous pouvez configurer VSCode pour formater automatiquement votre code à la sauvegarde. L'éditeur ne déroge pas à la règle, et vous pouvez donc créer un fichier de configuration `settings.json` présent dans un répertoire `.vscode` à la racine de votre projet, et qui pourrait contenir :

```
{
  "editor.formatOnSave": true,
  "editor.defaultFormatter": "esbenp.prettier-vscode",
  "editor.codeActionsOnSave": {
    "source.fixAll.eslint": true
  },
  "eslint.validate": [
    "javascript",
    "typescript"
  ]
}
```

Bien évidemment, il vous faudra installer les extensions VSCode pour ESLint et Prettier pour profiter de cette configuration ;-)

Avantages de ces outils

Pour résumer, voici quelques avantages d'ESLint et Prettier lorsqu'ils sont bien utilisés dans votre projet :

- Intégration précoce :
 - L'adoption d'ESLint et Prettier dès le début du projet évite d'avoir à reformater une grande base de code ultérieurement.
- Simplicité d'approche :
 - Une configuration standard permet de démarrer rapidement, avec la possibilité d'adapter progressivement les règles selon les besoins spécifiques de votre équipe.
- Automatisation du processus :
 - La configuration de vos outils pour formater automatiquement le code (à la sauvegarde par exemple) augmente l'efficacité, réduit les efforts manuels, limite la charge mentale et vous fait donc gagner du temps.
- Clarté documentaire :
 - Une documentation précise des choix de configuration assure que tous les membres de l'équipe comprennent les règles en place et travaillent de manière cohérente.

Ces pratiques conduisent à une base de code plus propre, plus cohérente et plus facile à maintenir, que vous soyez sur un projet individuel ou en groupe.

La gestion de version avec Git

Git est un logiciel de versioning créé en 2005 par Linus Torvalds (le créateur de Linux).

Un logiciel de versioning, aussi appelé logiciel de gestion de version, est un logiciel qui permet de conserver un historique des modifications effectuées sur un projet afin de pouvoir rapidement identifier les changements effectués et de revenir à une ancienne version en cas de problème.

Parmi les logiciels de gestion de versions, Git est le leader incontesté et il est donc indispensable pour tout développeur de savoir l'utiliser !

Pour faire simple, Git permet de coordonner le travail entre plusieurs personnes en conservant un historique des changements effectués sur des fichiers : il permet à différentes versions d'un même fichier de coexister. Les développeurs travaillant avec Git ont donc accès à l'historique des modifications pour l'intégralité du projet et peuvent ainsi savoir quels changements ont été fait par rapport à leur version des fichiers, qui a fait ces changements, et ainsi suivre l'ensemble du cycle de vie du projet...

Le lien du site officiel est : <https://git-scm.com/docs>

Modèle de gestion de version

Git est construit autour d'un modèle distribué : le code source du projet est toujours hébergé sur un serveur distant mais chaque utilisateur est invité à télécharger et à héberger l'intégralité du code source sur sa propre machine.

Ce modèle repose sur deux principes :

- Simplicité / flexibilité du travail :
 - Comme chaque utilisateur peut héberger le code complet du projet, il n'y a pas d'obligation d'être constamment connecté au serveur central et il est donc possible de travailler sur sa propre machine
- Sécurité :
 - Comme chaque utilisateur possède le code complet d'un projet, on peut utiliser la copie du projet d'un utilisateur comme back-up en cas de corruption du serveur central (dans la limite de ce qui aura été récupéré sur sa machine bien évidemment).

Et gitHub dans tout ça ?

Dans le langage des systèmes de versioning, la copie de l'intégralité des fichiers d'un projet et de leur version sur le serveur central est appelé un dépôt ou « repository » (« repo » en abrégé).

GitHub est un service en ligne d'hébergement de dépôts Git. C'est le plus grand hébergeur de dépôts Git du monde, même si d'autres hébergeurs de dépôts existent comme GitLab.

Installation de Git

Il est possible d'utiliser différents types d'interfaces pour installer et pour utiliser Git. Ici, nous allons utiliser la console et donc un langage en lignes de commande plutôt qu'une interface graphique (cette méthode est la seule permettant d'avoir accès à **toutes** les commandes Git).

Dans un premier temps, il faut donc installer Git sur votre poste. Si vous êtes sur une distribution linux, vous pouvez l'installer à partir des dépôts en utilisant une commande du genre :

```
# sudo apt install git
```

L'installation peut également être faite sans passer par le gestionnaire de paquet et en utilisant directement le site officiel : <https://git-scm.com/downloads> .

De même, si vous êtes sur Windows, un installeur est disponible depuis <https://git-scm.com/downloads/win> par exemple (ou mieux : passez sous Linux ! Il n'est jamais trop tard ^_^). Un installeur macOS est également disponible.

Pour savoir si Git est installé, il suffit de taper `$ git --version` dans un terminal, et vous devriez normalement obtenir la version de Git installée.

Paramétrage de Git

Une fois Git installé, nous allons paramétrer le logiciel afin d'enregistrer certaines données pour ne pas avoir à les fournir à nouveau plus tard.

Dans notre cas nous allons renseigner un nom d'utilisateur et une adresse mail que Git devra utiliser ensuite.

La commande `git config` permet de voir et de modifier les variables de configuration qui contrôlent tous les aspects de comportement de Git.

Nous allons également passer une option `--global` à notre commande. Celle-ci va nous permettre d'indiquer à Git que le nom d'utilisateur et l'adresse mail renseignés doivent être utilisés globalement (c'est-à-dire pour tout projet Git). Nous allons donc taper les commandes suivantes :

```
$ git config --global user.name "monUserGit"
```

Remplacez « `monUserGit` » par votre nom d'utilisateur Git

```
$ git config --global user.email monemail@provider.com
```

Remplacez « `monemail@provider.com` » par votre adresse email

Pour vous assurer que vos informations ont bien été enregistrées, vous pouvez taper :

```
$ git config user.name et $ git config user.email
```

Les informations enregistrées devraient être renvoyées

Fonctionnement d'un dépôt

Maintenant que vous disposez de Git sur votre machine, l'idée est avant tout de bien comprendre comment fonctionne un dépôt.

Création d'un dépôt

Il existe deux manières de créer un dépôt Git. La première consiste à démarrer un nouveau dépôt local à partir du nom spécifié : `$ git init [nom-du-projet]`.

Ensuite, après avoir créé notre dépôt sur GitHub, il est nécessaire d'initialiser ce dépôt distant avec un premier commit :

```
$ echo "# MON_PROJET" >> README.md
$ git add README.md
$ git commit -m "Initial commit"
$ git remote add origin [URL du dépôt distant]
$ git push -u origin master
```

De cette manière, vous avez donc créé votre dépôt d'abord en local, et ensuite, vous l'avez rattaché à votre dépôt distant.

Et si vous n'avez pas de dépôt distant, alors vous disposez de toutes les fonctionnalités de versionning de git mais bien évidemment uniquement en local sur votre machine.

La deuxième façon de créer un dépôt Git est de cloner un dépôt distant déjà existant (**et donc non vide**).

Le fait de cloner un dépôt correspond à télécharger tout le projet et l'ensemble de son historique de versions. Pour cela, vous utilisez la commande :

```
$ git clone [URL du dépôt distant]
```

Modifier un dépôt Git

Une fois que nous avons le dépôt en local, git se charge de détecter tout changement effectuer sur son contenu. Donc si vous souhaitez enregistrer une modification de fichier ou un ajout de fichier dans le dépôt Git, git le détectera et vous le signalera en utilisant la commande `$ git status` et il vous faudra utiliser deux commandes pour valider vos changements : `$ git add` et `$ git commit`.

Attention : le commit (la validation / l'enregistrement) d'un fichier se basera sur l'état de ce fichier au moment du git add. La commande `git commit` vient donc toujours après `git add`.

Exemples d'utilisation :

```
$ git add [chemin vers mon fichier]
```

```
$ git commit -m "message du commit"
```

Si vous souhaitez annuler les changements d'un fichier et le ramener à sa dernière version connue (au sens dernier commit) alors vous utiliserez la commande :

```
$ git reset HEAD [nom du fichier]
```

Tout comme la commande `git add`, il existe une commande qui supprime un fichier du répertoire de travail et de l'index :

```
$ git rm [nom du fichier].
```

Attention : le fait de "simplement" supprimer un fichier de votre système ne le retire pas de votre dépôt ! Vous devez donc passer par la commande `git rm`

Enfin, pour envoyer vos commits vers le dépôt distant, il vous faut utiliser la commande

```
$ git push
```

La dernière commande basique concerne l'action inverse, c'est à dire mettre à jour votre dépôt local en fonction de ce qui est présent sur votre dépôt distant (impératif pour un travail avec plusieurs développeurs par exemple) :

```
$ git pull
```

Exclure du suivi de version

Il est possible de préciser à Git d'exclure des fichiers et chemins de votre répertoire, par exemple contenant des fichiers temporaires, des modules, des variables sensibles, ...

Cette configuration se fait par l'intermédiaire d'un fichier textuel `.gitignore` présent à la racine de votre projet et contenant par exemple :

```
❖ .gitignore
1  node_modules
2  # Keep environment variables out of version control
3  .env
```

et vous pouvez utiliser la commande suivante pour lister tous les fichiers exclus :

```
$ git ls-files --other --ignored --exclude-standard
```

Nous avons donc vu ici les commandes qui sont impératives à connaître lorsque vous utilisez git. Bien évidemment il en existe d'autres très utiles que nous allons aborder ensemble.

Liste des commandes courantes

Configuration

Lien vers la doc : <https://git-scm.com/docs/git-config/fr>

Nous l'avons déjà vu plus haut, la commande `git config` permet de configurer les informations de l'utilisateur pour tous les dépôts locaux :

```
$ git config --global user.name "[nom]"
```

Définit le nom que vous voulez associer à toutes vos opérations de commit

```
$ git config --global user.email "[adresse email]"
```

Définit l'email que vous voulez associer à toutes vos opérations de commit

```
$ git config --global color.ui auto
```

Active la colorisation de la sortie en ligne de commande

Le répertoire Git est le répertoire central de tout projet géré et constitue la zone commune à tous les participants permettant de régler l'intégralité du contrôle des versions. Votre première étape avec Git consiste donc à créer un tel répertoire principal ou à le cloner (sous la forme d'une copie de travail).

Créer des dépôts

De même, ces commandes ont déjà été abordées pour démarrer un nouveau dépôt ou en obtenir un depuis une URL existante :

```
$ git init [nom-du-projet]
```

Permet de créer un dépôt local à partir du nom spécifié

<https://git-scm.com/docs/git-init/fr>

```
$ git clone [url]
```

Permet de télécharger un projet et tout son historique de versions

<https://git-scm.com/docs/git-clone/fr>

Effectuer des changements

Ces commandes commencent à vous apporter des informations plus précises sur les modifications en cours sur votre dépôt et permettent d'effectuer une opération de commit :

```
$ git status
```

Liste tous les nouveaux fichiers et les fichiers modifiés à commiter

<https://git-scm.com/docs/git-status/fr>

```
$ git diff
```

Montre les modifications de fichier qui ne sont pas encore indexées

<https://git-scm.com/docs/git-diff/fr>

```
$ git diff --staged
```

Montre les différences de fichier entre la version indexée et la dernière version

<https://git-scm.com/docs/git-diff/fr>

```
$ git add [fichier]
```

Ajoute un instantané du fichier, en préparation pour le suivi de version

<https://git-scm.com/docs/git-add/fr>

```
$ git rm [fichier]
```

Supprime le fichier du répertoire de travail et met à jour l'index

<https://git-scm.com/docs/git-rm/fr>

```
$ git commit -m "[message descriptif]"
```

Enregistre des instantanés de fichiers de façon permanente dans l'historique des versions

<https://git-scm.com/docs/git-commit/fr>

Grouper des changements avec les branches

Il est possible de nommer une série de commits et de combiner les résultats de travaux terminés en utilisant les branches :

```
$ git branch
```

Liste toutes les branches locales dans le dépôt courant

<https://git-scm.com/docs/git-branch/fr>

```
$ git branch [nom-de-branche]
```

Crée une nouvelle branche

<https://git-scm.com/docs/git-branch/fr>

```
$ git checkout [nom-de-branche]
```

Bascule sur la branche spécifiée et met à jour le répertoire de travail

<https://git-scm.com/docs/git-checkout/fr>

```
$ git merge [nom-de-branche]
```

Combine dans la branche courante l'historique de la branche spécifiée

<https://git-scm.com/docs/git-merge/fr>

```
$ git branch -d [nom-de-branche]
```

Supprime la branche spécifiée

<https://git-scm.com/docs/git-branch/fr>

Synchroniser les changements

Référencer un dépôt distant et synchroniser l'historique des versions avec le dépôt local

```
$ git fetch [nom-de-depot]
```

Récupère tout l'historique du dépôt nommé

<https://git-scm.com/docs/git-fetch/fr>

```
$ git merge [nom-de-depot]/[branche]
```

Fusionne la branche du dépôt dans la branche locale courante

<https://git-scm.com/docs/git-merge/fr>

```
$ git push [alias] [branche]
```

Envoie tous les commits de la branche locale vers le dépôt distant

<https://git-scm.com/docs/git-push/fr>

```
$ git pull
```

Récupère tout l'historique du dépôt nommé et incorpore les modifications

<https://git-scm.com/docs/git-pull/fr>

Changement au niveau des noms de fichiers

Il est également possible de déplacer et/ou supprimer des fichiers sous suivi de version :

```
$ git rm [fichier]
```

Supprime le fichier du répertoire de travail et met à jour l'index

<https://git-scm.com/docs/git-rm/fr>

```
$ git rm --cached [fichier]
```

Supprime le fichier du système de suivi de version mais le préserve localement

<https://git-scm.com/docs/git-rm/fr>

```
$ git mv [fichier-nom] [fichier-nouveau-nom]
```

Renomme le fichier et prépare le changement pour un commit

<https://git-scm.com/docs/git-mv/fr>

Enregistrer des fragments

Cette pratique permet notamment de mettre en suspens des modifications non finies pour y revenir plus tard. C'est ce qu'on appelle "Remiser son travail". Toutes ces commandes sont détaillées dans la documentation <https://git-scm.com/docs/git-stash/fr>

```
$ git stash
```

Enregistre de manière temporaire tous les fichiers sous suivi de version qui ont été modifiés

```
$ git stash list
```

Liste toutes les remises

```
$ git stash pop
```

Applique une remise et la supprime immédiatement

```
$ git stash drop
```

Supprime la remise la plus récente

Note importante : Je vous invite à expérimenter tous ces comportements afin d'être certain de ce qu'ils entraînent AVANT de les utiliser sur un projet important !

Vérifier l'historique des versions

Les commandes suivantes vous permettent de suivre et d'inspecter l'évolution des fichiers du projet :

```
$ git log
```

Montre l'historique des versions pour la branche courante

<https://git-scm.com/docs/git-log/fr>

```
$ git log --follow [fichier]
```

Montre l'historique des versions, y compris les actions de renommage, pour le fichier spécifié

<https://git-scm.com/docs/git-log/fr>

```
$ git diff [premiere-branche]...[deuxieme-branche]
```

Montre les différences de contenu entre deux branches

<https://git-scm.com/docs/git-diff/fr>

```
$ git show [commit]
```

Montre les modifications de métadonnées et de contenu incluses dans le commit spécifié

<https://git-scm.com/docs/git-show/fr>

Refaire des commits

Enfin, vous pouvez revenir sur des éléments en cours et gérer l'historique des corrections. La documentation officielle de cette commande `git reset` est disponible sur <https://git-scm.com/docs/git-reset/fr>

```
$ git reset [commit]
```

Annule tous les commits après [commit], en conservant les modifications localement

<https://git-scm.com/docs/git-reset/fr>

```
$ git reset --hard [commit]
```

Supprime tout l'historique et les modifications effectuées après le commit spécifié

<https://git-scm.com/docs/git-show/fr>

Convention de commit et Husky

Dans le but de faciliter le travail collaboratif (même à très grande échelle), des conventions de développement et d'usages ont été mises en place, et c'est réellement un plus que de les appliquer ! Parmi elles, il y a notamment la convention d'usage pour vos commits, dont vous pouvez suivre le projet sur : <https://www.conventionalcommits.org/fr/v1.0.0/> .

Son but est de standardiser vos messages de commits dans un premier temps mais également de pouvoir automatiser les vérifications avec Git hooks.

L'outil Husky

Husky est un outil qui permet d'utiliser des Git hooks plus facilement. La page du projet est accessible sur <https://typicode.github.io/husky/>

Que sont les Git hooks ?

Les Git hooks sont des scripts qui s'exécutent avant ou après des événements Git comme commit, push, etc. Ils permettent d'automatiser des tâches comme :

- Vérifier le formatage du code avant un commit
- Exécuter des tests avant un push
- Valider la syntaxe des messages de commit

Installation de Husky

```
$ npm install --save-dev husky
$ npx husky init
```

La seconde commande permet de créer un script de pre-commit dans le répertoire .husky à la racine de votre projet et met à jour notamment votre package.json avec le script prepare. Bien évidemment, vous pourrez l'adapter par la suite.

/!\

Selon votre version, les scripts des Hooks de husky pourront se retrouver
soit directement à la racine du repertoire .husky,
soit dans un sous-repertoire .husky/_/

/!\

Création d'un hook pre-commit

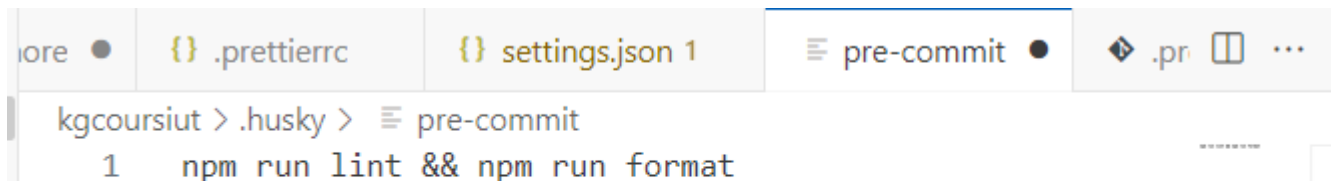
Le hook pre-commit s'exécute avant qu'un commit ne soit finalisé. C'est l'endroit idéal pour vérifier la qualité du code.

Si vous avez lancé le `npx husky init`, alors ce fichier a déjà été créé et peut contenir par exemple :



```
kgcoursiut > .husky > pre-commit
1 npm test
```

Etant donné que nous avons mis en place le linter et le formateur, il est possible d'ajuster ce pre-commit de la manière suivante :



```
kgcoursiut > .husky > pre-commit
1 npm run lint && npm run format
```

afin d'assurer la bonne exécution de ces scripts avant un commit.

De cette manière, vous avez commencé à automatiser des exécutions de scripts en lien avec vos commits, mais il vous appartient encore de les faire en respectant la convention.

Le “Commits Conventionnels”

Les "Conventional Commits" (ou commits conventionnels) constituent une spécification pour ajouter une signification sémantique aux messages de commit. Cette convention standardisée facilite notamment :

- La génération automatique de changelogs
- La détermination automatique de versions sémantiques (SemVer)
- La communication claire des changements aux coéquipiers et utilisateurs
- Le déclenchement de processus de build et de déploiement appropriés

Structure d'un Conventional Commit

Un message de commit conventionnel est de la structure suivante :

```
<type>[scope optional]: <description>

[optional body]

[optional footer(s)]
```

Ces différents éléments structurels permettant de communiquer sur ce qui est concerné par votre commit. Par exemple :

- fix: un commit de type fix corrige un bogue dans le code (cela est en corrélation avec PATCH en gestion sémantique de versions).
- feat: un commit de type feat introduit une nouvelle fonctionnalité dans le code (cela est en corrélation avec MINOR en gestion sémantique de versions).

Ou encore, un commit qui contient dans son footer le mot-clé BREAKING CHANGE:, ou dont le type/étendue est suffixé d'un !, introduit une rupture de compatibilité dans votre projet (ce qui est en corrélation avec MAJOR en gestion sémantique de versions). Un BREAKING CHANGE peut faire partie des commits de n'importe quel type.

Type

C'est le premier élément de votre commit. Le type définit la nature du changement et sa valeur fait partie des catégories suivantes :

- feat : Ajout d'une nouvelle fonctionnalité
- fix : Correction d'un bug
- docs : Modification de la documentation
- style : Changements qui n'affectent pas le sens du code (espaces, formatage, etc.)
- refactor : Changement de code qui ne corrige pas un bug et n'ajoute pas de fonctionnalité
- perf : Amélioration des performances
- test : Ajout ou correction de tests
- build : Changements qui affectent le système de build ou les dépendances externes
- ci : Changements dans la configuration CI ou les scripts
- chore : Autres changements qui ne modifient pas les fichiers src ou test
- revert : Annulation d'un commit précédent

Scope

Le scope est optionnel et indique la partie du code affectée par le changement (par exemple, auth, api, db).

Description

Une description **concise** du changement au présent. Ne pas capitaliser la première lettre et ne pas mettre de point à la fin.

Body

Le corps est optionnel et fournit des détails supplémentaires sur le changement. Il doit commencer par une ligne vide après la description.

Footer

Le footer est optionnel et peut contenir des références à des issues, des notes sur les changements non rétrocompatibles (breaking changes), etc.

Exemples de Conventional Commits

feat(auth): ajouter la fonction de réinitialisation de mot de passe

Cette fonctionnalité permet aux utilisateurs de réinitialiser leur mot de passe via un email de confirmation.

Closes #123

fix: corriger le bug d'authentification lors de la connexion avec Google

docs: mettre à jour la documentation de l'API REST

feat: ajouter la fonction de recherche avancée

BREAKING CHANGE: L'ancienne API de recherche n'est plus compatible.

Installation et configuration de commitlint

Pour vérifier que les messages de commit suivent la convention des Conventional Commits, nous pouvons utiliser commitlint, dont la page du projet est disponible sur <https://commitlint.js.org/>

```
$ npm install -D @commitlint/cli @commitlint/config-conventional
```

et une fois installé, vous pouvez créer un fichier de configuration nommé commitlint.config.js à la racine de votre projet :

```
export default { extends: ['@commitlint/config-conventional'] };
```

et ajouter ensuite un hook commit-msg pour valider le message de commit (l'usage du hook pre-commit n'est pas encore supporté au moment où j'écris ces lignes).

Créez donc un nouveau fichier commit-msg dans votre répertoire de hooks contenant

```
npx --no -- commitlint --edit $1
```

Avantages de cette approche

L'utilisation de Conventional Commits avec Husky présente plusieurs avantages :

- Histoire de projet plus lisible :
 - Les messages de commit standardisés facilitent la compréhension de l'historique du projet.
- Génération automatique de changelogs :
 - Les outils comme semantic-release peuvent générer automatiquement des notes de version basées sur les messages de commit.
- Versioning sémantique automatique :
 - Le type de changement (feat, fix, etc.) peut déterminer automatiquement la prochaine version.
- Qualité du code améliorée :
 - Les vérifications pré-commit garantissent que seul le code de qualité est commité.
- Réduction des erreurs :
 - Les vérifications automatiques réduisent le risque d'erreurs humaines.

En standardisant les messages de commit et en automatisant les vérifications, vous simplifiez donc la collaboration et améliorez ainsi la qualité globale de votre projet.

Intégration Continue / Déploiement Continu

Cette section concerne principalement la phase “finale” d’un développement mais il est cependant intéressant de la considérer très tôt dans la réalisation de vos projets.

Qu'est-ce que la CI/CD ?

La CI/CD (Intégration Continue/Déploiement Continu) est une méthodologie de développement logiciel qui vise à automatiser et à améliorer le processus de livraison de code. Cette approche permet aux équipes de développement de livrer des applications de haute qualité plus rapidement et de manière plus fiable.

Intégration Continue (CI)

L'intégration continue est la pratique qui consiste à intégrer fréquemment les modifications de code dans un dépôt partagé.

Chaque intégration est vérifiée par une construction automatisée (y compris les tests) pour détecter les erreurs d'intégration aussi rapidement que possible.

Cette centralisation permet d'assurer, au-delà des aspects de sauvegarde et restauration, que toutes les portions de codes réalisées soient traitées de la même manière.

Déploiement Continu (CD)

Le déploiement continu est une approche dans laquelle les modifications qui passent tous les tests automatisés sont automatiquement déployées en production. Le CD peut également signifier "Livraison Continue", où le processus automatisé s'arrête juste avant le déploiement en production, nécessitant une approbation manuelle pour la mise en production.

Cette politique est propre à chaque structure / chaque projet et permet de garder la main sur les timings de mise en production et ainsi assurer la continuité de service pour les utilisateurs.

Avantages de la CI/CD

Détection précoce des bugs

En intégrant fréquemment, les conflits et les bugs sont détectés tôt dans le cycle de développement, ce qui les rend beaucoup plus faciles à résoudre.

Feedback rapide

Les développeurs reçoivent un retour immédiat sur la qualité de leur code grâce aux tests automatisés et aux vérifications de qualité de code.

Réduction des risques

Les petites modifications sont plus faciles à déboguer que les grandes. En intégrant fréquemment, on réduit considérablement les risques liés aux déploiements.

Déploiements plus fiables

L'automatisation réduit les erreurs humaines qui peuvent survenir lors des déploiements manuels.

Livraison plus rapide

Le cycle de développement est accéléré grâce à l'automatisation des processus répétitifs, permettant aux équipes de se concentrer sur le développement de nouvelles fonctionnalités.

Pipeline CI/CD typique

Un pipeline CI/CD complet inclut généralement les étapes suivantes:

- Déclenchement :
 - Le pipeline est déclenché par un événement, comme un push sur le dépôt Git.
- Analyse de code :
 - Vérification de la qualité du code avec des outils comme ESLint et Prettier.
- Construction :
 - Compilation du code source.
- Tests :
 - Exécution des tests unitaires, d'intégration et de bout en bout.
- Analyse de sécurité :
 - Recherche de vulnérabilités dans le code et les dépendances.
- Déploiement de test :
 - Déploiement dans un environnement de test ou de staging.
- Tests de validation :
 - Vérification que l'application fonctionne correctement dans l'environnement de test.
- Déploiement en production :
 - Déploiement automatique ou manuel en environnement de production.
- Monitoring :
 - Surveillance continue de l'application en production.