

GIT.

Code toujours accessible - suivi des modifications -
expérimentations - relectures - partage

PARIS, LE 12 AVRIL 2018

Antoine Cheron

✉ antoine.cheron@fabernovel.com

🔗 <https://antoinecheron.github.io>

Inria
inventeurs du monde numérique

// FABERNOVEL
TECHNOLOGIES

Introduction.



01

GITHUB

Dans ce document, je considérerai que tu utilises **GitHub** comme dépôt pour ton code et que tu sais y créer un **repository**.

02

CRÉES UN COMPTE !

Si tu n'en as pas, crées-toi un compte sur <https://github.com>.

03

LE TERMINAL TU AIMERAS

Tu vas voir que le terminal c'est super cool !

01 //

Sauvegarder son code.

Point de départ.

1. Crées un dossier dans lequel tu placeras le code de ton projet.

```
foo@barPC ~ mkdir nomDuProjet & cd nomDuProjet
```

2. Initialises un git dans ce répertoire.

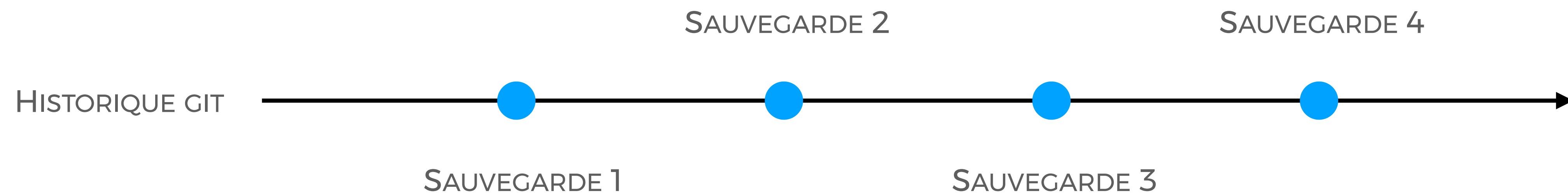
```
foo@barPC ~/nomDuProjet git init
```

→ CELA CRÉE UN GIT DANS LE DOSSIER.
TOUT L'HISTORIQUE SERA STOCKÉ
DANS LE DOSSIER .GIT

Fonctionnement.

Git est une sorte d'**historique** de ton code, dont la **sauvegarde** est à faire **manuellement**, et qui est capable de garder **plusieurs versions** de tes documents.

Git est un outil qui voit ton code comme un répertoire de documents.

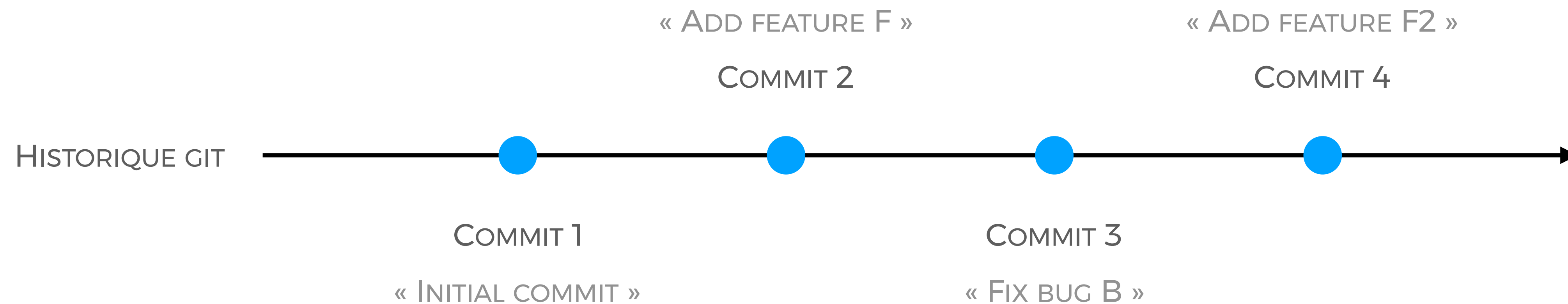


Fonctionnement.

Une sauvegarde est un **commit**.

Il est une bonne pratique d'associer **un message à chaque commit**.

Le schéma précédent devient ainsi :



Faire une sauvegarde.

La sauvegarde se fait en deux étapes.

1. Choisir les fichiers à sauvegarder

```
foo@barPC ~/nomDuProjet git add path/folder # tous les fichiers du dossier  
foo@barPC ~/nomDuProjet git add path/filename.f # seul ce fichier  
foo@barPC ~/nomDuProjet git add . # tous les fichiers modifiés depuis la dernière sauvegarde
```

2. Faire la sauvegarde

```
foo@barPC ~/nomDuProjet git commit -m « message du commit » # avec les guillemets !
```

Point sur la sauvegarde.

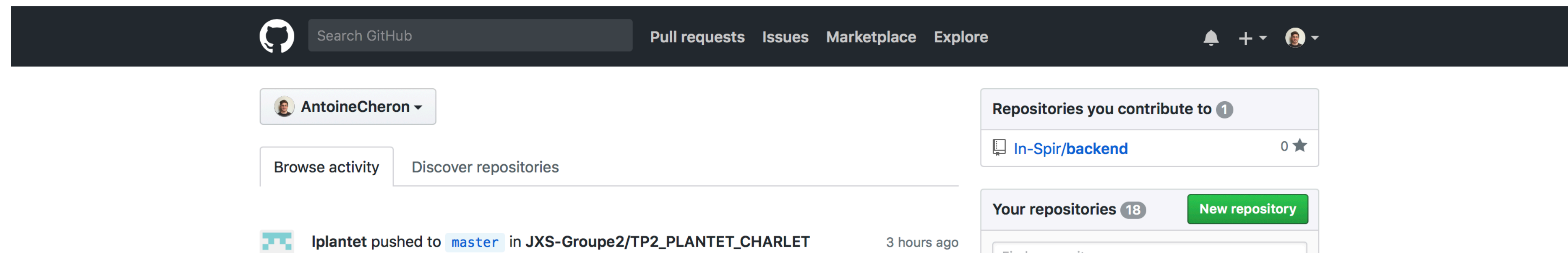
Tout ce que vous avez fait jusqu'à maintenant est resté sur votre ordinateur.

Regardons désormais comment envoyer ce qui est sur votre ordinateur sur GitHub, qui agira comme un Cloud gratuit pour votre code.

Sauvegarder sur GitHub.

La première fois...

1. Vas sur Github et crée un repository grâce au **GROS** bouton vert 😊



2. Fais ce qui est inscrit sur la page, à savoir:

```
# Commiter, tout ça tout ça (comme sur la slide 6 quoi...)
foo@barPC ~/nomDuProjet git remote add origin https://github.com/user/repo.git
foo@barPC ~/nomDuProjet git push -u origin master
```

Sauvegarder sur GitHub.

Les fois suivantes → beaucoup plus simple

1. Fais ton commit

```
foo@barPC ~/nomDuProjet git add ... # tu connais maintenant (cf. slide 6)  
foo@barPC ~/nomDuProjet git commit -m « message du commit » # tu connais aussi
```

2. Push ! (et ça se dit **pouche**, **pas peuche**, merci 😊)

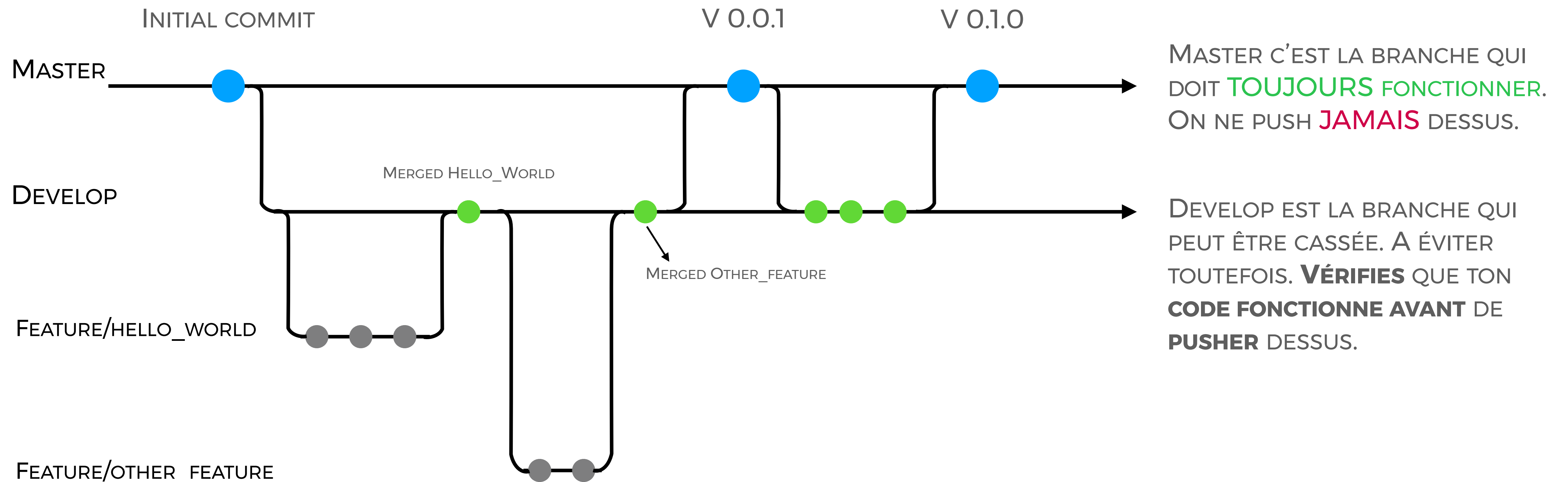
```
foo@barPC ~/nomDuProjet git push
```

02 //

Les branches - les bases.

Bonnes pratiques.

Celles que nous suivons en entreprise.



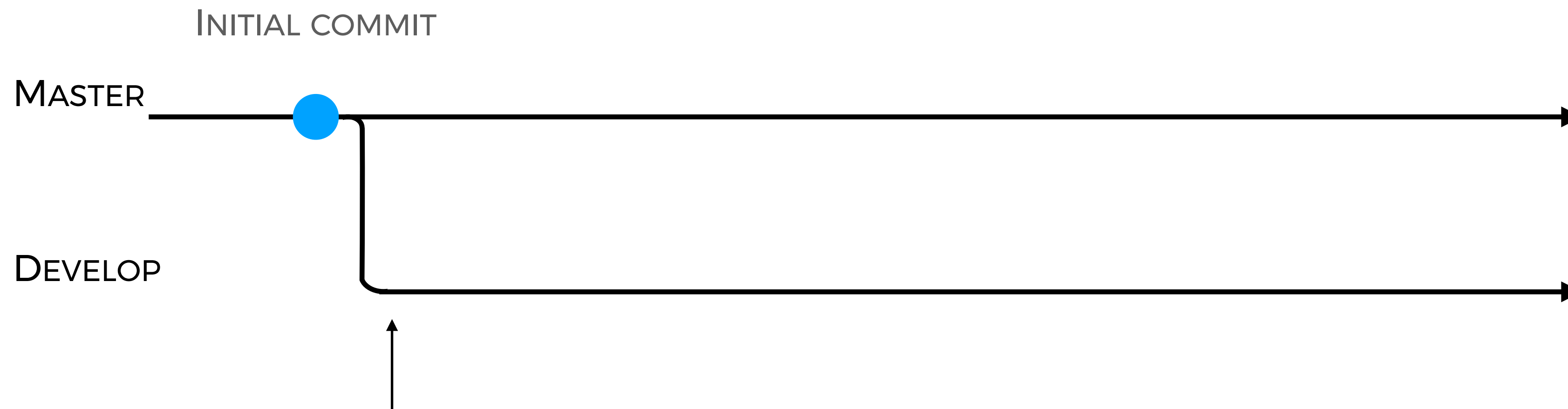
« Master, Develop, feature/hello_world et feature/other_feature sont ce qu'on appelle des branches »

Créer une branche.

Dans cet exemple, nous partirons de la branche master et créerons la branche develop.

```
foo@barPC ~/nomDuProjet <master> git checkout -b develop
```

Cette commande crée la branche et te places automatiquement dessus.



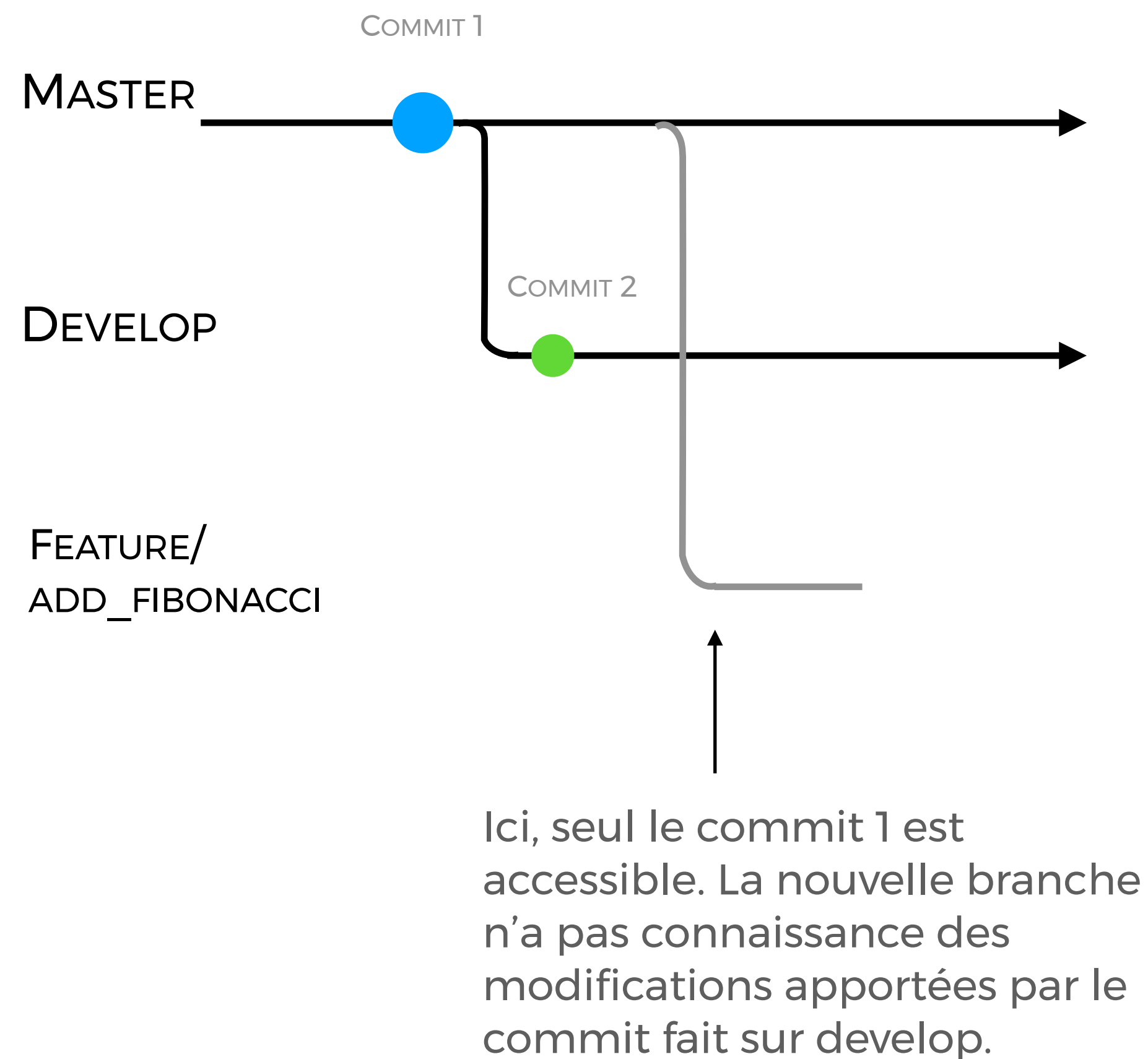
Tu te trouves désormais ici

Warning: ta branche se crée à partir du contenu de la branche sur laquelle tu étais lorsque tu as fais la commande « `git branch -b` »

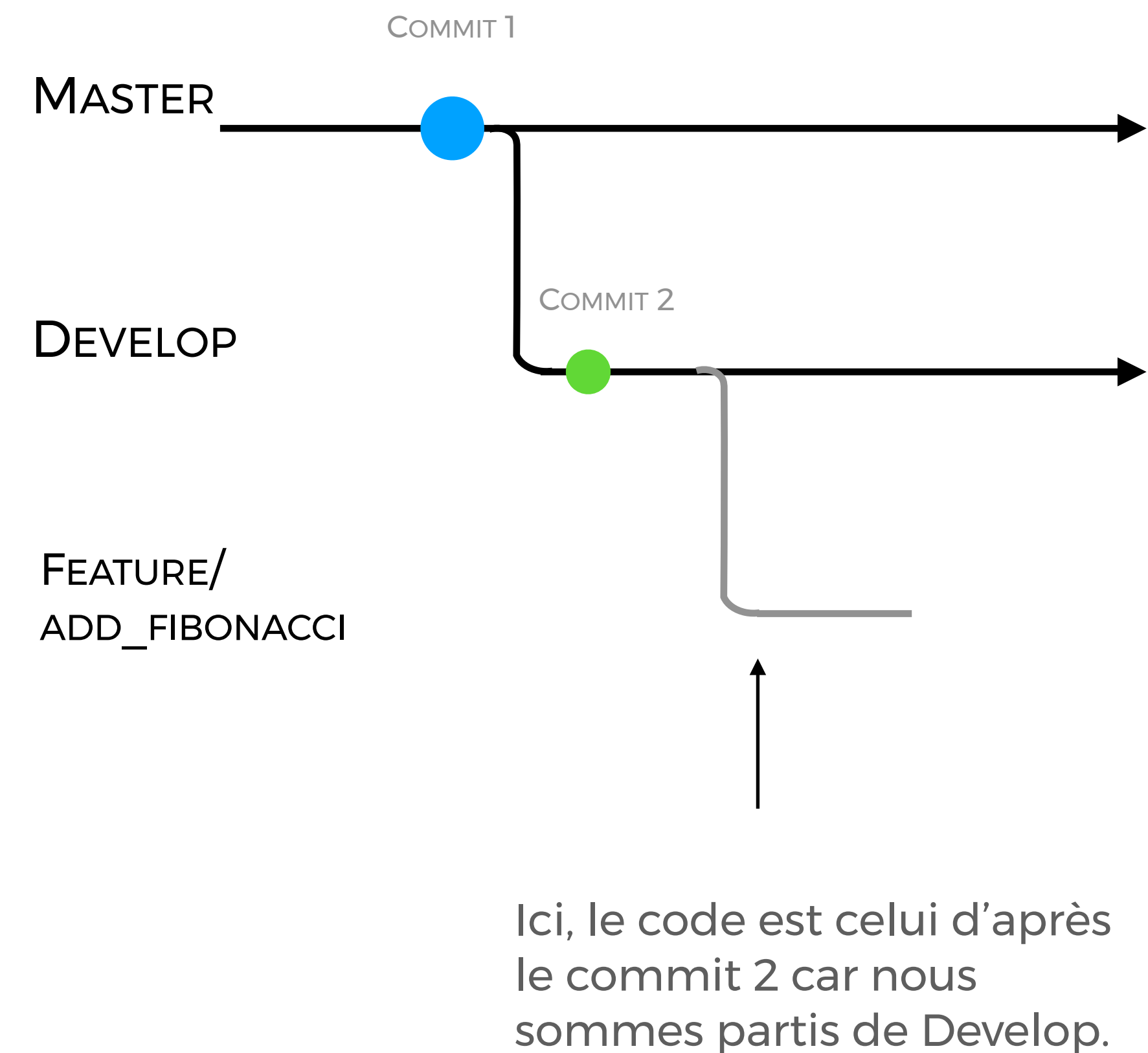
Illustration de la slide précédente.

Tu veux créer une branche « `feature/add_fibonacci` ». Prenons 2 exemples :

« `GIT BRANCH -B` » DEPUIS MASTER



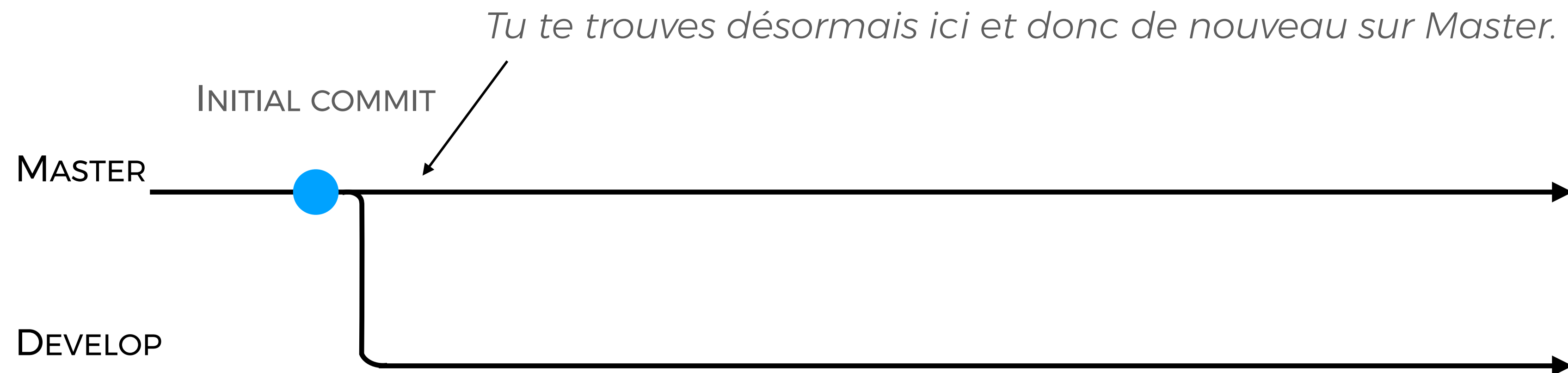
« `GIT BRANCH -B` » DEPUIS DEVELOP



Changer de branche.

Maintenant que nous sommes sur develop, retournons sur master.

```
foo@barPC ~/nomDuProjet <master> git checkout master
```



Se placer sur un commit en particulier.

Dans les slides précédentes, tu as découvert la commande `git checkout` pour créer et changer de branche. Saches que `checkout` sert aussi à se placer sur un commit bien précis grâce à l'identifiant unique du commit.

La branche sur laquelle nous travaillons est la suivante :



Disons que nous avons besoin de retourner au Commit « a ». La commande à taper est :

```
foo@barPC ~/nomDuProjet <branche-courante> git checkout 82b9aff0
```

Bonnes pratiques, on reprend.

Développes une fonctionnalité par branche.

- Ni Master ni Develop ne sont des branches pour faire son développement.
- Comme montré sur le schema, Master ne sert qu'à faire les releases.
- Develop est la branche sur laquelle sont ajoutées les fonctionnalités entre 2 releases.
- Pour chaque ajout/modification du code du projet crée une nouvelle branche à partir de Develop.
- Utilise ces préfixes pour nommer tes branches :
 - Feature → développement d'une nouvelle fonctionnalité. *Example : feature/person_api*
 - Fix → correction d'un bug. *Example : fix/person_always_null*
 - Misc → autres, par exemple un refactoring. *Example : misc/refactor_class_person*

Pusher le code de sa nouvelle branche.

Maintenant que tu sais comment te servir des branches, tu peux y taper ton code et le pusher.

Profites de la branche que tu as créée à partir de develop et ajoutes-y un petit morceau de code. Ensuite, commit le et disons que tu souhaites le pusher pour qu'un de tes collègues y aie accès. Si tu tapes simplement `git push`, voici ce que ton terminal te diras :

```
foo@barPC ~/nomDuProjet <misc/example> git push
fatal: The current branch misc/example has no upstream branch.
To push the current branch and set the remote as upstream, use

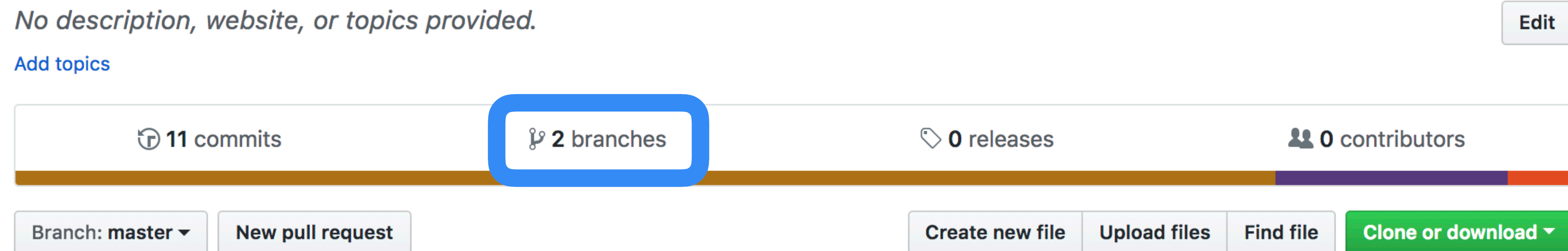
git push --set-upstream origin misc/example
```

Et bien en fait c'est très bien, c'est exactement ce que nous voulons. Copies-colles la commande et fait Entrée.

Envoyer ses ajouts/modifications sur Develop ou Master.

Plaçons-nous dans le contexte où tu as créé une branche depuis develop, y as ajouté une nouvelle fonctionnalité, qui fonctionne parfaitement car tu l'as bien testée et que tu l'as pushée sur GitHub. Tu es désormais prêt à rajouter cette fonctionnalité sur Develop. **Comment faire ?**

1. Vas sur les branches de ton repo sur GitHub (voir screen ci-dessous)





Envoyer ses ajouts/modifications sur Develop ou Master.

2. Fais une pull-request. Il s'agit d'une demande pour que tous les commits que tu as fait sur ta branche soit ajoutés à la branche que tu choisiras comme destination, ici develop.

Default branch

<code>master</code> Updated 3 months ago by AntoineCheron	Default	Change default branch
---	---------	-----------------------

Your branches


<code>gh-pages</code> Updated 4 months ago by Antoine Cheron	✓	11 3	 New pull request 
--	---	--------	--

Envoyer ses ajouts/modifications sur Develop ou Master.


3. Choisis la branche de destination, les relecteurs et explique ce que tu as fait dans cette branche, ainsi que la manière dont tes relecteurs pourront la tester.

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

 base: master ← compare: hors-serie-1

✓ Able to merge. These branches can be automatically merged.



ajoute correction hors serie 1

Write

Preview

AA ▾ B i “ <> 🔗 ⋮ ⋮ ⋮ ✓

↩ @ ★

Leave a comment

Attach files by dragging & dropping, [selecting them](#), or pasting from the clipboard.

M+ Styling with Markdown is supported

Create pull request

Reviewers

No reviews

Assignees

No one—assign yourself

Labels

None yet

Projects

None yet

Milestone

No milestone

Envoyer ses ajouts/modifications sur Develop ou Master.

4. Tes relecteurs pourront trouver ta PR (Pull-Request) dans l'onglet qui porte ce nom sur ton repository. Ils pourront ajouter des commentaires sur ton code en cliquant sur le petit + à gauche de la ligne.

Profites des PR pour que tes collègues te fasse des retours sur ton code afin d'en améliorer sa qualité. Encourages-les à venir te challenger sur tes noms de variables, de fonctions et sur l'efficacité et la complexité des tes algorithmes, l'architecture de ton code ainsi que sa lisibilité globale !!!

5. Prends en compte leurs commentaires en faisant les modifications nécessaires dans ton code. Fais ensuite un nouveau commit que tu pourras nommer « treat comments » par exemple et push de nouveau ton code. La PR sera automatiquement mise à jour.

→ le processus est le même pour pousser le code de Develop vers Master, fais simplement une PR.

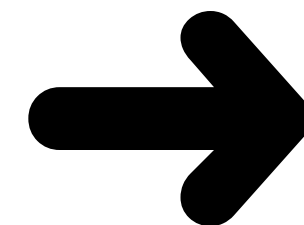
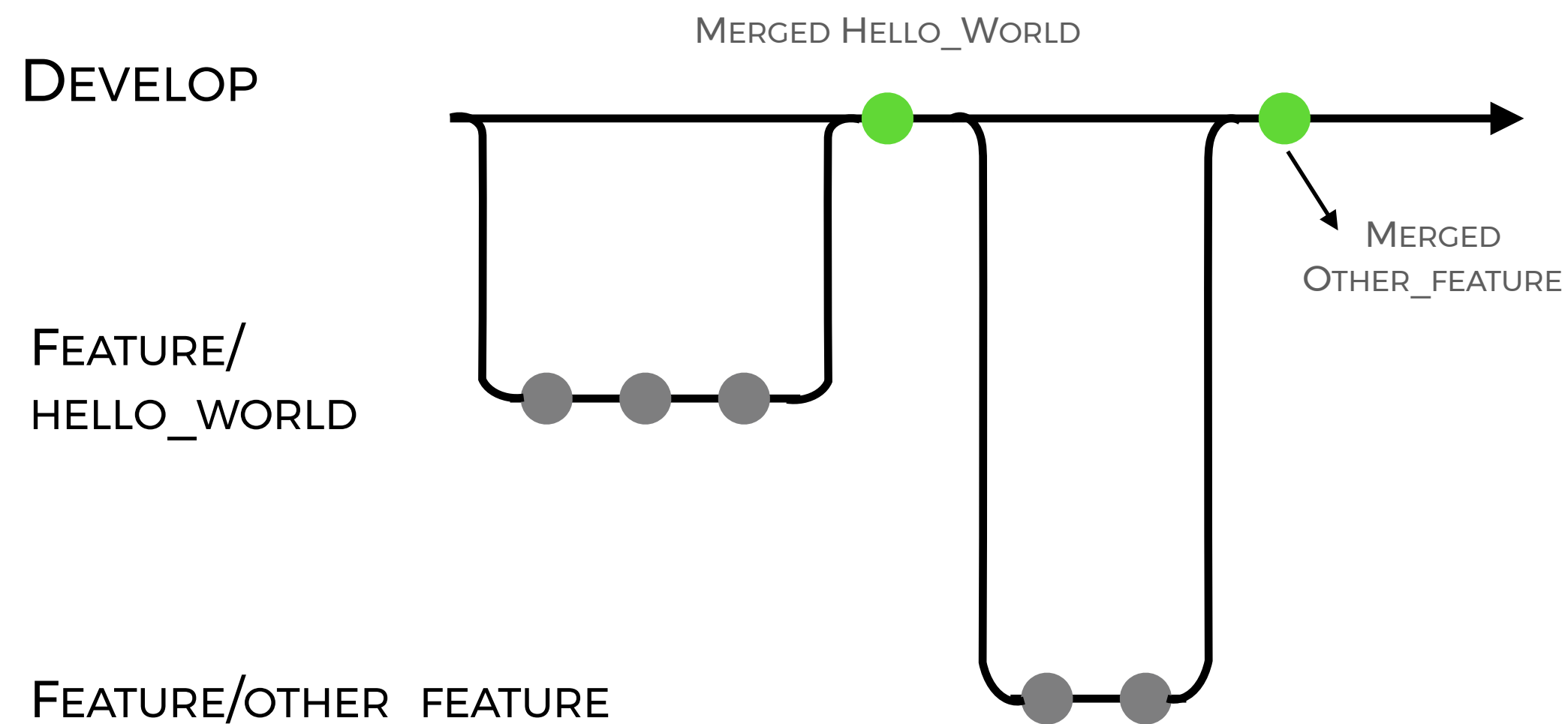
03 //

Les branches - avancé.

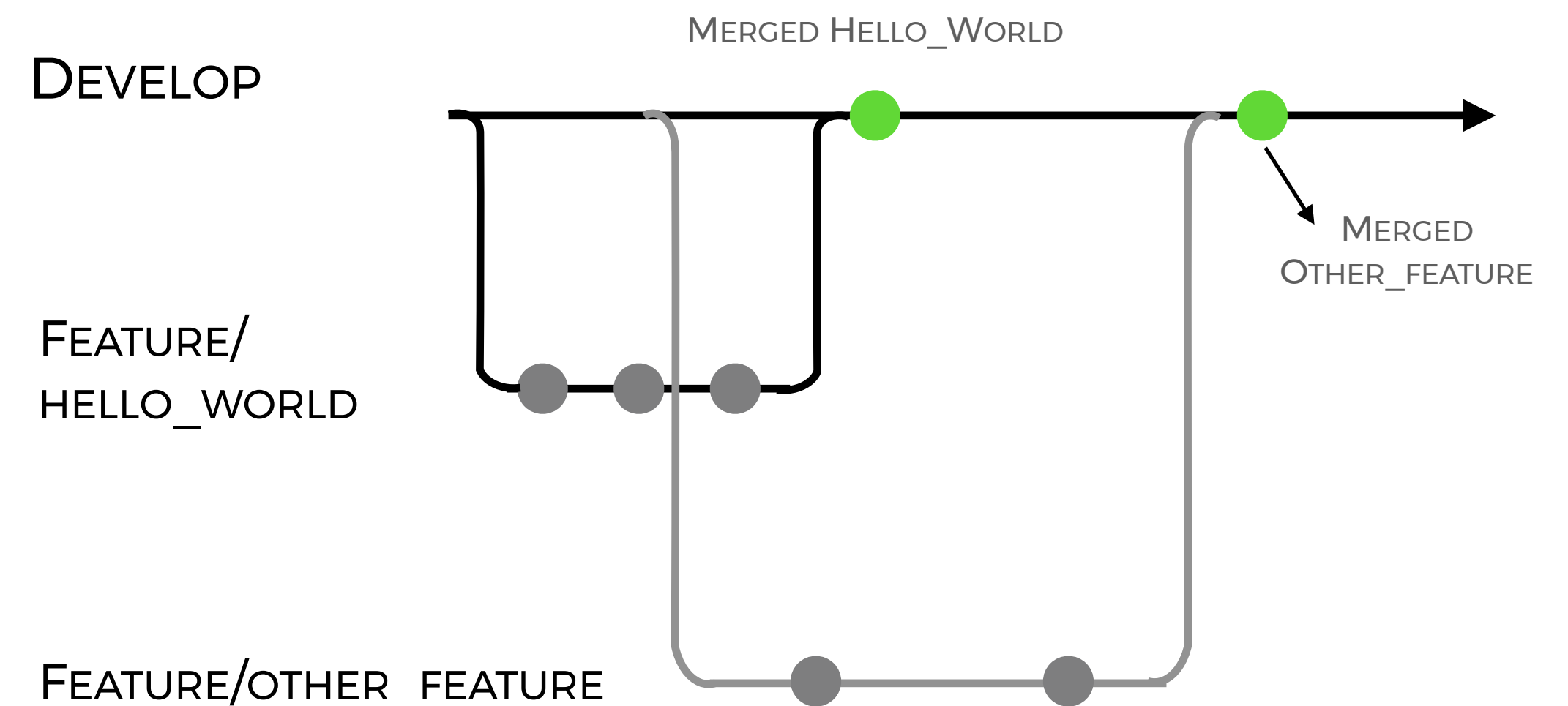
Se mettre à jour avant de pusher.

Maintenant que tu as les bases concernant les branches, penchons nous sur un cas très commun.

CA C'ÉTAIT LE CAS SIMPLIFIÉ :



LA RÉALITÉ C'EST PLUTÔT COMME ÇA :



Au moment de faire la PR de la branche feature/other_feature, GitHub te montreras que ta branche a 1 commit de retard sur Develop. **Il faut te mettre à jour !**

Se mettre à jour avant de pusher.

Comment faire ?

Reprenons notre terminal.

```
# Commençons à l'étape où il faut commiter le travail effectué
foo@barPC ~/nomDuProjet <feature/other> git add .
foo@barPC ~/nomDuProjet <feature/other> git commit -m « message explicite »
# Ici nous sommes en retard, il faut se mettre à jour
foo@barPC ~/nomDuProjet <feature/other> git fetch # met à jour l'historique de tout
votre git (sans télécharger les modifications, uniquement les id et dates des nouveaux commits et
nouvelles branches.
foo@barPC ~/nomDuProjet <feature/other> git rebase origin/develop # te met à jour
sur la branche develop.
# Ici nous sommes à jour, il est temps de pusher
foo@barPC ~/nomDuProjet <feature/other> git push
```

Point d'attention.

Pourquoi avoir fait `git rebase origin/develop` et pas `git rebase develop` ?

Parce qu'origin/develop est la branche de ton remote, c.a.d GitHub, tandis que develop est ta branche locale. Or, tu as la garantie que la branche remote est à jour, ce qui n'est pas le cas pour ta branche locale. Pour t'en assurer, il faudrait que tu retournes sur develop, la mette à jour avec un `git pull` et retournes sur ta branche pour faire le `rebase`, ce qui, tu le consentiras, est beaucoup moins pratique.

Se mettre à jour avant de pusher.

La gestion des conflits

Au cours de la procédure décrite à la slide précédente, il est possible que des conflits se produisent. Cela mettra en pause la commande `git rebase`.

Les conflits apparaissent lorsque tu as modifié un fichier à la même ligne que la personne qui a fait les commits qui s'ajoutent à ta branche lors du rebase.

Pour résoudre ces conflits, ouvre les fichiers concernés, choisis la version du code que tu veux garder, tu l'identifieras facilement, les lignes conflictuelles sont entourées de chevrons. Une fois cela fait, reprends ton terminal et fais :

```
# Pour continuer le rebase après avoir résolu les conflits
foo@barPC ~/nomDuProjet <feature/other> git add .
foo@barPC ~/nomDuProjet <feature/other> git rebase --continue
```


04 // Résumé.

Git, GitHub et les grands principes

1. Git permet de versionner son code en faisant des sauvegardes régulières et d'expérimenter de nouvelles fonctionnalités sans casser le code qui fonctionne grâce aux branches.
2. GitHub est un service en ligne qui permet de stocker ses répertoires Git dans le cloud, ainsi que les partager avec d'autres utilisateurs. Cela ouvre le champ des possibles à la relecture du code, à la création d'issues, et à l'intégration avec des outils d'automatisation.
3. Utilises toujours une branche Master pour les releases, une branche Develop pour le développement de nouvelles fonctionnalités entre 2 releases et une nouvelle branche chaque fois que tu souhaites modifier le code.
4. Fais des PR, fais les relire et relis celles de tes camarades. Toutes les remarques et questions sont les bienvenues dans ce contexte.

Exemple : le processus de création des branches, mise à jour et sauvegarde.

Cette fois, aucun commentaire, tu dois comprendre les commandes ci-dessous, sinon relis le slide.

```
foo@barPC ~/nomDuProjet <master> git checkout develop
foo@barPC ~/nomDuProjet <develop> git fetch & git pull
foo@barPC ~/nomDuProjet <develop> git checkout -b feature/a
# Développement...
foo@barPC ~/nomDuProjet <feature/a> git add .
foo@barPC ~/nomDuProjet <feature/a> git commit -m « description feature a »
foo@barPC ~/nomDuProjet <feature/a> git fetch
foo@barPC ~/nomDuProjet <feature/a> git rebase origin/develop
# Gérer les conflits si nécessaire
foo@barPC ~/nomDuProjet <feature/a> git push --set-upstream origin feature/a
# Faire sa Pull-Request sur GitHub
```

Nota Bene.

Pour simplifier votre organisation, un seul développeur doit développer une fonctionnalité de bout en bout. Ne travaillez jamais à 2 en même temps sur une même fonctionnalité et sur 2 ordinateurs à la fois. Si vous souhaitez travailler à 2, faites du pair-programming, c'est à dire que vous êtes 2 sur un seul ordinateur.

Pour aller plus loin...

1. Guide de style Git : <https://github.com/pierreroth64/git-style-guide>
2. La commande `cherry-pick` : <https://git-scm.com/docs/git-cherry-pick>
3. Regrouper plusieurs commits en un seul : <https://gist.github.com/patik/b8a9dc5cd356f9f6f980>

Amuses-toi !