GitHub - Premiers pas

60 minutes Normal

DataScientest • com

GitHub

II. Premiers pas sur la plateforme

Une fois le compte créé et Git installé, il est temps d'aborder la pratique.

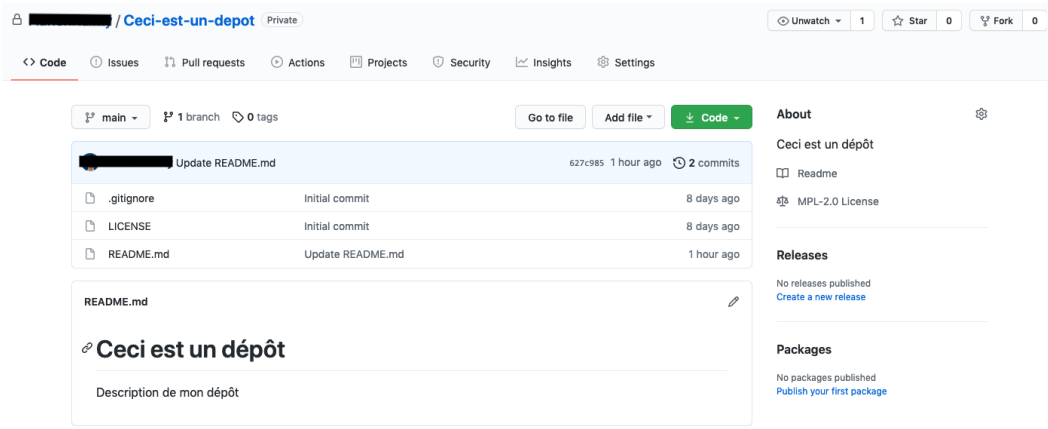
Les dépôts / repositories

La collaboration sur GitHub s'effectue principalement en termes de dépôts ou repositories en anglais, repo pour faire court. Si l'on souhaite commencer un projet et le partager, on peut **créer** notre propre dépôt, ou bien participer à un projet en cours sur un dépôt **déjà existant**. Dans les deux cas, il faut faire **le lien** entre un dépôt **distant** hébergé sur **GitHub**, et un dépôt **local** hébergé sur votre machine.

Pour créer un nouveau dépôt, il suffit de cliquer sur le bouton **New** accessible sur la barre latérale gauche de la page d'accueil, puis de le nommer et de le paramétrer à votre guise.

> Depuis la page d'accueil, créez un nouveau dépôt privé dont vous choisirez le nom. Ajoutez-y un fichier README.md qui servira de description pour votre dépôt.

Votre dépôt devrait maintenant apparaître dans la barre latérale gauche. Accédez à celui-ci en cliquant dessus.



Pour faire le lien entre notre dépôt distant et notre machine, on va **cloner** celui-ci sur notre machine. Pour ce faire, il suffit d'utiliser la fonction `git clone` suivie de l'URL de notre dépôt distant. Vous pouvez récupérer l'URL correspondant en cliquant sur **Code**.

La commande à effectuer en local sera donc de la forme :

```
1  git clone https://github.com/UserName/Mon-depot.git
2
```

> Vous serez sûrement amené à vous identifier, il s'agit des mêmes identifiants et mot de passe que ceux que vous avez renseignés en vous inscrivant.

Add / Commit / Push

Maintenant que notre dépôt est instancié, nous allons chercher à l'alimenter. Procédons par étapes.

Dans un premier temps, nous allons alimenter notre dépôt local. Pour le moment, nous resterons sur la branche **main**, la branche principale de notre dépôt.

> Dans votre dépôt local, créez un nouveau fichier `fichier_1.txt` dans lequel vous écrirez une phrase de votre choix telle que 'Ceci est mon premier fichier'.

Show / Hide solution

```
# On se place dans le bon dossier
cd Mon-depot

# On créer notre fichier texte
nano fichier_1.txt

```

Nous avons donc ajouté un fichier à notre dépôt local, mais il n'est pas encore apparu sur notre dépôt distant. Voyons l'état d'avancement de notre dépôt à l'aide de `git status`.

La fonction nous retourne le message suivant :

```
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committ

  fichier_1.txt

nothing added to commit but untracked files present (use "git
```

Notre fichier est dit "untracked" et apparaît en rouge dans le message. Il n'est pas encore "commit", c'est à dire que la modification faite à notre dépôt n'est pas encore enregistrée.

Pour l'ajouter à la liste des modifications à enregistrer, on utilisera la fonction `git add`.

> Ajoutez le fichier `fichier_1.txt` à la liste des fichiers à prendre en compte dans le prochain `commit`, puis afficher à nouveau le statut de notre dépôt.

Show / Hide solution

```
# Ajout du fichier
git add fichier_1.txt
# Statut du dépôt
git status

```

Cette fois-ci, notre fichier n'est plus "untracked", nous pouvons donc passer au `commit` pour enregistrer la modification effectuée.

Dans une optique d'exploiter GitHub pour du travail collaboratif, il est important d'accompagner un commit d'un message **court et clair** permettant de rendre lisible pour tous les participants au projets, l'historique des modifications faites. Pour ajouter un message, il suffit d'ajouter l'argument `-m` à la fonction `git commit`. Par exemple :

```
git commit -m 'Ajout de fichier_1.txt'

```

> Enregistrez les modifications faites au dépôt en précisant de façon clair et synthétique ces dernières dans le message accompagnant votre commit.

À chaque action faite, il est naturel de vérifier le statut de notre dépôt, encore une fois analysons ce que nous renvoie `git status`.

```
git status
------
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Tant que l'on ne pousse pas nos modifications sur le serveur distant, il est toujours possible **d'annuler** ou de **revenir** à un commit antérieur. Pour retracer l'historique de nos commits sur la branche principale, nous pouvons utiliser la commande `git log`, elle retournera l'ensemble des commits, accompagnés de leurs identifiants et des messages associés.

Pour revenir à un commit donné, on utilisera la fonction `git reset`.

Si l'on souhaite **annuler le précédent commit sans effacer les modifications** faites en local lors de celui-ci, on utilisera l'argument `--soft`:

```
git reset --soft HEAD~1
# L'argument HEAD~1 précise que l'on souhaite revenir au commit
# l'actuel étant (designé par HEAD)

```

Si l'on souhaite **annuler le commit ainsi que les modifications faites durant celui-ci**, on utilisera l'argument `--hard`:

```
git reset --hard HEAD~1

```

Pour communiquer nos modifications au dépôt distant, la commande à effectuer est `git push` comme mentionnée dans le message obtenue après le `git status`.

> Pousser les modifications faites sur le dépôt local vers votre dépôt distant.
>
> Vérifiez que les modifications ont bien été appliquées en vous rendant sur la page web GitHub de votre dépôt.

Show / Hide solution

```
# Sur le dossier du dépôt
git push

```

En actualisant la page web GitHub de notre dépôt, le fichier `fichier_1.txt` a bien été ajouté.

Fetch / Pull

Dans le cadre d'un projet collaboratif, il faut s'attendre à ce que d'autres participants effectuent des modifications et les incorporent au dépôt distant. Dès lors, le dépôt local sur lequel on travaille est voué à se désynchroniser par rapport aux modifications faites par d'autres. Pour récupérer ces changements, on utilisera les commandes `git fetch` et `git pull`.

À l'instar de la partie précédente, récupérer des changements effectués sur le dépôt distant se fait en deux étapes :

1. **Vérifier s'il y a eu des changements apportés au dépôt distant à l'aide de la commande `git fetch`.**
   Celle-ci téléchargera les métadonnées du dépôt sans pour autant télécharger les modifications faites. Elle permet donc de vérifier si des changements peuvent être importés sur le dépôt local.
2. **Importer les changements effectués sur le dépôt distant vers le dépôt local à l'aide de la commande `git pull`.**

Les branches

Avec ce que l'on vient de voir, on se rend rapidement compte que pour ne pas avoir de conflits de versions entre les différents participants d'un projet, il est préférable que chacun puisse travailler sur sa propre branche avant de penser à modifier la branche principale.

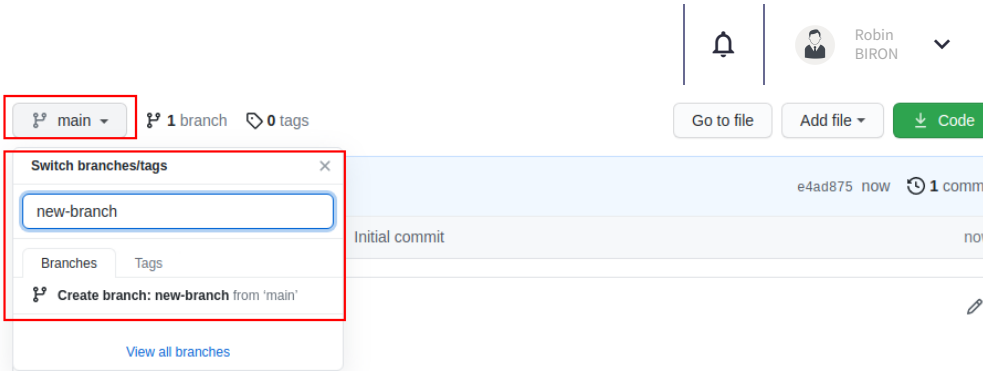
Pour créer une branche, on peut le faire de deux façons différentes :

- **Depuis le site de GitHub, sur la page "< > Code" du dépôt.**

Il suffit de cliquer sur le bouton déroulant associés aux branches dans la partie supérieure gauche du dépôt.

Par défaut vous serez sur la branche `main` (ou `master`). `main` étant le nouveau nom de la branche par défaut, `master` est l'ancien nom.

Pour créer une nouvelle branche, vous n'aurez qu'à écrire le nom de votre nouvelle branche dans le champ dédié :



Une fois que cette étape est faite, pour synchroniser votre dépôt local avec le dépôt distant, il suffit de télécharger les métadonnées associées avec `git fetch`, puis pour se placer sur cette nouvelle branche sur le dépôt local, utilisez la commande suivante :

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

```

- **Depuis le terminal en local**

La méthode conseillée, car il est souvent préférable de partir du local, vers le distant plutôt que l'inverse. Pour instancier une nouvelle branche, assurons-nous **avant tout** que notre dépôt local est synchronisé avec notre dépôt distant à l'aide des commandes `git fetch` et `git pull`. Puis dans un second temps, on effectuera les commandes suivantes :

```
# Pour créer la branche
git branch [nom-de-la-branche]

# Pour se placer sur celle-ci
git checkout [nom-de-la-branche]

```

Ensuite, il faut pousser ces informations sur le dépôt distant à l'aide de la commande suivante :

```
git push -u origin [nom-de-la-branche]

```

Où `-u` est l'argument `--set-upstream` permettant à notre branche d'être traçable, `origin` correspond par défaut au nom de notre dépôt sur lequel on pousse nos modifications.

> À l'aide de la méthode de votre choix, créez une nouvelle branche dans laquelle vous ajouterez un nouveau fichier texte `fichier_2.txt`.
>
> Vérifiez que vos changements ont bien été synchronisés sur votre dépôt distant en consultant le site une fois fait.

Show / Hide solution

```

# On se place sur la nouvelle branche
git checkout une_branche

# Une fois fait on pousse cette nouvelle branche sur notre dé
git push -u origin une_branche

# Notre branche faite, on peut créer notre nouveau fichier te
nano fichier_2.txt

# On le remplit avec du texte au choix, puis on l'ajoute au p
git add fichier_2.txt

# On commit puis on push
git commit -m "ajout de fichier_2.txt sur une nouvelle branch

# Toujours sur la branche
git push

```

Nous venons donc de créer une nouvelle branche et d'y ajouter un nouveau fichier. Lorsque l'on regarde sur la page du dépôt distant, ce fichier existe dans notre branche de développement mais n'existe pas sur notre branche principale `main`.

Supposons que nous ayons fait les ajouts nécessaires sur notre branche et que nous voulons maintenant partager notre travail avec les autres collaborateurs, nous devons faire basculer les modifications apportées à notre branche sur la branche principale du dépôt. Pour cela, on fusionnera

cela on peut utiliser la commande `git diff` qui s'utilise comme suit :

```
git diff [nom-de-la-première-branche] [nom-de-la-seconde-branche
```

Cette commande listera tous les fichiers, dossiers et autres modifications qui différent entre les deux branches. Une fois que l'on s'est assuré que ces modifications sont les bonnes, on peut passer à la fusion des branches à l'aide de la fonction `git merge` **depuis la branche principale** qui sera à la réception de cette fusion. La fonction s'utilise comme suit :

```
git merge [nom-de-la-branche-à-merge]
```

> En utilisant la fonction `git checkout`, placez-vous dans la branche principale.
>
> Depuis cette dernière, fusionnez votre branche principale avec votre branche de développement.

Show / Hide solution

```
# On se place sur notre branche principale
git checkout main

# On fusionne les deux branches
git merge une_branche
```

Une fois fait, il suffit de pousser les modifications sur le dépôt distant à l'aide de `git push` et notre branche principale sera à jour. La branche de développement est maintenant obsolète, on peut la supprimer en local à l'aide de la commande suivante:

```
git branch -D [nom-de-la-branche]
```

Pour la supprimer sur le **dépôt distant**, il faudra aller sur le site de votre dépôt, cliquer sur **branch** et supprimer la branche devenue obsolète.

Jusqu'à maintenant, les méthodes de travail présentées restent des cas d'école et ne reflètent pas tout à fait le workflow type que l'on peut rencontrer lorsque l'on participe à un projet en coopération sur GitHub. En effet, fusionner une branche de développement vers la branche principale ne se fait pas aussi simplement surtout lorsqu'il s'agit d'un travail sensible aux modifications, les ajouts et changements apportés doivent toujours être **revus par un administrateur désigné**.

Les Pull Requests

Pour encadrer ses fusions de branches, GitHub propose un système de **Pull Request** (PR). Dans la plupart des cas, une personne de l'organisation sera désignée pour surveiller et autoriser les fusions de branches vers la branche principale du projet. Cet outil a pour but de limiter les **merge conflicts** ou conflits de modifications, qui peuvent se produire lorsque deux acteurs d'un projet modifient en parallèle sur deux branches différentes, des parties communes du projet. Dans ce genre de situation, les acteurs soumettront une Pull Request à l'administrateur du projet et ce dernier devra se charger d'effectuer les modifications nécessaires pour résoudre les conflits afin d'autoriser le *merge* des branches.

> Pour tester le système de pull request, créez une nouvelle branche dans votre dépôt et ajoutez y un fichier quelconque qui n'est pas déjà présent sur la branche principale ou modifiez un fichier existant.

Show / Hide solution

```

# On se place sur la nouvelle branche
git checkout une_nouvelle__branche

# Une fois fait on pousse cette nouvelle branche sur notre dé
git push -u origin une_nouvelle_branche

# Notre branche faite, on peut créer notre nouveau fichier te
nano fichier_3.txt

# On le remplit avec du texte au choix, puis on l'ajoute au p
git add fichier_3.txt

# On commit puis on push
git commit -m "ajout de fichier_3.txt sur une nouvelle branch
```

21

Une fois cette étape faite, rendez vous sur la page principale de votre dépôt et allez sur l'onglet **Pull Request**. Soumettez une nouvelle pull request avec le bouton "*New pull request*", décrivez le contenu de vos changements apportés dans l'espace prévu à cet effet. Vous pouvez également assigner un relecteur (*reviewer*), et un label pour décrire davantage vos changements. Du côté administrateur, ce dernier peut voir quels fichiers ont été ajoutés, modifiés ou supprimés, apporter ses propres changements à ces mêmes fichiers et finalement, autoriser le **merge** de la branche. GitHub notifie automatiquement l'administrateur de la possibilité de fusionner les branches. Le cas contraire, il sera également notifié d'un **merge conflict** qu'il faudra résoudre en modifiant les fichiers qui posent problème.

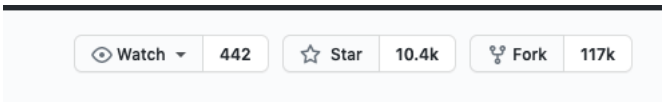
Enfin, si aucun problème n'est détecté par GitHub et l'administrateur, ce dernier peut autoriser la pull request et un nouveau commit sera créé comme précédemment dans le paragraphe introduisant le merge. GitHub proposera de supprimer la branche désormais obsolète, en local il nous suffira de `pull` les changements apportés sur la branche principale, puis de **supprimer la branche locale** elle aussi devenue obsolète.

Fork

Une autre manière de participer à un projet est de **fork** ce dernier. En résumé, il s'agit de copier l'intégralité d'un dépôt GitHub créé par quelqu'un d'autre et d'en faire un de vos dépôts personnels, le cloner, le modifier... comme si c'était le votre. Ainsi, vous pourrez travailler dessus sans modifier l'architecture propre du dépôt, essayer vos changements de votre côté et une fois que ceux-ci vous conviennent, et que vous pensez qu'ils contribueront au projet originel, vous pouvez soumettre une **pull request**.

C'est notamment grâce à ce système que les projets open source trouvent des contributeurs : n'importe qui peut forker le dépôt public d'un projet open source, tenter de l'améliorer, et proposer ces modifications via une pull request. Ces changements seront alors examinés par les administrateurs du projet open source, puis acceptés ou refusés. L'administrateur peut également discuter avec le contributeur via la pull request, notamment pour lui demander d'améliorer ses modifications avant qu'il les intègre (mise à jour de la documentation, ajout de test, de commentaires, etc.).

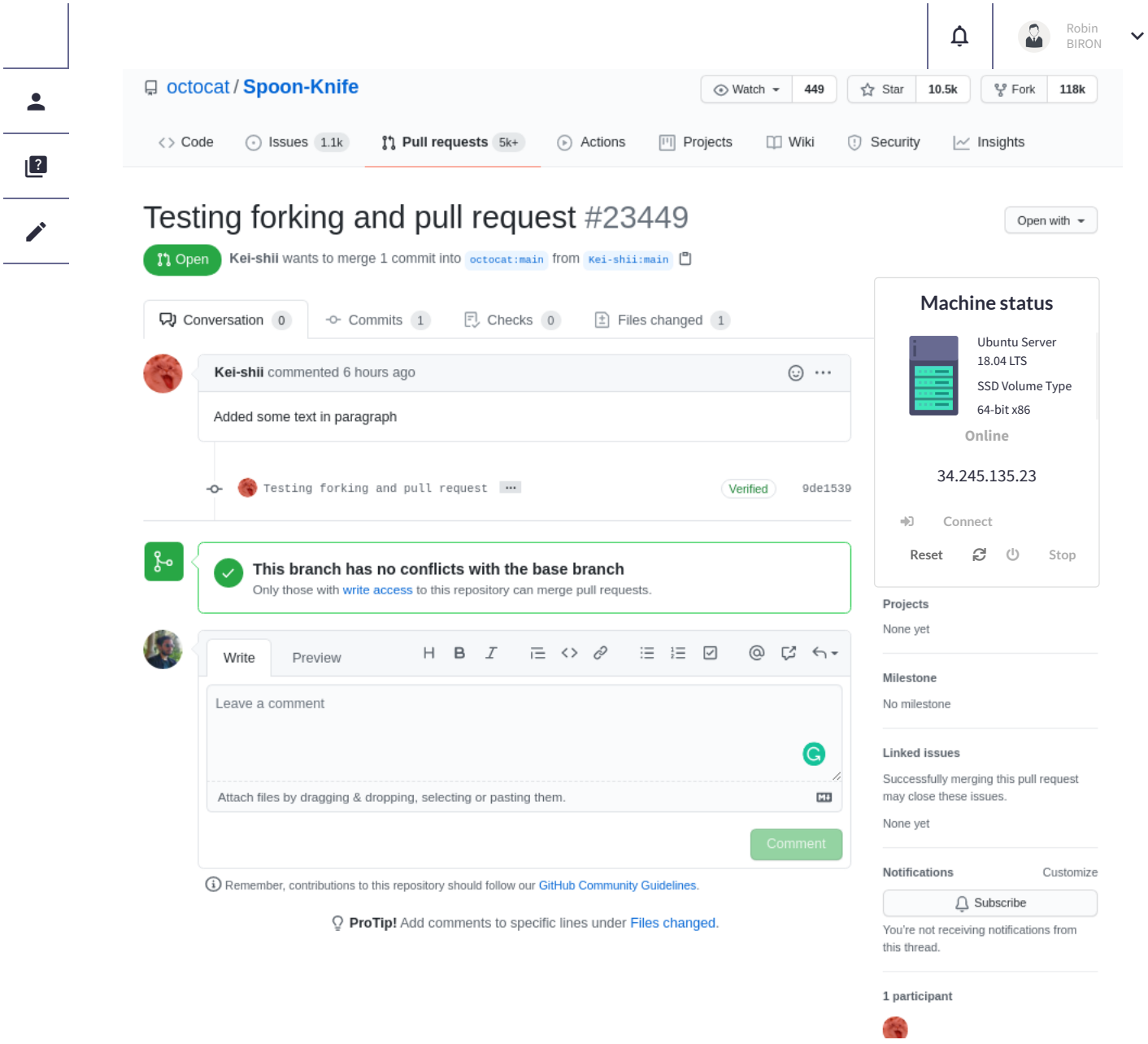
Pour "Forker" un dépôt, cliquez sur le bouton éponyme situé en haut à droite du dépôt.



Dès lors, comme si c'était votre dépôt, vous pourrez le cloner et effectuer toutes les actions qui vous semblent positives au développement du projet.

> Pour vous y essayer, "forkez" le dépôt test Spoon-Knife, effectuez y des modifications et soumettez une pull request au dépôt originel.

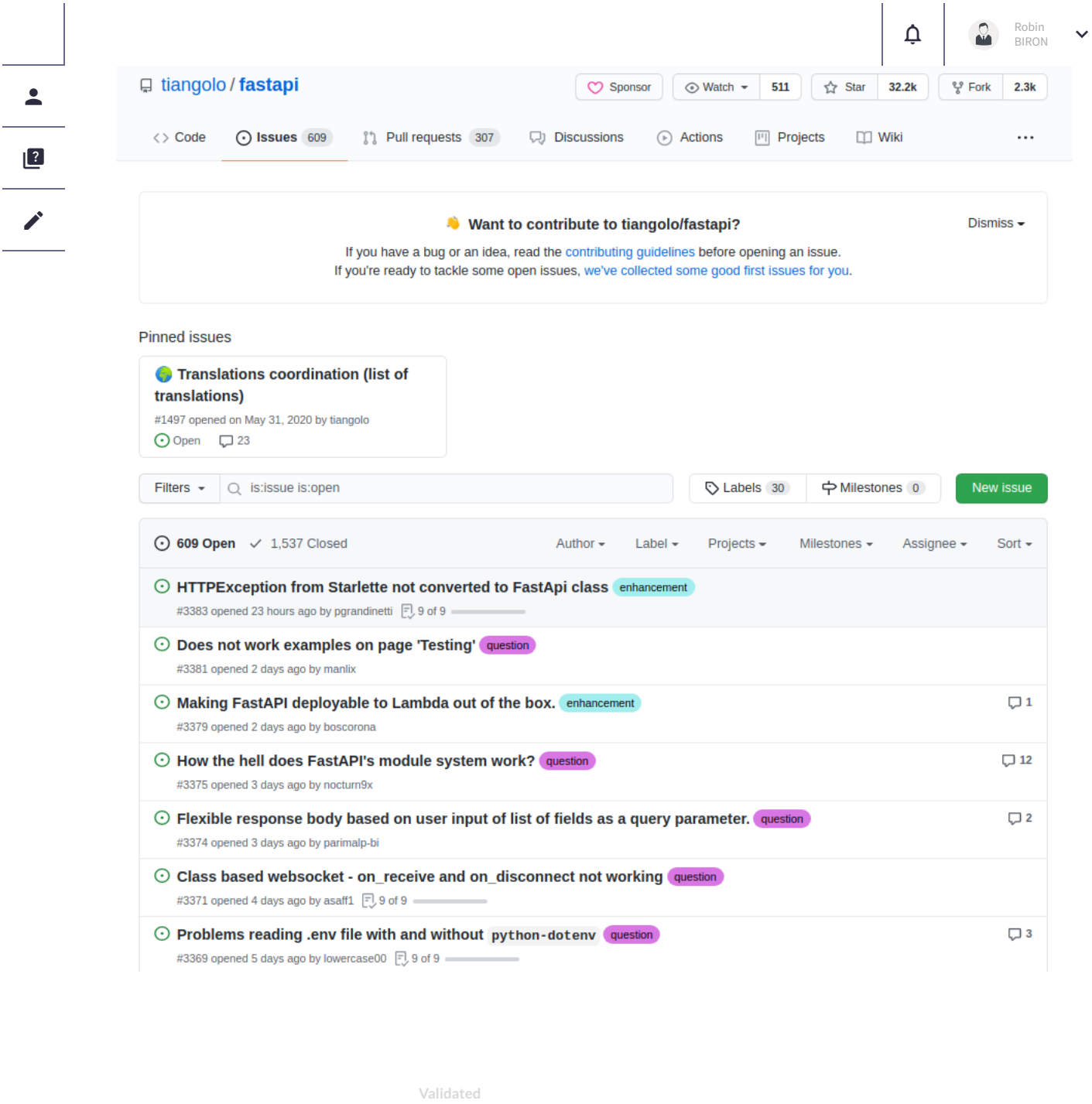
Après avoir "forké" le dépôt, vous pouvez également créer une pull request vers le dépôt original. Vous verrez que vous pouvez ajouter une description à votre pull request pour préciser les changements apportés, et discuter directement sur la PR de vos changements avec les administrateurs et les autres contributeurs du projet :



Issues

Vous pouvez également créer des "*issues*" (problèmes) sur un de vos dépôt, ou un dépôt public d'un projet open source. Une *issue* peut servir à faire remonter un bug, demander l'ajout d'une fonctionnalité, ou simplement poser une question quant au projet.

Voici par exemple une capture d'écran des issues du dépôt GitHub de FastAPI :


Robin BIRON
tiangolo / fastapi
Sponsor
Watch 511
Star 32.2k
Fork 2.3k
Code
Issues 609
Pull requests 307
Discussions
Actions
Projects
Wiki
Want to contribute to tiangolo/fastapi?
Dismiss
If you have a bug or an idea, read the contributing guidelines before opening an issue.
If you're ready to tackle some open issues, we've collected some good first issues for you.
Pinned issues
Translations coordination (list of translations)
#1497 opened on May 31, 2020 by tiangolo
Open 23
Filters
is:issue is:open
Labels 30
Milestones 0
New issue
609 Open
1,537 Closed
Author
Label
Projects
Milestones
Assignee
Sort
HTTPException from Starlette not converted to FastApi class enhancement
#3383 opened 23 hours ago by pgrandinetti 9 of 9
Does not work examples on page 'Testing' question
#3381 opened 2 days ago by manlix
Making FastAPI deployable to Lambda out of the box. enhancement 1
#3379 opened 2 days ago by boscorona
How the hell does FastAPI's module system work? question 12
#3375 opened 3 days ago by nocturn9x
Flexible response body based on user input of list of fields as a query parameter. question 2
#3374 opened 3 days ago by parimalp-bi
Class based websocket - on_receive and on_disconnect not working question
#3371 opened 4 days ago by asaff1 9 of 9
Problems reading .env file with and without python-dotenv question 3
#3369 opened 5 days ago by lowercase00 9 of 9

Validated