Rapport d'audit sur la qualité du code source de GéoNature

Logilab
104 boulevard Auguste Blanqui
FR-75013 PARIS
contact@logilab.fr

référence	MNHN06-210614-RAP-1
version	1.0
date	14 juin 2021
auteur	Logilab

Table des matières

Tous droits réservés au Muséum National d'Histoire Naturelle

Tous droits réservés au Muséum National d'Histoire Naturelle

CHAPITRE 1

Qualité code source

La qualité logicielle vise à répondre à deux questions : Le code source est-il correct ? Le code source est-il maintenable et facile à faire évoluer ?

Nous traiterons la première question dans la partie de l'audit dédiée aux tests, et nous nous allons nous concentrer sur la deuxième question dans ce chapitre.

Un code source maintenable et évolutif permet aussi bien aux développeurs actuels qu'aux nouveaux arrivants de faire évoluer l'application et de maintenir l'existant de façon efficace. Les prérequis sont un code source qui :

- est bien documenté,
- suit un style,
- a une complexité minimale.

Pour évaluer ces points, nous nous sommes servis des outils spécifiques aux langages de programmation utilisés. Il est aussi à noter que nous nous sommes concentrés sur les commentaires dans le code source lui-même afin de déterminer la qualité de la documentation.

1.1 JavaScript/TypeScript

Pour évaluer la qualité du code *frontend*, nous avons regardé :

- l'architecture du projet
- les versions des bibliothèques et cadriciels utilisés,
- le formatage du code,
- la complexité du code,
- les commentaires présents dans le code,
- les tests.

Tous droits réservés au Muséum National d'Histoire Naturelle

1.1.1 Architecture et bibliothèques

Installation

Pour pouvoir travailler sur le code JS, il est nécessaire de disposer d'une base de données configurée pour lancer les scripts de génération de fichiers de configuration. Ceci implique donc de se trouver dans la configuration exacte d'installation de l'application, ce qui est un frein au développement *frontend « pur »*.

Architecture

Le dossier `frontend` a une architecture classique de projet Javascript : un dossier `src` qui contient les sources. Celui-ci contient :

- un dossier `app` contenant différents modules et des composants utilisés dans `app`,
- un dossier `conf` avec la configuration de l'application,
- un dossier `custom` avec les images, les feuille de style et les composants `footer` et `introduction` à personnaliser,
- des dossiers d'*assets* externes ou propres à l'application de base,
- un dossier d'environnement (2 fichiers initialisant la variable `environment.production` à `true` ou `false`),
- des chemins créés dans le *tsconfig* pour renvoyer vers différents modules du projet (`@geonature_common`, `@geonature`, `@geonature_config`).

Les dépendances entre fichiers soit sont en chemins relatifs (`import { ModuleService } from './services/module.service';`) soit utilisent les chemins du *tsconfig* (`import { AuthGuard } from '@geonature/routing/routes-guards.service';`). Cette différence n'est pas choquante tant qu'elle reste cohérente. Il faudrait idéalement choisir quel mode d'import est préféré pour le projet et s'y tenir.

Le choix pourra être guidé par la forme de publication que souhaite prendre le projet *frontend* geonature. On pourrait, par exemple, envisager de publier un paquet `@geonature/components` sur npmjs pour rendre les composants de base disponibles dans un paquet. Alternativement, si on considère qu'ils ne sont pas dissociables de l'application de base, on pourrait créer un `@geonature/baseline`. Dans chaque cas, l'idée est que les développeurs de nouveaux composants puissent installer aisément un module grâce à la commande `npm install`.

De même, pour éviter des imports se basant sur la structure figée du projet, par exemple `import { ModuleConfig } from "../../../../../../external_modules/occhab/frontend/app/module.config";`, il serait pertinent de publier la partie *frontend* de chaque module externe comme module sur npmjs. Les modules dépendant d'autres modules peuvent alors s'entre-référencer. La gestion des versions compatibles entre elles en serait également simplifiée. En effet, les modules dont dépendra l'application finale et leurs versions seraient spécifiées explicitement dans le `package.json`.

Versions des modules

La commande npm outdated [1] vérifie quelles dépendances du projet sont obsolètes et pourraient être mises à jour.

Les résultats peuvent être consultés dans un fichier `npm-outdated` fourni en complément de ce rapport.

- Sur les 77 modules du `package.json`, 62 sont remontés par la commande et pourraient être mis à jour.
- Certains paquets comme `popper.js` sont dépréciés.

Nous notons, en particulier, les versions des cadriciels qui commencent à dater :

- Angular est utilisé dans sa version 7.2.6 (fixée) alors que la version 11 d'Angular a été récemment publiée
- Typescript est limité à la v3 alors que la v4 est sortie.

Mettre à jour les dépendances (notamment Angular) dans sa dernière version va demander un réusinage du code mais permettra de profiter de nouvelles fonctionnalités et d'éviter des failles de sécurité.

1. https://docs.npmjs.com/cli/v7/commands/npm-outdated

Tous droits réservés au Muséum National d'Histoire Naturelle

Organisation du `package.json`

Différence entre *dependencies* et *devDependencies* : les *dependencies* sont les paquets sur lesquels se basent l'application et qui sont utilisés lors de l'exécution. Ils sont inclus dans le paquet final. Les *devDependencies* sont les paquets utiles au développement du projet (compilation, test, analyse du code, etc.). Ils ne sont pas insérés dans le paquet final.

Les bonnes pratiques du code TypeScript ne se prononcent pas vraiment sur l'emplacement des modules `@types` (*dependencies* ou *devDependencies*). Certaines sources préconisent toutefois de les mettre dans les *devDependencies* quand le projet est une application *frontend*. Cela évite de surcharger inutilement le paquet final.

D'autres préfèrent mettre le module `@types` dans la même catégorie que le module qu'il concerne. ette dernière option serait plus adaptée pour les projets de type « bibliothèque » qui ont vocation à devenir des modules à part entière.

Dans tous les cas, il est important d'être cohérent sur un choix et de s'y tenir. En l'état, tous les modules `@types` des dépendances sont au même endroit que le module qu'ils concernent sauf `@types/jquery` qui se trouve dans les *devDependencies* alors que `jquery` est dans les *dependencies*.

Lorsque la version 5 de bootstrap sera sortie, il sera intéressant de l'utiliser. En effet, cette version va supprimer la dépendance de bootstrap à jquery. La bibliothèque jquery pourra alors être supprimée du `package.json` car elle n'est pas utilisée par le projet *frontend*.

Nous avons remarqué que certaines dépendances (popper.js, jquery, …) semblent dédiées à l'affichage de pages d'administration sous forme de gabarits jinja depuis le *backend*. Ces dépendances ne sont pas utilisées par l'application *frontend* ; elles vont donc encombrer inutilement le paquet final. De plus, il est possible que les versions attendues par les pages d'administration puissent entrer en conflit avec des dépendances de dépendances de l'application *frontend* lors d'une mise à jour (Exemple peu probable : jquery spécifiée en v3 dans le `package.json` et une bibliothèque de l'application *frontend* qui nécessite jquery en v4).

Nous conseillons d'isoler ces paquets de l'application *frontend* pour les rendre propres à l'application *backend*, puisque c'est le cas en pratique. Cet isolement peut être mis en œuvre de différentes façons :

- télécharger les fichier référencés directement dans le dossier `static` du *backend* pour qu'ils soient servis par flask. Leur version sera ainsi fixée et comme ces dépendances ne seront pas référencées dans le `package.json`, il n'y aura pas de problème de compatibilité.
- utiliser des liens CDN dans le code pour récupérer, par exemple, la dernière version de la bibliothèque.

1.1.2 Qualité de code

Prettier et TSLint sont déjà présents et configurés. Ce sont de bons indicateurs de la qualité de code (formatage, complexité du code, etc.) Toutefois, ils n'ont pas l'air d'être appliqués au code. Leur exécution renvoie des erreurs.

Nous détaillons leurs résultats ci-dessous.

Prettier

Nous avons lancé la commande suivante pour observer si les fichiers du projet *frontend* sont formatés de manière uniforme et conforme à la configuration du projet.

*npx prettier -c "**/*.{ts,html}"*

De nombreuses erreurs de formatage sont présentes. Ces problèmes peuvent être résolus facilement et automatiquement avec l'option `--write` de prettier.

Nous proposons d'ajouter une commande « format » dans le `package.json` pour formater les fichiers. Cette commande pourrait être, par exemple :

```
prettier --write '**/*.{html,ts}'.
```

Comme décrit dans la section sur la qualité de code côté *back*, il est possible de définir des *pre-commit hooks* qui s'exécutent avant la création d'un *commit*. Il peut être intéressant d'ajouter la commande de formatage dans un *pre-commit hook*.

Lint

Nous avons lancé la commande suivante, qui s'appuie sur TSLint comme configuré dans le projet : `ng lint`

Il est à noter que TSLint est déprécié depuis 2019 [2]. Nous préconisons de le remplacer par ESLint [3].

TSLint renvoie des erreurs liées à la configuration du *linter*, probablement dues à une montée de version de TSLint 4 à 5 : des règles de formatage ont été dépréciées ou supprimées dans la v5 mais la configuration TSlint du projet *frontend* n'a pas changé pour prendre en compte ces modifications

Beaucoup d'erreurs de formatage sont détectées par TSLint (elles étaient déjà signalées par Prettier).

Les fichiers générés depuis le *backend*, comme notamment `frontend/src/conf/app.config.ts`, comportent des erreurs de formatage. C'est un problème à traiter lors de la génération de ces fichiers (côté *back* et dans les fichiers `.sample`).

D'autres erreurs de *lint* sont signalées comme l'utilisation de == au lieu de === en TypeScript, des imports mal ordonnés, etc. Les erreurs remontées sont faciles à corriger.

Documentation

Le projet intègre Compodoc qui génère de la documentation de code pour Angular. Cet outil est assez puissant et permet notamment de présenter sous forme de diagramme les interdépendances des composants et services.

La documentation est publiée en ligne : http://docs.geonature.fr/frontend/

Nous notons que certains commentaires sont en français et d'autres en anglais. Il serait plus judicieux qu'ils soient tous en anglais.

Compodoc présente également des statistiques sur la couverture de documentation : http://docs.geonature.fr/frontend/coverage.html. La couverture de documentation est de 5% sur la documentation en ligne. En local, la commande utilise le fichier `tsconfig.app.json` qui exclut les fichiers de test. La couverture de documentation est de 12% dans cette configuration, ce qui reste améliorable.

Note : Il faut ajouter un `/` à la fin de la commande :

```
npx compodoc -p src/tsconfig.app.json --output=../docs/build/html/frontend/
```

dans le `package.json` afin de pouvoir naviguer dans les fichiers générés.

1.2 Python

L'application GeoNature contient plus que 20 000 lignes de code source Python. Une partie des outils que nous utilisons pour évaluer la qualité de code source Python sont déjà intégrés dans le projet (Flake8, Pylint, Black). En plus de ces outils, nous avons lancé Radon et Pydocstyle.

Les outils et leurs résultats sont exposés dans les sections suivantes.

2. https://palantir.github.io/tslint/
3. https://eslint.org/

Tous droits réservés au Muséum National d'Histoire Naturelle

1.2.1 Flake8

La tâche principale de Flake8 [4] est de vérifier si le code source suit les recommandations de style PEP 8 [5]. Comme cet outil fait déjà partie de la boîte à outils du projet, nous partons du principe que le lecteur connaît l'outil et les recommandations PEP 8.

En lisant la configuration existante :

```
[flake8]
ignore=E402
exclude = .git,venv,frontend,__pycache__
max-complexity = 5
```

la première chose qui apparaît est qu'elle n'est pas compatible avec l'outil Black, qui est aussi utilisé par le projet. Notamment, la longueur maximale des lignes par défaut n'est pas adaptée. L'erreur correspondante (E501) est la deuxième la plus fréquente avec 1483 occurrences.

Après avoir adapté la configuration pour la rendre compatible avec Black [6] :

```
[flake8]
ignore = E402,E203,W503
max-line-length=99
exclude = .git,venv,frontend,__pycache__
max-complexity = 5
```

elle n'apparaît plus que 409 fois, et seulement 212 fois après avoir lancé Black. Les erreurs E203 (*whitespace before ":"*) et W503 (*line break occurred before a binary operator*) sont à ignorer.

Pour l'analyse des autres erreurs signalées, nous nous appuyons sur les résultats produits par la configuration actuelle. Parmi les autres erreurs remontées par Flake8, les erreurs F405 (42 occurrences), F403 (5 occurrences) et F821 (6 occurrences) sont les plus pertinentes. Les deux premières peuvent masquer des erreurs dans le code source, et la dernière peut même indiquer la présence de bogues.

La prévalence d'erreurs liées au style, comme p.ex. W191 (*indentation contains tabs*) avec 2087 occurrences ou E226 (*missing whitespace around arithmetic operator*) avec 588 occurrences indique qu'il n'y a pas de procédure en place qui renforce le respect de ces recommandations de style.

Il est possible de s'attaquer à ce problème selon deux directions. D'un côté, il faudrait encourager les développeurs à suivre ce style pendant le développement, par exemple en rajoutant des liens vers des *plugins* PEP 8 pour les éditeurs les plus répandus, comme Python-autopep8 [7] pour Visual Studio ou Syntastic [8] pour Vim.

Nous avons aussi fait de bonnes expériences avec la définition d'un pre-commit hook [9] qui lance des outils comme Flake8 et qui empêche la création d'un *commit* en cas de non-respect du style.

L'autre angle d'attaque est l'intégration d'un outil comme Flake8 ou Pylint dans l'intégration continue que nous discuterons dans le chapitre consacré à ce sujet.

Finalement, Flake8 remonte aussi des erreurs (C901) qui peuvent indiquer des parties du code source trop complexes. Nous avons utilisé un autre outil (Radon) pour faire ce genre de mesures, et nous analyserons ces résultats dans la section dédiée à cet outil.

Les résultats de Flake8 se trouvent dans un fichier `flake8` fourni en complément de ce rapport.

4. https://flake8.pycqa.org/en/latest/
5. https://www.python.org/dev/peps/pep-0008/
6. https://black.readthedocs.io/en/stable/compatible_configs.html#flake8
7. https://marketplace.visualstudio.com/items?itemName=himanoa.Python-autopep8
8. https://github.com/vim-syntastic/syntastic
9. https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks

Tous droits réservés au Muséum National d'Histoire Naturelle

1.2.2 Pylint

Pylint [10] est un outil similaire à Flake8, il permet non seulement de vérifier le respect des recommandations PEP 8, mais aussi le suivi d'un style particulier, car il est très configurable. Comme Flake8 et Black, Pylint est déjà en usage et adapté au projet GeoNature. Par conséquent, le résultat final de Pylint est correct (7.66/10).

En examinant le fichier de configuration, nous nous sommes rendus compte qu'il n'est pas non plus adapté à un usage simultané avec Black. Après avoir intégré les changements nécessaires [11] :

```
diff --git a/backend/.pylintrc b/backend/.pylintrc
index 191bfd574..35205e274 100644
--- a/backend/.pylintrc
+++ b/backend/.pylintrc
@@ -128,7 +128,9 @@ disable=print-statement,
         dict-keys-not-iterating,
         dict-values-not-iterating,
         E1101,
-        C0103
+        C0103,
+        C0330,
+        C0326
```

et lancé Black sur le code source, le score final est monté à 7.94/10.

Nous jugeons que Pylint est plus adapté pour suivre l'évolution de la qualité du code source d'un point de vue global plutôt que pour un suivi quotidien dans le processus de développement (comme une partie du pipeline de la CI). Des outils plus légers, comme Flake8, se prêtent mieux à ce cas d'usage.

Du fait qu'une partie des résultats se retrouve dans les résultats des autres outils (Flake8, Black, Pydocstyle) et que nous avons eu des soucis avec des faux positifs liés à des bibliothèques tierces, nous ne discuterons pas plus avant les résultats de Pylint. Ces résultats peuvent être consultés dans un fichier `pylint` fourni en complément de ce rapport. Pour limiter le nombre de faux positifs, nous l'avons lancé en laissant de côté les erreurs liées aux imports.

1.2.3 Black

Black [12] est un outil de formatage automatique pour du code source Python. Tout comme Flake8 et Pylint, Black fait déjà partie de la boîte à outils du projet, et ne nécessite donc pas une introduction détaillée.

Nous nous sommes aperçus que l'outil Black n'est pas fonctionnel dans l'état actuel suite à une incompatibilité de version du module Click. La version de Black qui est renseignée dans les dépendances n'est, en fait, plus compatible avec une version de Click inférieure à 7.0, un oubli dans les dépendances du projet qui est corrigé dans la version suivante [13].

Afin de respecter les dépendances actuelles, nous avons corrigé manuellement le paquet pour pouvoir lancer Black. Les résultats sont, par conséquent, donnés avec réserves. La version de Click utilisée (6.7) est une version assez ancienne [14] ; il est fortement recommandé d'installer une version plus récente de cette bilbiothèque, ce qui remettrait Black en état de marche.

Il y a deux fichiers (`backend/geonature/core/gn_permissions/routes.py` et `backend/geonature/core/users/routes.py`) dans lesquels des tabulations traînent à la fin de certaines lignes, ce qui empêche Black de les formater.

Comme avec Flake8, il est possible d'intégrer Black dans le flux de travail d'un développeur à l'aide d'un *pre-commit hook*. De plus, l'option `--check` permet d'intégrer l'outil facilement dans le pipeline de la CI.

Les résultats de Black peuvent être consultés dans un fichier `black` fourni en complément de ce rapport.

10. https://pylint.org/
11. https://github.com/psf/black/blob/master/docs/compatible_configs.md#pylint
12. https://black.readthedocs.io/en/stable/
13. https://black.readthedocs.io/en/stable/change_log.html#packagingi
14. https://pypi.org/project/click/#history

 Tous droits réservés au Muséum National d'Histoire Naturelle

1.2.4 Radon

Radon [15] est un outil regroupant plusieurs métriques de qualité de code source, notamment :

- le nombre cyclomatique (qui est aussi mesuré par Flake8),
- les métriques d'Halstead,
- différentes métriques de base et
- l'index de maintenabilité.

Comme ce dernier index fait l'objet de contestations [16], nous ne l'avons pas inclus dans l'analyse des résultats.

Les résultats des différentes métriques peuvent être consultés dans les différentes sections d'un fichier `radon` fourni en complément de ce rapport.

Le nombre cyclomatique est présumé connu, puisqu'il est déjà intégré dans la configuration de l'outil Flake8. Radon classe les blocs de code source en fonction de leur nombre cyclomatique dans des catégories allant de A à F, de manière ascendante. Les résultats sont très positifs. Sur les 734 blocs de code source, seuls 4 figurent dans une des catégories jugées complexes (D à F), et aucun bloc n'entre dans la catégorie F. La plus grande partie du code source figure dans les catégories A et B, avec seulement 33 blocs qui se retrouvent dans une catégorie égale ou supérieure à C. Malgré les erreurs qui sont remontées par Flake8, la complexité du code source est donc jugée raisonnable.

Les résultats qui se trouvent dans l'annexe ont été produites en lançant la commande :

```
radon cc --min A --max F . -s -a
```

depuis la racine du projet.

Les métriques d'Halstead [17] ont pour but de déterminer à quel point le code est accessible aux développeurs. Nous présentons ici les résultats pour les métriques « volume », « difficulty », « effort » et « bugs ».

Les différentes métriques d'Halstead ont été calculées en lançant :

```
radon hal . | awk /"^    volume:|^[a-z]"/
radon hal . | awk /"^    difficulty:|^[a-z]"/
radon hal . | awk /"^    effort:|^[a-z]"/
radon hal . | awk /"^    bugs:|^[a-z]"/
```

« Volume » nous donne une idée de « la quantité de code source » qu'il faut comprendre pour pouvoir y contribuer. Cette métrique pointe donc vers des fichiers qui sont potentiellement longs ou qui contiennent beaucoup de code source hétérogène.

Les fichiers les plus « volumineux » sont :

```
data/scripts/import_mtd/run_import_mtd.py:                      11362.
↪209594103047
data/scripts/import_mtd/import_jdd_and_ca.py:                   7840.849864282301
data/scripts/import_ginco/import_mtd.py:                        6644.013662307062
backend/geonature/core/gn_meta/routes.py:                       2652.
↪4180978518293
contrib/occtax/backend/repositories.py:                         1481.
↪5799303197637
backend/geonature/core/gn_commons/repositories.py:              1409.
↪4105445469831
backend/geonature/core/gn_meta/repositories.py:                 1166.
↪7667651340112
backend/geonature/utils/utilssqlalchemy.py:                     1040.
↪4240751139428
backend/geonature/core/gn_synthese/routes.py:                   930.7557569766511
backend/geonature/core/gn_synthese/utils/query_select_sqla.py:  892.5271150784394
```

15. https://radon.readthedocs.io/en/latest/
16. http://avandeursen.com/2014/08/29/think-twice-before-using-the-maintainability-index/
17. https://radon.readthedocs.io/en/latest/intro.html#halstead-metrics

Tous droits réservés au Muséum National d'Histoire Naturelle

« Difficulty » indique si le code source est difficile à comprendre, en fonction de la récurrence et de l'homogénéité du code source.

Les fichiers les plus « difficiles » sont :

```
backend/geonature/core/gn_meta/routes.py:                          8.580645161290322
backend/geonature/core/gn_commons/repositories.py:                 8.1875
backend/geonature/core/gn_synthese/utils/query_select_sqla.py:     6.911764705882353
backend/geonature/utils/utilssqlalchemy.py:                        6.
↪6647058823529415
contrib/occtax/backend/repositories.py:                            6.391304347826087
contrib/gn_module_occhab/backend/blueprint.py:                     6.346153846153846
contrib/gn_module_validation/backend/query.py:                     5.739130434782608
backend/geonature/core/gn_synthese/utils/query.py:                 5.660377358490566
backend/geonature/core/gn_meta/models.py:                          5.433962264150943
backend/geonature/core/users/routes.py:                            5.273972602739726
```

« Effort » est dérivé des deux métriques précédentes, et nous donne une idée de quels fichiers peuvent poser le plus de problèmes à de nouveaux développeurs :

```
data/scripts/import_mtd/run_import_mtd.py:                         29916.
↪456112133023
data/scripts/import_mtd/import_jdd_and_ca.py:                      27685.
↪832706625115
data/scripts/import_ginco/import_mtd.py:                           23491.81926815728
backend/geonature/core/gn_meta/routes.py:                          22759.45851705118
backend/geonature/core/gn_commons/repositories.py:                 11539.
↪548833478424
contrib/occtax/backend/repositories.py:                            9469.228250304577
backend/geonature/utils/utilssqlalchemy.py:                        6934.120453553513
backend/geonature/core/gn_synthese/utils/query_select_sqla.py:     6168.937413042155
backend/geonature/core/gn_meta/repositories.py:                    5193.380329591061
backend/geonature/core/gn_meta/models.py:                          4603.112689493099
```

La métrique « bugs » est une métrique résultant de la mise en relation du « volume » et d'une valeur empirique. Elle indique le nombre de bogues potentiellement introduits par une partie de code source. Les fichiers qui ont potentiellement introduit le plus de bogues sont :

```
data/scripts/import_mtd/run_import_mtd.py:                         3.787403198034349
data/scripts/import_mtd/import_jdd_and_ca.py:                      2.
↪6136166214274335
data/scripts/import_ginco/import_mtd.py:                           2.214671220769021
backend/geonature/core/gn_meta/routes.py:                          0.
↪8841393659506098
contrib/occtax/backend/repositories.py:                            0.
↪49385997677325455
backend/geonature/core/gn_commons/repositories.py:                 0.
↪4698035148489944
backend/geonature/core/gn_meta/repositories.py:                    0.
↪3889222550446704
backend/geonature/utils/utilssqlalchemy.py:                        0.
↪3468080250379809
backend/geonature/core/gn_synthese/routes.py:                      0.
↪31025191899221705
backend/geonature/core/gn_synthese/utils/query_select_sqla.py:     0.
↪2975090383594798
```

Comme ces métriques sont interdépendantes, il n'est pas surprenant que les mêmes fichiers soient renvoyés à chaque fois. En revanche, il est à noter qu'une partie de ces fichiers contiennent des blocs de code source jugés trop complexe par rapport au nombre cyclomatique :

Tous droits réservés au Muséum National d'Histoire Naturelle

```
contrib/occtax/backend/repositories.py
    F 163:0 get_query_occtax_filters - E (35)
data/scripts/import_mtd/import_jdd_and_ca.py
    F 853:0 insert_CA - D (27)
data/scripts/import_ginco/import_mtd.py
    F 918:0 insert_CA - D (27)
```

D'autre part, les fichiers dans `contrib/` et `data/scripts/` sont jugés mal documentés par Pydocstyle, ce qui peut rendre difficile l'intégration de nouveaux développeurs dans le code source.

Radon nous a donc permis d'identifier les dossiers `data/scripts/` et `contrib/` comme de bons candidats à une opération de refonte / amélioration.

1.2.5 Pydocstyle

Pydocstyle [18] est un outil qui vérifie que les recommandations de style des *docstrings* PEP 257 [19] sont suivies. Dans un premier temps nous l'avons lancé uniquement pour déterminer le niveau de documentation du code source. Les *docstrings* ne sont pas utilisées d'une façon conséquente dans GeoNature et ne semblent pas respecter un style cohérent. Même si la génération automatique de la documentation du code source n'est pas jugée nécessaire, des docstrings sont utiles, par exemple pour les développeurs qui utilisent des éditeurs affichant des suggestions d'auto-complétion.

Bien que Pydocstyle n'inclut pas, par défaut, les fichiers de tests, notre expérience nous a montré qu'ajouter des *docstrings* aux différents cas de test permet de facilement juger l'exhaustivité des tests. Un style de *docstring* possible serait :

```
"""Test <functionality>.

Trying: <how functionality is tested>
Expecting: <expected behaviour>
"""
```

Cette description permet de voir en un clin d'œil comment une fonctionalité donnée est testée, et comment elle devrait se comporter. Ceci est surtout utile pour voir si un cas de test existe déjà ou si un test échoue.

Les résultats de Pydocstyle peuvent être consultés dans un fichier `pydocstyle` fourni en complément de ce rapport.

1.2.6 Conclusion

Le projet GeoNature suit déjà la plupart des bonnes pratiques pour le développement Python (PEP 8, *docstrings*, tests unitaires, etc.) Toutefois, certaines de ces pratiques ne sont pas suivies jusqu'au bout. Bien qu'il ne soit pas nécessairement utile (ni même possible) d'avoir un score Pylint parfait, avoir du code source pour lequel Flake8 ne remonte plus d'erreurs est, d'expérience, possible pour un effort raisonable. Cela apporte beaucoup à la maintenabilité. Par ailleurs, certaines bonnes pratiques comme Black ou les tests automatisés (Travis CI) semblent avoir été abandonnées à un moment donné.

Nous proposons de faire le tri dans les outils qui sont actuellement utilisés (ou au moins intégrés) dans le projet. Pour ceux qui devraient continuer d'être utilisés, il faudrait définir des conditions précises (par exemple Flake8 ne devrait pas remonter d'erreurs, la note score Pylint devrait être supérieure à un minimum, etc.) et insérer ces outils dans l'intégration continue. Ce point sera évoqué plus longuement dans le chapitre dédié aux tests et à l'intégration continue.

Pour tous les outils qui ne sont pas en usage actif et qui ne seront pas remaniés, le mieux serait de supprimer du dépôt toutes leurs traces (fichiers de configuration, installation de l'outil et de ses dépendances, etc.)

18. http://www.pydocstyle.org/en/stable/
19. https://www.python.org/dev/peps/pep-0257/

Tous droits réservés au Muséum National d'Histoire Naturelle

1.3 PostgreSQL

Le projet GeoNature contient 97 scripts PostgreSQL. Le périmètre de l'audit ne permettant pas l'analyse approfondie de cette base de code, nous nous sommes concentrés sur quelques fichiers pour discuter des questions concernant l'intégration de nouveaux développeurs, à savoir :

- Est-ce que le code source est bien formaté et les requêtes sont lisibles ?
- Est-ce que les fonctions et les vues sont bien documentées ?

Contrairement aux la parties dédiées au code source Python et au code source Javascript, cette partie est donc plus courte et ne traite que du formatage et de la documentation du code source PostgreSQL.

Nous avons utilisé SQLFluff [20] pour faire une analyse du style du code source. Cet outil permet aussi de formater automatiquement le code source. Comme les résultats n'étaient pas très différents du code source de départ, nous ne les avons pas inclus. Le besoin serait surtout d'homogénéiser le style du code source, ce qui améliorerait la lisibilité.

Un peu comme le code source Python, le code source PostgreSQL est documenté mais d'une façon très hétérogène. Une partie des objets est documentée par des commentaires dans le code source, et une autre partie en ajoutant un commentaire à l'objet à l'aide de la commande `COMMENT`. D'autres ont été supprimés, mais se trouvent toujours dans le manuel (par exemple la fonction `delete_and_insert_area_taxon` qui a été remplacée par une vue). Il faudrait commencer par définir une stratégie cohérente pour la documentation du code source PostgreSQL (par exemple ajouter des commentaires à tous les objets de certains types), puis mettre à jour le manuel.

1.3.1 Conclusion

Dans l'état actuel, il est difficile en tant que nouvel arrivant de rentrer dans le code source PostgreSQL. Bien que le formatage du code source permette la plupart du temps de suivre le contenu assez facilement, l'emplacement des commentaires n'est pas toujours cohérent. Il y a parfois des commentaires au milieu du code, à des endroits gênants pour la lecture. Du code source très commenté est souvent plus difficile à comprendre.

La documentation aussi bien du côté du manuel que du côté du code source est parfois incomplète voire obsolète.

En résumé, comme pour le code source Python, des bonnes pratiques sont déjà mises en place (formatage du code source, même s'il ne semble pas être automatique, documentation du code source et de la base de donnée, etc.) mais il reste à les renforcer.

20. https://github.com/sqlfluff/sqlfluff

 Tous droits réservés au Muséum National d'Histoire Naturelle

CHAPITRE 2

Tests

2.1 JavaScript / TypeScript

Des tests existent dans le projet *frontend*, mis en place avec Karma et Jasmine. Ces deux outils sont ceux préconisés par AngularJS et des fichiers de configuration de test sont créés à l'initialisation du projet.

Jasmine [21] est le cadriciel de test. Karma [22] permet d'exécuter les tests dans un navigateur au choix.

Les tests sont rédigés dans des fichiers `.spec.ts`. Le projet frontend contient les fichiers de test suivants :

```
admin.component.spec.ts
app.component.spec.ts
page-not-found.component.spec.ts
sidenav-items.component.spec.ts
data-form.service.spec.ts
dynamic-form.component.spec.ts
multiselect.component.spec.ts
municipalities.component.spec.ts
taxonomy.component.spec.ts
map.component.spec.ts
```

La commande `npm run test` (ou `ng test`) renvoie des erreurs lors de l'exécution des tests :

```
ERROR in src/app/GN2CommonModule/form/data-form.service.spec.ts(34,13): error␣
↪TS2339: Property 'searchTaxonomy' does not exist on type 'DataFormService'.
src/app/GN2CommonModule/form/data-form.service.ts(178,8): error TS2339: Property
↪'map' does not exist on type 'Observable<any>'.
src/app/GN2CommonModule/form/data-form.service.ts(242,8): error TS2339: Property
↪'map' does not exist on type 'Observable<any>'.
src/app/GN2CommonModule/form/data-form.service.ts(274,76): error TS2339: Property
↪'map' does not exist on type 'Observable<Object>'.
src/app/GN2CommonModule/form/genericForm.component.ts(38,8): error TS2339:␣
↪Property 'distinctUntilChanged' does not exist on type 'Observable<any>'.
src/app/GN2CommonModule/form/multiselect/multiselect.component.spec.ts(3,39):␣
↪error TS2307: Cannot find module './select-search.component'.
src/app/GN2CommonModule/form/taxonomy/taxonomy.component.ts(91,8): error TS2339:␣
↪Property 'filter' does not exist on type 'Observable<any>'.
```

(suite sur la page suivante)

21. https://jasmine.github.io/
22. https://karma-runner.github.io/1.0/index.html

Tous droits réservés au Muséum National d'Histoire Naturelle

(suite de la page précédente)

```
src/app/GN2CommonModule/form/taxonomy/taxonomy.component.ts(123,8): error TS2339:␣
↪Property 'do' does not exist on type 'Observable<string>'.
src/app/GN2CommonModule/form/taxonomy/taxonomy.component.ts(134,14): error TS2339:␣
↪Property 'catch' does not exist on type 'Observable<Taxon[]>'.
src/app/GN2CommonModule/service/media.service.ts(109,10): error TS2339: Property
↪'switchMap' does not exist on type 'Observable<any>'.
```

Il semble que les tests n'ont pas évolué avec l'API (propriétés supprimées, modules introuvables). Beaucoup d'erreurs sont liées à un manque d'import d'opérateurs propres au module `rxjs` : `import 'rxjs/add/operator/map'`.

Une fois ces problèmes résolus, il y a des imports implicites que Karma et Jasmine imposent d'être explicités (notamment les imports des composants de `@angular/material`) directement dans les fichiers *specs* :

```
import { MatButtonModule, MatCardModule, MatIconModule } from '@angular/material';

...
describe('AdminComponent', () => {

...
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [AdminComponent],
      imports: [
        MatButtonModule,
        MatIconModule,
        MatCardModule
    ],
    }).compileComponents();
  }));
...

});
```

En l'état, les tests consistent uniquement à vérifier que le composant testé a bien été instancié lors du test. La couverture de test est donc inexistante.

L'écriture de tests est un moyen efficace de s'assurer que le développement d'une nouvelle fonctionnalité, ou du réusinage de code n'a pas créé un bogue.

Les bibliothèques Karma et Jasmine sont poussées et documentées par AngularJS, il est donc tout à fait possible d'écrire les tests en les utilisant.

Pour plus de robustesse, il est conseillé de mettre en place :

- des tests unitaires : tester les fonctions des services, des comportements très précis dans les modules, etc.
- des tests d'intégration : souvent basés sur l'interface et sur un scénario, ils consistent à vérifier que tous les composants de l'application fonctionnent correctement ensemble.

À noter que les tests unitaires ne doivent pas nécessiter de connexion vers une base de données, ne doivent pas faire d'appels sur le réseau, ne doivent pas modifier le système de fichiers, et doivent pouvoir être exécutés en parallèle les uns des autres. Écrire des test unitaires de fonctionalités qui auraient besoin d'une connexion vers Internet ou vers une base de données est tout à fait possible en utilisant des bouchons de test (*stubs*, *fakes*, *spies*) ou des simulacres(*mocks*). La documentation officielle d'AngularJS présente des exemples de ces utilisations [23].

Il est possible d'utiliser des fixtures [24] pour initialiser des données ou des états avant chaque test.

Dans l'idéal tous les composants doivent être testés par des **tests unitaires**. Pour se lancer dans la rédaction des tests, on peut prendre un composant ou service principal, dont le comportement est plutôt figé, par exemple le composant d'authentification (*login*), et vérifier que les fonctions de bases sont exécutées. Par exemple, pour le composant de

23. https://angular.io/guide/testing-components-scenarios#provide-service-test-doubles
24. https://angular.io/guide/testing-components-basics#componentfixture

 Tous droits réservés au Muséum National d'Histoire Naturelle

login, on va vouloir vérifier que lorsqu'on entre l'identifiant et le mot de passe dans le formulaire puis qu'on appuie sur le bouton de soumission, la fonction `loginOrPwdRecovery` est appelée.

Pour s'assurer que suffisamment de tests ont été écrits, la commande `ng test --no-watch --code-coverage` permet d'obtenir des statistiques de couverture de test. Comme précisé dans la documentation de test AngularJS [25], il est possible d'imposer une couverture minimale de test dans `karma.conf.js`.

Les **tests d'intégration** cherchent à vérifier que les composants s'intègrent bien les uns aux autres. Ils peuvent être écrits en se basant sur des scénarios ou des cas tests, ces derniers sont souvent créés à partir de l'interface utilisateur.

Les tests d'intégration sont généralement effectués avant la publication d'une version et si tous les tests unitaires réussissent. Ils peuvent être automatisés. Afin de réaliser ces tests, il est nécessaire, pour chaque fonctionnalité, de définir des critères d'acceptation [26]. Ces critères sont à discuter avec tous les acteurs du logiciel en question et permettent de définir précisément ce qui est attendu pour la fonctionnalité.

Selenium [27], et plus spécifiquement ses liaisons Python [28], permet d'effectuer des tests automatisés sur des interfaces Web à partir d'une instance d'intégration du projet.

Nous proposons d'organiser le travail en *Test-Driven-Development* (TDD) pour les prochaines fonctionnalités. Ce type de développement consiste à produire les tests de la fonctionnalité avant de la coder, ou tout au moins en même temps qu'on la développe. Il permet de morceler le code plus aisément (et de le tester plus facilement).

Pour que chaque fonctionnalité soit testée en phase de développement, il est possible d'utiliser des systèmes d'intégration continue (CI).

2.2 Tests Python

Dans l'état actuel de l'application, il n'est pas possible de lancer les tests en local sans créer une base de données. Bien que cela soit plutôt dans la nature d'une application construite au dessus d'une base de données, la partie de la documentation qui traite des tests Python [29] ne contient ni cette information ni n'explique comment mettre en place la base de données. Le cadre assez étroit de l'audit ne nous a pas permis de mettre en place une instance de GeoNature (le chapitre sur le déploiement automatique discute des problèmes du déploiement d'un environnement de développement). D'autre part, les tests automatiques de Travis CI ne sont plus actifs. Par voie de conséquence, nous n'avons pas pu lancer les tests et nous nous limiterons ici à une discussion sur la façon de lancer, de manière efficace, les tests et les outils de gestion de la qualité logicielle.

À part l'intégration continue par Travis CI qui n'est plus utilisée, rien n'est prévu pour lancer les différents outils de gestion de qualité logicielle (Flake8, Pylint, Black, Pytest) de façon structurée. Utiliser un outil d'automatisation comme par exemple Tox [30] permettrait d'une part de faciliter l'usage de ces outils en les lançant tous avec une seule commande, et d'autre part d'homogénéiser les environnements dans lesquels ils sont utilisés en les lançant dans un environnement virtuel contrôlé. De plus, Flake8 et Pytest peuvent être configurés par le même fichier que Tox, ce qui permet d'unifier les différents fichiers de configuration.

2.2.1 Exemple de fichier de configuration Tox

Le fichier de configuration Tox du projet GeoNature pourrait être :

```
[tox]
envlist = py37,flake8,black
skipsdist = true

[pytest]
testpaths = tests
```

(suite sur la page suivante)

25. https://angular.io/guide/testing-code-coverage
26. https://wikiagile.cesi.fr/index.php?title=Comment_cr%C3%A9er_des_crit%C3%A8res_d%27acceptation_%3F
27. https://www.selenium.dev/
28. https://selenium-python.readthedocs.io/
29. http://docs.geonature.fr/development.html#pytest
30. https://tox.readthedocs.io/en/latest/index.html

Tous droits réservés au Muséum National d'Histoire Naturelle

(suite de la page précédente)

```
addopts = -s
python_paths = .

[flake8]
ignore = E402
exclude = .git,venv,frontend,__pycache__
max-complexity = 5

[testenv]
deps =
  -r requirements.txt
  -r requirements-dev.txt
commands =
  {envpython} -m pytest

[testenv:flake8]
skip_install = true
commands = flake8

[testenv:black]
skip_install = true
command = black --check --config pyproject.toml .
```

 Tous droits réservés au Muséum National d'Histoire Naturelle

CHAPITRE 3

Tests automatiques et intégration continue

Le projet GeoNature a des tests automatisés en place qui sont lancés avec Travis CI [31]. Avant de proposer des améliorations, nous allons examiner l'existant de plus près.

La première chose qui saute aux yeux est qu'aucun *build* n'a été créé depuis plus d'un an [32], bien qu'il y ait eu des développements durant cette période. Les tests automatisés semblent avoir été désactivés.

Par ailleurs, nous avons constaté quelques problèmes mineurs. La version Python renseignée dans le fichier de configuration de Travis CI est la 3.5 alors que cette version n'est plus supportée depuis la version 2.5.0 de GeoNature [33]. Le lien du badge des tests automatisés pointe vers le pipeline de la branche de développement et non vers celui de la branche principale. Ceci est inhabituel et peut donner une mauvaise première impression du projet attendu que cette branche est moins stable que la branche principale. Finalement, bien qu'une partie de la configuration concerne le *frontend*, elle est commentée et ne semble pas avoir été en usage depuis longtemps.

3.1 Remaniement de l'existant

Comme les tests automatiques sont un outil essentiel d'un développement moderne, il faut commencer par réactiver le pipeline Travis CI. Un point d'attention particulier est la *Fin du support illimité de Travis CI* (page 16), que nous discuterons plus tard. Une fois que le pipeline pourra à nouveau être déclenché, il faudra mettre à jour sa configuration. Comme évoqué dans l'introduction, il faut corriger la version Python. La recommandation semble être d'utiliser la version 3.7 [34], mais la version 3.6 est officiellement toujours supportée. Dans le cas idéal, les tests seront donc lancés pour ces deux versions de Python. En pratique, cela peut ne pas être possible (car, par exemple, la durée du pipeline est trop longue). Dans ce cas, il est préférable de tester avec la version la plus répandue chez les utilisateurs.

Finalement, il serait préférable de pointer le lien du badge vers le pipeline de la branche principale et non vers celui de la branche de développement. En effet, le pipeline de la branche de développement est plus souvent cassé que celui de la branche principale (qui devrait être vert à tout moment). Afficher le badge de la branche de développement donne donc non seulement une fausse mais surtout une mauvaise impression de la stabilité du projet. D'autre part, il est perturbant d'arriver sur la branche principale du projet sur GitHub et d'être renvoyé sur le pipeline d'une autre branche sur Travis CI.

31. https://travis-ci.com/
32. https://travis-ci.org/github/PnX-SI/GeoNature/branches
33. http://docs.geonature.fr/CHANGELOG.html#manidae-2020-09-30
34. http://docs.geonature.fr/installation-standalone.html#python-3-7-sur-debian-9

Tous droits réservés au Muséum National d'Histoire Naturelle

3.2 Améliorations

Dans l'état actuel, les tests Python et la construction de la documentation se trouvent dans la même étape. L'inconvénient de cette façon de procéder est qu'il faut attendre que les tests Python soient terminé même si on ne s'intéresse qu'à la création de la documentation. En cas d'échec, il n'est pas évident d'identifier l'origine de l'erreur, il faut lire le journal d'exécution (*logs*) pour voir laquelle des deux parties a cassé le pipeline. Séparer les deux en des tâches (*jobs*) séparés dans une même étape (stage) [35] permettrait d'un côté de les exécuter en parallèle et de l'autre d'avoir une sortie organisée par tâche.

Actuellement, le pipeline Travis CI est utilisé pour trois tâches : lancer les tests Python, créer le manuel et déployer le manuel. Nous avons vu dans le chapitre traitant de la qualité du code source que les recommandations de style ne sont pas observées. Pourtant il est possible d'intégrer assez facilement la plupart des outils qui vérifient le respect de ces recommandations. Il serait envisageable de rajouter une étape (*stage*) nommée « lint » qui inclurait :

- une tâche(*job*) pour vérifier que Flake8 s'exécute sans remonter d'erreur,
- une tâche qui vérifie que Black n'a pas de fichiers à formatter, et
- une tâche qui vérifie que le code source a un score Pylint minimal.

De plus, si GeoNature adoptait la distribution par paquets PyPI ou par images Docker, il serait possible de les construire automatiquement à l'aide d'une étape (*stage*) « build » qui ne serait déclenchée que pour les nouvelles versions [36].

Ces étapes devraient également être mises en place pour le test du *frontend*. Si la séparation du *frontend* et du *backend* est mise en place, les tests *frontend* peuvent être exécutés en parallèle [37] des tests *backend*. Ces tests *frontend* incluraient également une étape (*stage*) « lint » avec TSLint et Prettier, une étape de test pour vérifier que la construction du projet aboutit et pour lancer les tests automatiques (par exemple avec Karma et Jasmine). La génération de la documentation Compodoc pourrait également être intégrée au pipeline de CI et publiée dans les *GitHub pages* au même titre que la documentation Sphinx.

La sauvegarde du résultat d'une tâche (*job*) est possible dans certains outils d'intégration continue comme GitLab-CI [38]. Ces artéfacts peuvent être utilisés d'une étape à l'autre et pourraient être utilisés notamment pour la documentation.

3.3 Fin du support illimité de Travis CI

Un point d'attention particulier concernant la solution Travis CI est la fin du support inconditionnel des projets au code source ouvert [39]. Le projet GeoNature utilise Travis CI en tant que dépôt public, et basculera donc vers un plan qui prévoit un montant fixe de ressources qui peut être renouvelé sous réserve [40]. Il en résulte donc une instabilité pouvant aboutir à l'arrêt de l'outil d'intégration continue.

Il y a quatre possibilités pour gérer cette situation :

- tester le plan prévu pour les projets code source ouvert [41],
- adopter un plan payant [42],
- migrer vers un autre service tiers, ou
- migrer vers un service hébergé par le projet GeoNature ou une des organisations maintenant le projet.

La première option serait de passer sur le nouveau plan, et voir si le montant fixe de ressources et son renouvellement répondent aux besoins du projet. L'avantage de cette solution est qu'il n'y a ni coût supplémentaire, ni développement à faire pour le moment. L'inconvénient est qu'il n'est pas clair si le crédit de ressources se renouvelle après avoir fait

35. https://docs.travis-ci.com/user/build-stages
36. https://docs.travis-ci.com/user/conditional-builds-stages-jobs/
37. https://docs.travis-ci.com/user/build-matrix/#using-different-programming-languages-per-job
38. https://docs.gitlab.com/ee/ci/
39. https://blog.travis-ci.com/2020-11-02-travis-ci-new-billing
40. https://docs.travis-ci.com/user/billing-faq/#what-if-i-am-building-open-source
41. https://docs.travis-ci.com/user/billing-overview/#free-plan
42. https://docs.travis-ci.com/user/billing-overview/

 Tous droits réservés au Muséum National d'Histoire Naturelle

une demande initiale [43] ou s'il s'agit d'un crédit unique [44]. Dans tous les cas, il n'y a aucune garantie que la demande soit acceptée ni que la situation ne soit pas réévaluée un jour.

Adopter un des plans payants donnerait un accès garanti aux ressources de Travis CI. En revanche, l'inconvénient manifeste est de risquer de payer cher pour un service qui pourrait être utilisé gratuitement. Comme la demande de ressources de GeoNature est très modeste, il est peu probable que cette solution soit rentable.

La troisième option serait de migrer vers un autre service qui propose des plans adaptés au besoin du projet GeoNature, comme par exemple GitHub Actions, qui est gratuit pour des entrepôts publics [45]. Toutefois, cette organisation tierce peut changer à tout moment ses conditions d'utilisation, ce qui remettra le projet dans la même situation qu'avec Travis CI.

Le coût de migrer vers une solution hébergée par le projet GeoNature dépend largement de l'infrastructure déjà accessible pour le projet. Bien que ce soit la meilleure option en terme d'indépendance et de pérennité, les coûts de la mise en place *ex nihilo* et de maintenance seraient disproportionnés par rapport aux besoins de GeoNature. Cette solution est seulement envisageable si le projet a accès à une infrastructure partagée existante. Dans ce cas, il existe des solutions qui permettraient de déclencher l'intégration continue sur cette infrastructure à partir de GitHub.

43. https://docs.travis-ci.com/user/billing-faq/#what-if-i-am-building-open-source
44. https://docs.travis-ci.com/user/billing-overview/#free-plan
45. https://docs.github.com/en/github/setting-up-and-managing-billing-and-payments-on-github/about-billing-for-github-actions

Tous droits réservés au Muséum National d'Histoire Naturelle

Tous droits réservés au Muséum National d'Histoire Naturelle

CHAPITRE 4

Déploiement automatique

Il y a deux cas d'usage pour le déploiement d'une application. Le cas le plus évident est le déploiement de l'application pour les utilisateurs finaux. Le second cas est souvent oublié bien qu'il ne soit pas moins important : il s'agit du déploiement d'un environnement de développement de l'application. Cces deux cas d'usage correspondent à des besoins différents.

Le déploiement de l'application dans un environnement de production est destiné à l'usage par des utilisateurs finaux. Il doit garantir une certaine qualité du service, notamment en ce qui concerne la stabilité. Par conséquent, il faut que la façon de déployer mène à un environnement stable et reproductible. La robustesse du déploiement l'emporte sur d'autres considérations comme par exemple la vitesse du déploiement.

L'environnement de développement, par contre, doit surtout être rapide à déployer. Rien ne gêne plus l'intégration d'un nouveau développeur qu'un environnement qui ne peut être déployé de façon efficace et autonome. Enfin, il faut que ce déploiement ait un impact minimal sur l'environnement de travail dans lequel il est installé et il doit pouvoir effacé sans laisser de trace. Proposer une solution facile à déployer et à supprimer de la machine permet non seulement d'intégrer facilement les nouveaux arrivants, mais peut aussi encourager des contributeurs occasionnels.

4.1 Déploiement actuel

Le manuel propose deux manières de déployer GeoNature. La première [46] (dite « installation globale ») déploie une version « complète » qui contient non seulement GeoNature, mais aussi TaxHub et UsersHub. La version minimale [47] (dite « installation autonome »), quant à elle, ne contient que l'installation de GeoNature. En plus de ces deux façons décrites dans le manuel, un déploiement à l'aide de Docker Compose [48] est en cours de développement. Ce dernier étant encore « en cours de développement » au moment de cet audit, nous ne l'évoquerons que très brièvement à la fin du chapitre.

L'installation globale aussi bien que l'installation autonome s'appuient sur un certain nombre de scripts Bash. L'analyse détaillée des scripts de déploiement sortirait du cadre de cet audit, nous recommandons cependant l'excellent outil d'analyse statique de scripts Bash ShellCheck [49].

La principale différence entre ces deux installations, outre l'installation de TaxHub et UsersHub, est que lors de l'installation globale, un script Bash (`install_all.sh`) s'occupe de lancer les autres scripts Bash (`install_db.sh` et `install_app.sh`) dans le bon ordre et gère l'installation des dépendances tandis que lors de l'installation autonome un plus grand nombre d'opérations manuelles sont nécessaires. Ces deux déploiements ont évidemment pour

46. http://docs.geonature.fr/installation-all.html
47. http://docs.geonature.fr/installation-standalone.html
48. https://github.com/PnX-SI/GeoNature-docker
49. https://github.com/koalaman/shellcheck

Tous droits réservés au Muséum National d'Histoire Naturelle

but premier de mettre en place une instance de GeoNature. Bien que l'installation autonome semble un peu plus orientée vers les développeurs, elle part également du principe que GeoNature s'installe dans un environnement qui lui sera dédié. Ainsi, les scripts d'installation vont venir faire des modifications assez conséquentes dans l'environnement (par exemple, modifier la configuration de PostgreSQL). Pour l'instant, il n'existe donc pas de façon de déployer une instance « jetable » qui ne laisse pas de trace sur le système (il est à noter que le déploiement par Docker vise à combler ce manque).

Le déploiement d'une infrastructure complexe par scripts Bash a plusieurs inconvénients. Après avoir exposés des remarques générales sur cette méthode, nous analyserons plus précisément des points particuliers au projet GeoNature.

Il existe des bonnes pratiques pour l'écriture des scripts Bash mais nous n'en connaissons pas dédiées spécifiquement aux scripts de déploiement. Comme de tels scripts ont pour but de changer le système sur lequel ils sont exécutés, ils ont souvent besoin de droits élevés pour cela (e.g. droits admnistrateur). Il est donc impératif qu'ils soient écrits d'une façon sécurisée (est-ce que toutes les commandes n'ont que les droits dont elles ont réellement besoin ?), fiable (est-ce que le système est dans un état stable si le script échoue à mi-chemin ?) et que leurs résultats soient reproductibles. De plus, il faut que les scripts soient adaptables à des changements pouvant intervenir dans l'environnement pour lequel ils ont été écrits. Écrire des scripts qui répondent à toutes ces exigences n'est pas évident.

Dans le cas de GeoNature, les trois scripts principaux (`install_all.sh`, `install_app.sh` et `install_db.sh`) sont évidemment écrits en supposant que l'environnement est réservé exclusivement à l'application de GeoNature. Le script `install_db.sh`, par exemple, adapte le fichier de configuration de PostgreSQL aux besoins du projet, un service qui est potentiellement partagé avec d'autres applications. En tant que développeur, il n'est donc pas possible d'utiliser ces scripts de déploiement pour installer une instance isolée de GeoNature sur son poste de travail.

De plus, il n'existe que peu de contre-mesures pour le cas où l'un des scripts serait lancé à plusieurs reprises. Par exemple, dans plusieurs cas, le script `install_all.sh` modifie des fichiers de configuration en y ajoutant des lignes. Si ce même script est lancé de nouveau, ces mêmes lignes vont être ajoutées une autre fois dans les fichiers en question. Bien que cet exemple particulier ne semble pas gêner l'installation, il montre le problème. Ce problème se pose surtout si l'exécution d'un script échoue. Par exemple, le script `install_db.sh` télécharge à plusieurs moments des fichiers à importer dans la base de données ; que se passera-t-il si la connexion réseau tombe au milieu des téléchargements ? Est-ce que la base sera dans un état stable si le script est relancé ? Bien que les scripts visent à « généraliser » l'installation, les soucis décrits ci-dessus peuvent mener à des déploiements individuellement très particuliers.

L'installation globale comme l'installation autonome demandent beaucoup de manipulations manuelles avant de pouvoir lancer les scripts d'installation. L'objectif de ces manipulations semble être de créer un environnement (mettre à jour la liste des sources, créer un utilisateur dédié) dans lequel les scripts pourront être lancés. Toute opération à réaliser manuellement par l'utilisateur peut introduire des erreurs. Bien que la documentation fournisse toutes les informations nécessaires, elle est parfois difficile à suivre.

Avant de conclure cette section, nous voudrions faire une remarque concernant les bonnes pratiques. Bien que ce ne soit pas une règle universelle, l'avis général est qu'il est préférable de lancer un script avec `sudo` pour lui donner explicitement les droits administrateurs, et d'abandonner ces droits pour les commandes qui n'en ont pas besoin. Cela rend plus visibles les droits qui sont nécessaires au script et donne plus d'autonomie à l'utilisateur.

4.1.1 Conclusions

La conclusion la plus importante de l'évaluation de l'existant est que ni l'installation globale ni l'installation autonome ne permettent à un développeur de mettre en place un environnement « jetable » dans lequel il pourra facilement tester ses développements. Même le déploiement par Docker dont le but semble être de mieux séparer l'application GeoNature de son environnement est plus orienté vers le déploiement d'une instance que vers la création d'un environnement de développement. Ne pas pouvoir lancer facilement les tests Python en arrivant sur le projet est d'autant plus problématique que les tests automatisés lancés par Travis CI ne sont plus en usage actif.

Une des conséquences de ce manque d'environnement de développement peut être que des développeurs occasionnels ne vont pas tester leurs contributions, ou qu'ils préfèreront ne pas contribuer du tout, du fait que la mise en place d'un environnement isolé sera considérée comme trop longue et trop compliquée (par exemple mettre en place une machine virtuelle pour y installer GeoNature). Quant aux développeurs « nouveaux arrivants » sur le projet, ils risquent de se créer un environnement de développement personnalisé, adapté à leur poste de travail. D'expérience, cela va rendre difficile la reproduction des erreurs ou la détermination de leurs origines.

 Tous droits réservés au Muséum National d'Histoire Naturelle

Il nous semble donc prioritaire d'ajouter le déploiement d'un environnement de développement « jetable » minimal permettant de facilement lancer les tests Python. Dans un deuxième temps, il faudrait viser à améliorer et à maintenir la qualité des scripts de déploiement ou à les remplacer par un système de déploiement automatique, comme par exemple Ansible [50].

4.2 Propositions

Enfin, pour conclure ce chapitre, nous exposons ici brièvement des options existantes pour traiter les problèmes exposés ci-avant.

4.2.1 Amélioration de l'existant

L'amélioration des scripts de déploiement et l'amélioration de la documentation est la solution la moins coûteuse à court terme. Pour améliorer les scripts de déploiement, deux pistes possibles sont :

- faire le ménage dans les scripts à l'aide d'un outil comme ShellCheck [51],
- réduire les sources d'erreurs, par exemple en téléchargeant tous les fichiers nécessaires au début d'un script.

La documentation oscille un peu entre donner des instructions point par point et expliquer les étapes à suivre dans des séquences de texte. Ces deux styles ont chacun leurs avantages et inconvénients, mais il serait préférable que seulement l'un des deux soit utilisé.

4.2.2 Déploiement par Docker et Docker Compose

Un déploiement par Docker / Docker Compose est en cours de développement [52]. Tout comme l'installation globale et l'installation autonome, ce déploiement a pour but premier de déployer une application GeoNature. Nous avons essayé sans grand succès de le modifier pour uniquement lancer les tests Python. Le point d'attention le plus important pour ce déploiement est donc d'éviter de reproduire les soucis déjà existants, et d'en faire un déploiement qui soit plus maniable par les développeurs.

4.2.3 Déploiement automatique

Bien que ce ne soit pas un point d'intérêt particulier du projet en ce moment, il est à noter qu'un déploiement par un outil de déploiement automatique, comme Ansible [53], déjà cité, ou Salt [54], permettrait de gérer quelques uns des problèmes mentionnés plus haut :

- définition de l'environnement dans lequel l'application devrait être déployée,
- gestion des fichiers de configuration,
- gestion des droits,
- reproductibilité.

50. https://docs.ansible.com/ansible/latest/index.html
51. https://github.com/koalaman/shellcheck
52. https://github.com/PnX-SI/GeoNature-docker
53. https://docs.ansible.com/ansible/latest/index.html
54. https://github.com/saltstack/salt

Tous droits réservés au Muséum National d'Histoire Naturelle

Tous droits réservés au Muséum National d'Histoire Naturelle

CHAPITRE 5

Gestion des migrations grâce à Alembic

Le projet Alembic [55] est un outil en Python permettant la gestion des migrations d'une base de données. À chaque fois que les évolutions de l'application nécessitent une modification de la structure de la base de données, un script de migration permet d'effectuer ces modifications.

L'utilisation de ce type d'outil a plusieurs avantages. Tout d'abord, la structure de la base de données peut évoluer sans être reconstruite *ex nihilo*. Il est donc possible de mettre à jour des bases existantes tout en conservant les données qu'elles contiennent. C'est un avantage notamment lorsque l'application qui doit évoluer est déjà déployée en production. Un autre avantage est la gestion des versions de la base de données. Alembic intègre une gestion précise du versionnement de la structure de la base de données, ce qui permet de savoir très exactement dans quel état elle se trouve. Cette gestion est montante et descendante, c'est-à-dire qu'il est aussi possible de revenir en arrière dans les modifications si celles-ci n'ont pas supprimé de données.

5.1 Utiliser Alembic dans GeoNature

Lorsque nous comparons ce fonctionnement avec ce qui existe déjà dans le projet GeoNature nous remarquons beaucoup de similitudes. Aujourd'hui, les modifications de la base de données passent par des scripts de migration qui ont un nom spécifique permettant de savoir les versions qui sont impactées par ces scripts. Ce fonctionnement est similaire à celui d'Alembic. Néanmoins, la seule référence pour déterminer quelles migrations doivent être exécutées est définie dans les noms des fichiers de migration. Bien que cela soit fonctionnel, il est possible que cela aboutisse à des erreurs si un des fichiers n'est pas bien nommé. De plus, il est nécessaire que la base de données suive la version de l'application GeoNature, mais il n'existe aucune référence à la version nécessaire dans la base de données elle-même. Si les scripts ne sont pas exécutés en même temps que les mises à jours de l'application il est possible de se retrouver avec des conflits entre la base de données et la version de l'application.

Pour ces raisons, il serait intéressant d'utiliser Alembic pour la gestion des migrations de la base de données de GeoNature. Par ailleurs, Alembic est lié à SQLAlchemy [56] qui permet un lien direct entre les migrations et le modèle de données. Cela permet de générer automatiquement les fichiers de migration nécessaires lorsque le modèle SQLAlchemy est modifié. L'application GeoNature utilisant SQLAlchemy comme ORM, il est tout à fait possible d'exploiter ce lien, ce qui simplifiera l'écriture des scripts de migration. La surcouche Marshmallow [57], au-dessus de SQLAlchemy, utilisée dans le projet ne posera pas de problème puisque le modèle initial est défini avec SQLAlchemy.

55. https://alembic.sqlalchemy.org
56. https://www.sqlalchemy.org/
57. https://marshmallow.readthedocs.io/en/stable/

Tous droits réservés au Muséum National d'Histoire Naturelle

5.2 Passage à Alembic

Pour le passage à Alembic sur le projet, deux solutions sont possibles. Tout d'abord il est possible de garder les migrations existantes et de n'utiliser Alembic qu'à partir de l'état actuel de la structure de la base de données. Le premier script de migration sera un peu conséquent puisqu'il prendra en compte toute la structure d'un coup dans un seul script de migration. Ce sera l'équivalent d'une initialisation de la base de données. L'autre solution serait de transformer tous les scripts de migration actuels en scripts Alembic. Ce travail pourrait, en partie, s'automatiser, néanmoins il sera très probablement nécessaire de modifier certaines choses dans les scripts pour s'assurer qu'ils peuvent s'exécuter correctement. De plus, il serait très compliqué d'automatiser les scripts de descente dans les migrations. Effectivement, pour générer les scripts de montée de version Alembic, il pourrait être envisagé de prendre les scripts actuels pour les mettre dans la fonction « upgrade », mais il n'est pas possible de générer automatiquement l'inverse du script déjà écrit. Ce travail serait donc assez fastidieux, bien que les scripts de descente de version ne soient pas indispensables pour l'utilisation d'Alembic.

Le choix entre ces deux solutions pourra se faire en fonction des besoins de migration des bases. S'il est intéressant de garder la possibilité de migrer une ancienne base de données dans la dernière version de GeoNature, alors il sera intéressant de transformer les anciens scripts. Sinon ne gérer des scripts Alembic qu'à partir de l'état actuel de la base de données est probablement la solution la plus simple. De plus, un script Bash pourrait être envisagé pour passer la base de données dans l'état actuel avant de basculer sur l'utilisation d'Alembic pour les migrations futures.

5.3 Problèmes à éviter

Nous avons vu que l'utilisation d'Alembic a beaucoup d'intérêt dans la gestion des modifications de la structure de la base de données. Néanmoins, plusieurs points peuvent poser problème et demandent une attention particulière. Tout d'abord, il est tentant d'utiliser l'ORM de SQLAlchemy (ou Marshmallow) pour écrire les scripts de migrations Alembic, cela peut cependant poser des problèmes dans certains cas, par exemple, si on ajoute une nouvelle colonne sur une table et s'il est nécessaire d'initialiser sa valeur en fonction des valeurs d'autres colonnes. En effet, le modèle SQLAlchemy va se trouver dans la nouvelle version (avec la nouvelle colonne) alors que la base de données est encore dans un état intermédiaire (sans la nouvelle colonne), le temps que les scripts soient exécutés.

Prenons un exemple simple. Si nous effectuons deux migrations sur une table `Person`, l'une pour rajouter la date de naissance et la seconde pour ajouter l'âge, qui sera calculé en fonction de la date de naissance, le modèle final, qui est présent dans le code, contient les champs `age` et `birthdate` avant que la base de données n'ait effectivement ces colonnes. Donc, dans le script de migration qui va calculer l'âge, si nous demandons de récupérer toutes les personnes, SQLAlchemy essaiera de récupérer le champ `age`, qui est présent dans le modèle, mais cela provoquera une erreur puisque ce champ n'est pas encore disponible dans la base de données.

Pour contourner ce problème, deux solutions sont possibles. La première est d'utiliser uniquement des requêtes SQL afin de s'assurer que les requêtes exécutées prennent en considération l'état courant de la base de données. Une deuxième solution est de redéfinir, dans le script de migration, la sous-partie du modèle SQLAlchemy nécessaire pour la migration et d'utiliser ce modèle dans le script. Par exemple ici, redéfinir la classe `Person` contenant le champ `birthdate` mais pas le champ `age`. Utiliser ce modèle temporaire permettra d'assurer un modèle cohérent avec l'état de la base de données lors de l'exécution du script.

Un second problème pouvant intervenir concerne l'utilisation des scripts de redescente de version. Les scripts Alembic comportent une fonction `upgrade` pour monter en version et une fonction `downgrade` pour descendre. Cette fonctionnalité, bien qu'intéressante, peut aussi apporter son lot de problèmes. Il est important de faire attention à la gestion des données et à la potentielle perte d'information entre ces étapes. Par exemple si nous faisons un script qui supprime une colonne, le script de redescente permettra de rajouter la colonne. Néanmoins les données qui étaient contenues dans la colonne seront, perdues même si le script de redescente est exécuté. Il faut avoir conscience qu'Alembic permet de manipuler la structure de la base de données, mais il ne versionne en rien les données qui y sont stockées. Le passage d'une version à une autre peut donc avoir quelques limitations et il est important de faire attention lorsqu'une migration est appliquée sur la base de données de production.

 Tous droits réservés au Muséum National d'Histoire Naturelle

CHAPITRE 6

Conclusion

L'audit que nous avons mené dans le cadre de cette étude a porté à la fois sur la qualité du code logiciel de GeoNature, et sur l'intégration continue mise en place dans le projet. Le code logiciel est séparé en deux parties : une partie dite *frontend* écrite en JavaScript avec la bibliothèque Angular et exécutée dans le navigateur Web, et une partie dite *backend* écrite en Python et exécutée sur le serveur. Des conseils ont également été émis vis-à-vis de l'utilisation de la bibliothèque Alembic dans le projet.

On trouvera ci-après un résumé des constatations et des préconisations du rapport d'audit avec à chaque fois, un lien vers les sections du rapport où se trouvent les analyses détaillées.

6.1 Mise en forme et formatage du code source

Afin de faciliter la lecture et la maintenance du code logiciel, il est courant d'imposer des règles précises de mise en forme du code (une seule instruction par ligne, indentation lorsqu'on entre dans un bloc, etc.) Divers outils existent pour vérifier le respect des règles et, éventuellement, les appliquer automatiquement dans les cas simples.

Pour Python, Geonature utilise les outils Flake8, Pylint et Black. Il apparaît cependant que leurs configurations ne sont pas compatibles ; nous suggérons donc d'adapter les configurations Flake8 et Pylint pour éviter ces incohérences. Un nombre important d'erreurs demeurent dont on trouvera la liste complète dans les fichiers fournis en complément de ce rapport.

Ces erreurs sont liées, pour l'essentiel, à l'absence de procédures permettant de forcer le respect des règles de style. Nous proposons donc, d'une part, d'inciter les développeurs à utiliser des greffons (*plugins*) dans leurs éditeurs de code afin d'appliquer directement les règles de mise en forme et, d'autre part, de définir un *pre-commit hook* (fonction de rappel exécutée avant l'enregistrement des modifications dans l'entrepôt de code source) qui lance les outils de vérification de la mise en forme et bloque l'enregistrement des modifications en cas de problème. Par ailleurs, on pourrait intégrer l'outil Flake8 dans les tests exécutés par la plateforme d'intégration continue (*CI*).

L'outil Black ne fonctionne pas correctement actuellement en raison d'une incompatibilité de version du module *Click*. Nous recommandons donc d'utiliser dans le projet une version plus récente de cette bibliothèque. Comme pour Flake8, nous proposons d'utiliser un *pre-commit hook* et d'intégrer Black dans les tests de la plateforme d'intégration continue.

Nous conseillons aux équipes en charge du projet de GeoNature de convenir de configurations à recommander aux développeurs extérieurs.

Par ailleurs, nous avons utilisé l'outil Radon d'analyse de la complexité du code. Celle-ci est raisonnable, toutefois les dossiers `data/scripts` et `contrib` pourraient faire l'objet d'une refonte. Une analyse manuelle confirme qu'il serait judicieux de reprendre ce code.

Tous droits réservés au Muséum National d'Histoire Naturelle

6.2 Documentation

Les chaînes de documentation Python (*docstrings*) permettent de documenter les divers éléments du code source (modules, classes, fonctions, etc.) Elles ne sont pas utilisées de façon conséquente dans GeoNature et ne semblent pas respecter un style cohérent. D'autre part, les tests automatiques ne sont pas, ou sont très peu, documentés. Cependant, ajouter des chaînes de documentation des différents cas de tests permettrait de mieux comprendre leurs objecttifs et ainsi de mieux juger de l'exhaustivité des tests.

Côté PostgreSQL, une documentation existe mais elle manque de cohérence. Ce sont parfois des commentaires dans le code source, parfois des utilisations de la commande `COMMENT`. Il serait souhaitable de définir une stratégie unique, puis de l'appliquer à l'ensemble de la partie PostgreSQL. D'autre part, une grande part du manuel est aujourd'hui obsolète ; le mettre à jour serait judicieux.

6.3 Tests

L'intégration continue est l'ensemble des processus qui permettent d'effectuer des opérations d'intégration du code depuis la sauvegarde des changements jusqu'à la mise à disposition d'une nouvelle version installable, à chaque modification du code source dans l'entrepôt de référence. Par le passé, le projet Geonature a utilisé la plateforme Travis CI pour mettre en œuvre cette intégration continue. Aujourd'hui, elle n'est plus utilisée et rien ne semble prévu pour lancer les outils de gestion de qualité logicielle (Flake8, Pylint, Black, Pytest) de façon structurée.

Nous proposons d'utiliser l'outil d'automatisation Tox dans un environnement virtuel contrôlé. Nous avons d'ailleurs suggéré un *fichier de configuration* (page 13) Tox pour le projet.

Concernant la partie *frontend* (code Javascript dans le navigateur Web), les tests Jasmine et Karma ne semblent pas avoir évolué récemment et ne sont plus compatibles avec l'API actuelle. D'autre part, il faudrait expliciter certains imports implicites. Enfin, et c'est le point le plus ennuyeux, les tests consistent uniquement à vérifier que le composant a bien été instancié et ne vérifient aucune fonctionnalité. La couverture de test est donc très faible. Par conséquent, nous conseillons de mettre en place, petit à petit, des tests unitaires et des tests d'intégration. Dans ce rapport, nous détaillons la bonne utilisation des tests et proposons de discuter des critères d'acceptation avec tous les acteurs impliqués dans le projet. Enfin, nous suggérons d'organiser le travail en mode *dirigé par les tests* (TDD ou *Test-Driven-Development*).

Tout comme pour le code Python, il faudrait que les tests de la partie *frontend* soient lancés par l'intégration continue.

6.4 Tests automatiques et intégration continue

Lors de notre analyse de l'intégration continue avec Travis CI , nous avons relevé un certain nombre d'erreurs et d'incohérences. Nous avons donc proposé diverses *améliorations* (page 16), notamment pour mieux identifier l'origine des erreurs en séparant les *étapes* (*steps*) et les *processus de travail* (*jobs*). Si la séparation du *frontend* et du *backend* est correctement mise en place, il est possible de lancer les tests en parallèle. La génération de la documentation *Compodoc* pourrait également être intégrée à l'intégration continue, et ainsi être publiée dans des pages *GitHub* sur la forge logicielle.

Nous proposons dans ce rapport plusieurs options pour gérer la situation créée par la *fin de vie de Travis* (page 16).

 Tous droits réservés au Muséum National d'Histoire Naturelle

6.5 Déploiement automatique

Nous avons constaté que ni l'installation globale, ni l'installation autonome ne permettent à un développeur de mettre en place aisément un environnement « jetable » dans lequel il pourra tester ses développements. Aujourd'hui, les scripts d'installation sont prévus pour l'installation d'un serveur dédié à Géonature et vont réaliser des modifications dans la configuration globale du serveur.

Ne pas pouvoir lancer facilement les tests en arrivant sur le projet est d'autant plus problématique que les tests automatisés de la plateforme d'intégration *Travis CI* ne sont plus en usage actif. Il nous semble donc prioritaire d'ajouter au projet un environnement « jetable » minimal pouvant être facilement déployé pour lancer les tests et développer de nouvelles fonctionnalités.

Dans ce rapport, nous proposons plusieurs pistes pour *améliorer* (page 21) l'existant. La solution du *déploiement via Docker et Docker Compose* (page 21) est évoquée, ainsi que d'autres solutions via des outils de déploiement automatique comme Ansible ou Salt.

6.6 Alembic

Enfin, une section du rapport est dédiée à l'outil Alembic, lié à SQLAlchemy, qui permet de mettre en place des migrations de données lors des évolutions du modèle de données. Ce point est, en effet, un des problèmes auquel est confronté le projet aujourd'hui, et l'outil Alembic a déjà été identifié comme une solution possible.

Nous proposons *deux solutions* (page 24) pour le passage à Alembic, et nous attirons l'attention sur quelques problèmes possibles ainsi que sur la meilleure façon de les contourner.

Tous droits réservés au Muséum National d'Histoire Naturelle