Git

- git bisect
- GitHub Copilot □□
- Git town □□

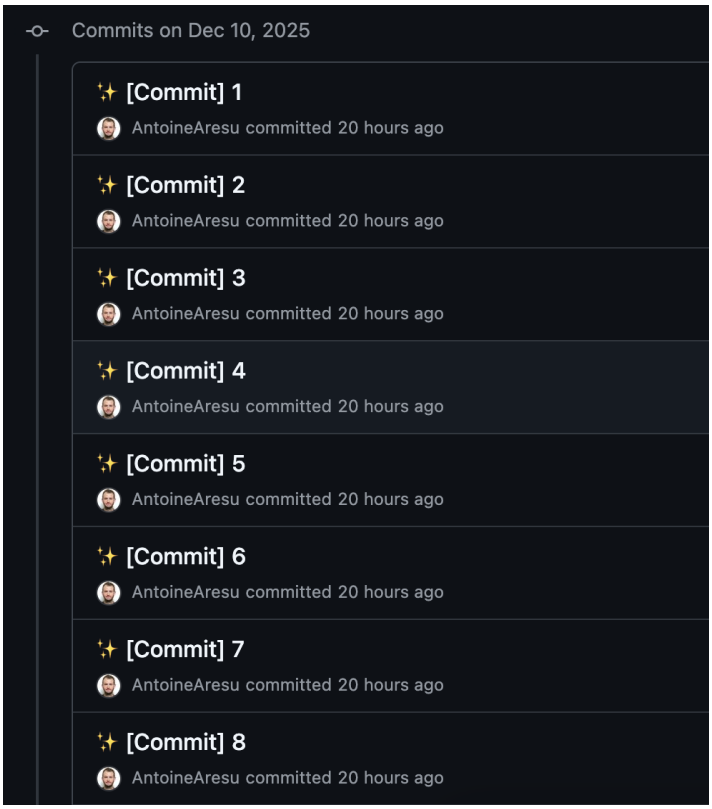
git bisect

À quoi ça sert ?

- Permet de mettre en évidence un commit qui a induit un changement dans un historique de commits
- Ce changement peut correspondre à un bug, à un test qui casse... tout changement d'état

Comment ça marche

- Prenons l'exemple le plus simple : j'ai développé une feature, et j'ai induis un bug
- Je sais qu'au premier commit (commit 1), je n'avais pas de bug
- Je sais qu'au dernier commit (commit 8), j'ai un bug



Au lieu de checkout sur chaque commit pour constater si le bug existe (ce qui impliquerait de vérifier 8 commits), je peux utiliser git bisect :

- Je vais devoir flaguer un commit "good" (sans bug) **-> commit 1**
- Et un commit "bad" (avec bug) **-> commit 8**
- Ensuite, au lieu de vérifier commit par commit, git bisect va séparer la feature en 2 : il va checkout sur le commit 4 pour savoir si il est "good" ou "bad".

- Et ainsi de suite ! Si le commit 4 est "good", le bug se trouve forcément entre le commit 4 et le commit 8, donc il va me faire vérifier le commit 6 etc... Jusqu'à pouvoir identifier le commit qui a induit une erreur

En pratique :

Entrer en mode bisection :

```
git bisect start
```

Flag initial d'un commit "good" et "bad", en indiquant la référence du commit (optionnel si je me trouve sur ce commit :

```
git bisect good cebcb8b352b80af4dfa92e554ac5a608dffa7be1 #(SHA commit 1)
git bisect bad #le SHA peut être optionnel si je suis positionné sur le commit 8
```

Flag des commits proposés durant la bisection :

```
git bisect good
#OU
git bisect bad
```

Fin de la session bisection :

```
git bisect reset
```

Ceci correspond au mode manuel de git bisect, mais il est aussi possible de l'automatiser. Pour cela, reproduire les 2 premières étapes mentionnées plus haut. Puis lancer :

```
git bisect run <monScript>
```

Le script peut être n'importe quoi : lancer un test unitaire, exécuter un script php, bash etc...
La bissection sera alors automatique, et git bisect pourra flaguer les commits automatiquement en fonction du retour du script, et donc mettre en évidence le commit problématique seul

GitHub Copilot □□

> GitHub Copilot est un assistant de programmation basé sur l'IA (développé par GitHub et OpenAI) qui analyse ton code en temps réel et propose des suggestions pour coder plus vite. Il est intégré directement dans PhpStorm via un plugin officiel.

Prerequis

- Un compte GitHub actif avec un abonnement GitHub Copilot (individuel, Business ou Enterprise)
- PhpStorm version 2023.1 ou superieure (compatible avec le plugin officiel)

Installation

1. Aller dans **File > Settings > Plugins** (ou `Ctrl+Alt+S` puis « Plugins »)
2. Chercher **GitHub Copilot** dans l'onglet *Marketplace*
3. Cliquer sur **Install** puis redémarrer PhpStorm
4. Se connecter à GitHub via la notification qui apparaît en bas à droite

---

Complétion de code automatique

Dès que tu commences à taper, Copilot propose des suggestions en gris directement dans l'éditeur.

Raccourci	Action
`Tab`	Accepter la suggestion
`Échap`	Rejeter la suggestion
`Alt + ]` / `Alt + [`	Suggestion suivante / précédente
`Ctrl + Entrée`	Ouvrir le panneau de toutes les suggestions
`Alt + Shift + \`	Déclencher une suggestion manuellement

Astuce : écrire un commentaire avant ton code améliore considérablement la pertinence des suggestions. Exemple :

```
// Fonction qui valide un email et retourne true si valide
function validateEmail(string $email): bool {
```

Copilot Chat

Le panneau Chat permet d'interagir avec son code en langage naturel. Il est accessible via **Tools > GitHub Copilot > Open Chat**, ou en cliquant sur l'icône ci-dessous dédié dans la barre latérale droite de PhpStorm.

Les modes du Chat

Un menu déroulant en bas de la fenêtre permet de choisir entre quatre modes.

Le mode **Ask** est le plus simple : on pose une question, Copilot répond sans modifier le code. Idéal pour comprendre un bout de code, explorer une approche ou débloquer une erreur.

Le mode **Edit** permet de décrire une modification en langage naturel sur un ou plusieurs fichiers. Copilot applique les changements et affiche un diff à valider. Utile quand on sait ce qu'on veut mais qu'on ne veut pas l'écrire soi-même.

Le mode **Plan** génère un plan détaillé des étapes à suivre pour accomplir une tâche, sans toucher au code. Pratique pour cadrer une fonctionnalité complexe avant de se lancer.

Le mode **Agent** est le plus autonome : Copilot analyse le projet, choisit les fichiers à modifier et itère jusqu'à compléter la tâche. À réserver aux tâches complexes bien définies — il consomme plus de requêtes premium que les autres modes.

Choix du modèle

Le sélecteur de modèle est disponible en haut de la fenêtre Chat. Plusieurs modèles sont disponibles selon l'abonnement : **GPT-4o** est le modèle par défaut (bon équilibre qualité/vitesse), **GPT-4.1** est plus performant sur les tâches complexes, **Claude Sonnet / Opus** excellent en compréhension de contexte et refactoring, **Gemini 2.5 Pro** est particulièrement efficace sur les données.

En cas de doute, le mode **Auto** laisse Copilot choisir le modèle le plus adapté à chaque requête.

Commandes slash

Dans le Chat, des commandes slash permettent de déclencher des actions rapidement sur le code sélectionné :

- `/explain` — explique le code sélectionné

- `/fix` — corrige les problèmes détectés
- `/tests` — génère des tests unitaires
- `/doc` — génère la documentation PHPDoc
- `/optimize` — propose des optimisations

Générer un message de commit avec Copilot

Dans la fenêtre de commit de PhpStorm, l'icône Copilot située au-dessus du champ de message permet de générer automatiquement un message en analysant le diff des fichiers stagés. Il ne reste plus qu'à relire, ajuster si besoin, et valider.

Personnaliser le format des messages générés

Par défaut, Copilot génère des messages dans son propre format. Il est possible de lui fournir des instructions globales pour qu'il respecte les conventions du projet.

Pour cela, aller dans **Settings > Tools > GitHub Copilot > Customizations**, puis dans la section **Git Commit Instructions**, cliquer sur **Global**. Cela génère et ouvre le fichier `global-git-commit-instructions.md` directement dans l'éditeur, prêt à être renseigné.

Ci-dessous les instructions configurées pour que Copilot génère des messages de commit conformes aux conventions de l'équipe tech, intégrant l'utilisation de Gitmoji.

````
# GitHub Copilot Instructions

## Commit Messages

Always generate commit messages following the format below and using the appropriate emoji for the type of change being made. This helps maintain a clear and consistent commit history, making it easier for everyone to understand the purpose of each commit at a glance.

### Format
```
<emoji> [<scope>] <short description>
[optional body]

[optional footer]
```

### Types et emojis associés (liste officielle Gitmoji)
````

Emoji	Code	Type à utiliser	Description
□	`:sparkles:`	`feat`	Introduce new features
□□	`:bug:`	`fix`	Fix a bug
□□	`:ambulance:`	`fix`	Critical hotfix
□□	`:adhesive_bandage:`	`fix`	Simple fix for a non-critical issue
♻	`:recycle:`	`refactor`	Refactor code
□□	`:art:`	`style`	Improve structure / format of the code
□□	`:lipstick:`	`style`	Add or update the UI and style files
□□	`:memo:`	`docs`	Add or update documentation
□□	`:bulb:`	`docs`	Add or update comments in source code
□	`:white_check_mark:`	`test`	Add, update, or pass tests
□□	`:test_tube:`	`test`	Add a failing test
□□	`:wrench:`	`chore`	Add or update configuration files
□□	`:hammer:`	`chore`	Add or update development scripts
⬆	`:arrow_up:`	`chore`	Upgrade dependencies
⬇	`:arrow_down:`	`chore`	Downgrade dependencies
□	`:heavy_plus_sign:`	`chore`	Add a dependency
□	`:heavy_minus_sign:`	`chore`	Remove a dependency
□□	`:pushpin:`	`chore`	Pin dependencies to specific versions
□□	`:package:`	`chore`	Add or update compiled files or packages
□□	`:see_no_evil:`	`chore`	Add or update a .gitignore file
⚡	`:zap:`	`perf`	Improve performance
□□	`:construction_worker:`	`ci`	Add or update CI build system
□□	`:green_heart:`	`ci`	Fix CI Build
□	`:rewind:`	`revert`	Revert changes
□□	`:fire:`	`chore`	Remove code or files
□□	`:rocket:`	`chore`	Deploy stuff
□□	`:tada:`	`chore`	Begin a project
□□	`:lock:`	`fix`	Fix security or privacy issues
□□	`:closed_lock_with_key:`	`chore`	Add or update secrets
□□	`:bookmark:`	`chore`	Release / Version tags
□□	`:rotating_light:`	`fix`	Fix compiler / linter warnings
□□	`:construction:`	`chore`	Work in progress
□□	`:chart_with_upwards_trend:`	`feat`	Add or update analytics or track code
□□	`:globe_with_meridians:`	`feat`	Internationalization and localization
✏	`:pencil2:`	`fix`	Fix typos
□	`:rewind:`	`revert`	Revert changes

□□	`:twisted_rightwards_arrows:`	`chore`	Merge branches
□□	`:alien:`	`fix`	Update code due to external API changes
□□	`:truck:`	`refactor`	Move or rename resources
□□	`:page_facing_up:`	`chore`	Add or update license
□□	`:boom:`	`feat`	Introduce breaking changes
□□	`:bento:`	`chore`	Add or update assets
♿	`:wheelchair:`	`feat`	Improve accessibility
□□	`:speech_balloon:`	`feat`	Add or update text and literals
□□	`:card_file_box:`	`feat`	Perform database related changes
□□	`:loud_sound:`	`chore`	Add or update logs
□□	`:mute:`	`chore`	Remove logs
□□	`:children_crossing:`	`feat`	Improve user experience / usability
□□	`:building_construction:`	`refactor`	Make architectural changes
□□	`:iphone:`	`feat`	Work on responsive design
□□	`:goal_net:`	`fix`	Catch errors
□□	`:dizzy:`	`feat`	Add or update animations and transitions
□□	`:wastebasket:`	`refactor`	Deprecate code that needs to be cleaned up
□□	`:passport_control:`	`feat`	Work on authorization, roles and permissions
□□	`:monocle_face:`	`chore`	Data exploration/inspection
⚰	`:coffin:`	`refactor`	Remove dead code
□□	`:necktie:`	`feat`	Add or update business logic
□□	`:stethoscope:`	`chore`	Add or update healthcheck
□□	`:bricks:`	`chore`	Infrastructure related changes
□□□□	`:technologist:`	`chore`	Improve developer experience
□□	`:safety_vest:`	`feat`	Add or update code related to validation
□□	`:label:`	`chore`	Add or update types
□□	`:seedling:`	`chore`	Add or update seed files
□□	`:triangular_flag_on_post:`	`feat`	Add, update, or remove feature flags
□□	`:mag:`	`chore`	Improve SEO
⚗	`:alembic:`	`chore`	Perform experiments
□□	`:camera_flash:`	`test`	Add or update snapshots
□□	`:thread:`	`feat`	Add or update code related to multithreading
□□	`:money_with_wings:`	`feat`	Add sponsorships or money related infrastructure
□□	`:t-rex:`	`refactor`	Code that adds backwards compatibility
✈	`:airplane:`	`feat`	Improve offline support
□□	`:clown_face:`	`test`	Mock things
□□	`:egg:`	`chore`	Add or update an easter egg
□□	`:busts_in_silhouette:`	`chore`	Add or update contributor(s)

Règles
- L'**emoji** est obligatoire et doit correspondre au type de modification
- Le **scope** est obligatoire et doit être entre crochets, indiquant la partie du code affectée (ex: `[API]`, `[UI]`, `[Auth]`)
- La **description** doit être courte (50 caractères max), en minuscules, sans point final
- Le **body** est optionnel, séparé par une ligne vide, pour expliquer le "pourquoi"
- Utiliser `BREAKING CHANGE:` dans le footer si la modification est non rétrocompatible

Exemples
```
 [Home] add new account number display
 [API] handle null response on user fetch
♻ [Service] extract payment logic into PaymentService
 [Deps] update symfony to 6.4.1
 [Auth] add unit tests for JWT expiration
 [API] remove deprecated v1 endpoints

BREAKING CHANGE: v1 endpoints have been removed
```

<emoji> [<scope>] <short description>

Bonnes pratiques

- **Nommer ses variables clairement** : Copilot s'appuie sur le contexte sémantique
- **Toujours relire le code généré** : les suggestions sont un point de départ, pas une vérité
- **Ne jamais écrire de secrets dans le code** lors de l'utilisation de Copilot (clés API, mots de passe...)

Liens utiles : Documentation officielle | Plugin JetBrains

Git town □□

> Git town est un outil pour simplifier l'utilisation de Git CLI, permettant de garder facilement toutes ses branches à jour, même lorsqu'on a une arborescence de branches complexe.

Créer une branche

Le première commande a connaître est celle qui permet de créer une nouvelle branche : **hack**

```
git town hack 'feature/my-feature'
```

Git town va d'abord fetch les derniers changements du repository, puis mettre à jour la branche main, avant de créer la branche de feature à partir de main. De cette façon, on est sûr d'avoir une branche de feature à jour avec la dernière version de main.

Publier sa branche

Une fois que j'ai fini et commité le travail sur ma branche, je peux publier celle-ci et ouvrir une pull request grâce à une simple commande : **propose**

```
git town propose
```

Avant de pousser la branche locale, git town va mettre à jour la branche parente locale (main par exemple) et rebase la branche de feature sur celle-ci. Ensuite, il push la branche et ouvre directement un onglet de navigateur sur la page de création d'une PR. Plus qu'à mettre à jour le titre/la description et valider !

Créer une branche enfant

En ma plaçant sur la branche dont je veux créer un enfant, je peux le faire simplement avec **append**

```
git town append 'feature/children-feature'
```

Encore une fois, git town va mettre à jour toutes les branches parentes avant de créer la nouvelle.

Mettre à jour sa branche

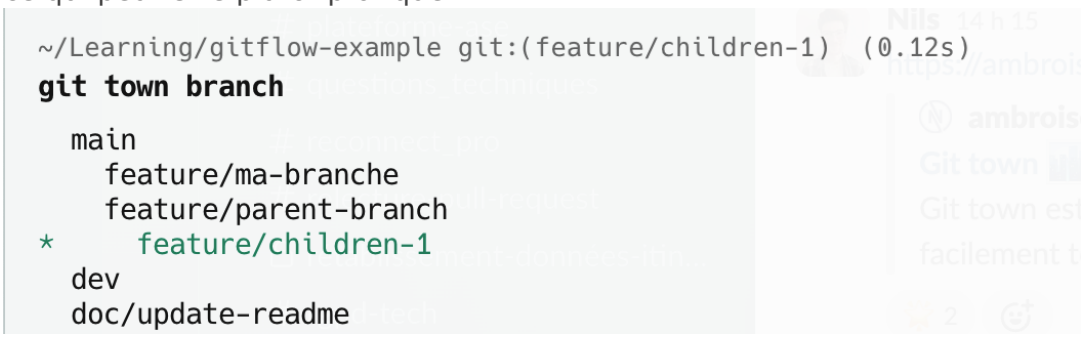
Si j'ai besoin de pull ou push des changements de ma branche, je peux le faire très simplement avec **sync**

```
git town sync
```

Git town va mettre à jour **toutes** les branches parentes, puis la branche actuelle, puis pousser la branche locale. S'il y a des modifications non commitées, il effectue un stash avant toute chose, puis un pop en dernière étape, de façon à reprendre facilement de là où on en était.

Afficher les branches locales

La commande **branch** permet de lister les branches locales et d'afficher ça de façon hiérarchique, ce qui peut être plutôt pratique :



Configurer les branches pérennes (perennial branches)

Par défaut, git town considère uniquement `main` comme branche permanente. Si l'on souhaite ajouter la branche `dev` comme branche permanente afin de pouvoir utiliser git town de la même façon, il faut l'indiquer explicitement pour que git town la traite comme une racine valide et ne tente pas de la rebase sur `main` :

```
git town config perennial-branches dev
```

Cette commande est à exécuter une fois par repo. Après ça, git town sait que `dev` est une branche permanente, et `sync` la mettra à jour sans chercher à la rattacher à une branche parente.

Changer la branche parente (set-parent)

Il peut arriver qu'une branche ait été créée depuis `main` mais qu'on veuille finalement la rattacher à `dev` , ou simplement que l'on ait créé une branche de façon classique et que l'on souhaite la rattacher pour pouvoir `sync` ensuite. Dans ce cas :

```
git town set-parent dev
```

Git town va alors considérer `dev` comme branche parente, ce qui a deux effets concrets : `sync` mettra d'abord `dev` à jour avant de mettre à jour la branche courante, et `propose` ouvrira la PR vers `dev` plutôt que vers `main`.

Autre informations

Il faut penser à configurer la stratégie désirée pour la mise à jour des branches (merge ou rebase). Il faudra indiquer quelle est la branche main sur le repo lors de la première utilisation.

En cas de conflit lors d'une opération, git town s'interrompt. Il suffit alors de résoudre les conflits et de taper la commande **git town continue**

Liens utiles : Documentation officielle | Github