SpaceFox

Zeste de savoir

J'ai porté une lib Python en Java

22 mars 2019

Table des matières

1. Introduction

J'ai porté une bibliothèque Python en Java . Maintenant que la v1.0.0 est sortie, je vais revenir un peu sur le pourquoi de cet exercice, et ce que j'en ai tiré.

2. Pourquoi ?

Déjà parce que je trouve que cette bibliothèque manquait à Java (et à tous les langages à JVM du même coup).

Parce qu'au cours de ma carrière, j'ai très majoritairement participé à des projets déjà existants ; et quand j'ai créé des projets, ils étaient destinés à rester internes à l'entreprise ou au client, avec tout ce que ça implique en terme de procédures, de (non-)test et de (non-)documentation.

Ça a donc pour moi l'occasion d'expérimenter ce que ça pouvait donner de faire une lib de A à Z, avec tout ce que ça implique :

— Une API si possible agréable à utiliser
— Une documentation
— Une couverture de tests corrects
— Déployer une bibliothèque sur un dépôt Maven public pour qu'elle soit disponible via les moyens standards
— Si c'est compliqué de réécrire une bibliothèque depuis un langage aussi différent de Java

3. Portage Python → Java

En fait, le portage du code Python en Java a été *de loin* la partie la plus simple. D'accord, le code n'était pas spécialement compliqué à la base d'un point de vue algorithmique. Je ne saurais pas dire s'il était très *pythonique* mais je ne me rappelle pas avoir spécialement galéré à comprendre quoi que ce soit.

Les deux points qui m'ont donné le plus de fil à retordre sont :

3.1. La gestion d'Unicode par Java

Java utilise UTF-16 comme représentation interne , mais date de l'époque où il n'existait que le Basic Multilingual Plane (« plan 0 »), donc avec un type `char` de 16 bits.

> **Aparté technique**
>
> Unicode définit des caractères abstraits numérotés selon des points de code (« code point »), et ces points de code sont rangé dans 17 plans . Chaque point de code a une représentation dans les différentes normes UTF (généralement UTF-8 et UTF-16).
> Or, l'utilisation des plans autres que le plan 0 nécessite d'utiliser des caractères de substitution et deux « mots » UTF-16 de 16 bits chacun. Spécificité qui n'est *absolument pas gérée par le type natif* `char` *de Java*. Donc, toute gestion des caractères dans les plans supplémentaires doit se faire en gérant les caractère de substitution à la main.

Les plans autres que le plan 0 n'étaient qu'une curiosité de linguiste que la plupart des logiciels ignoraient joyeusement… jusqu'à ce qu'on y colle les *emojis* et donc que tout le monde les utilise. C'est exactement la même raison (mais avec UTF-8) qui fait qu'on doit maintenant utiliser `utf8mb4` et non `utf8` dans MySQL/MariaDB.

Cela dit, une fois qu'on a compris l'astuce, c'est pas très compliqué à gérer, c'est juste chiant.

3.2. Les méthodes avec paramètres facultatifs

J'ai voulu conserver l'API, mais certaines méthodes avaient des paramètres facultatifs en Python, chose qui n'existe pas en Java. Il m'a fallu les remplacer par des surcharges de méthodes, donc *plusieurs* méthodes en Java.

Idem avec des types uniques mais qui peuvent être multiples en Java :

- `String` ou `char`
- `Collection` ou tableau

Donc, pour une méthode Python, j'ai entre une et six méthodes Java.

J'aurais pu me contenter de la méthode la plus générique à chaque fois, mais ça aurait obligé certains types, et à spécifier systématiquement tous les paramètres, ce qui n'est pas le plus agréable à utiliser.

3.3. Les déclarations de types

Python permet de retourner des gros dictionnaires. Java ne permet pas ça (enfin, si, on peut toujours renvoyer de grosses `Map`, mais ça ne se fait pas).

Donc il a fallu créer des types, ce qui est tout de suite plus verbeux.

4. La règle des 80/80

C'est une règle empirique qui dit qu'une fois qu'on a le code qui fait 80 % de ce qu'on veut, il reste encore 80 % du boulot à faire. Et j'ai eu tout à fait l'occasion de la vérifier.

Parce qu'une fois qu'on a le code qui fait le gros de ce qu'on veut et les tests unitaires sur ce qu'il était nécessaire de tester pendant son développement, il reste – liste non exhaustive :

— Mettre en place l'architecture de couverture de code.
— Mettre en place l'intégration continue.
— Rendre utilisable le code de génération des données depuis les sources Unicode.
— Protéger les entrées utilisateur.
— Rendre les API agréables à utiliser.
— Tester tout ce petit monde.
— Écrire la documentation (on ne peut pas se contenter de copier la doc Python à l'identique).
— Générer la Javadoc.
— Écrire les pas-tout-à-fait-doc, comme le *README* et la description du projet.
— Choisir la licence.
— Mettre en place l'outil de déploiement Maven.
— Tester l'outil de déploiement.
— Tagguer une version.
— Release une version.
— La communication sur ZdS, LinuxFR…

… et j'en oublie sans doute !

D'ailleurs, parlons un peu de Maven…

5. Rendre sa bibliothèque accessible : bienvenue en enfer

Pour que les utilisateurs de langages à JVM puissent utiliser ma bibliothèque facilement, il est indispensable de la publier sur un dépôt. Dans le cas d'un projet open-source, l'un des plus connus est l'OSS Repository Hosting de Sonatype. Il permet de rendre son code disponible dans Maven Central ~~et JCenter, qui sont les deux dépôts « de base » du monde Java~~[1] et pas dans JCenter (dépôt existe mais est vide, parce que je l'avais créé pour test).

Et c'est là que le bât blesse : on est au niveau 0 de l'ergonomie tout le long de la procédure, c'est catastrophique.

1. Ne me demandez pas pourquoi il y en a deux et quelle est la différence, je n'ai jamais compris.

Je passe sur l'obligation de faire une demande à validation manuelle pour l'ouverture du dépôt, ça permet d'éviter le parasitisme de nom de domaine / package et ça a l'air très fortement industrialisé en sous-main.

Quant au reste…

La moitié de la documentation n'est pas écrite. Elle est en fait planquée dans les vidéo Youtube liées dans la doc. Vidéos qui ne sont pas référencées, donc accessibles que via les liens, lesquels sont dispersés dans toute la doc écrite.

L'interface d'admin est digne du début des années 2000 : c'est microscopique, laid, et surtout très loin des standards actuels d'ergonomie (obligation de rafraîchir la page pour voir la progression d'une tâche).

Le processus est tout sauf clair : théoriquement (je soupçonne qu'on peut zapper des étapes) il faut créer un ticket qui active le dépôt, y mettre un artefact de SNAPSHOT, aller dans l'interface, vérifier, envoyer un artefact de release sur le dépôt de snapshot, « fermer » le dépôt ce qui va provoquer sa validation, si la validation est OK « release » le dépôt, ce qui va provoquer… sa suppression en même temps que la création du dépôt de release (et donc le faire disparaitre de l'interface), puis au bout de 10 minutes ça va être publié, mais pas dans la recherche qui est disponible au bout de 2 heures. Sauf la première fois où il n'y a pas de publication automatique, il faut commenter le ticket pour que ça soit activé à la main.

Ouf.

Mais, le vrai problème est…

5.1. GPG : l'ennemi du développeur ?

Les paquets à disposition du public doivent être signés avec GPG.

Je connais plein de libristes qui ne jurent que par ça. J'ai rarement vu un outil aussi mal foutu. Je ne sais pas tellement à quoi c'est dû, mais en vrac :

- Il y a des tonnes de doc.
- L'immense majorité se contente de donner des lignes de commande à taper sans vraie explication.
- Une bonne partie est complètement obsolète.
- Il y a des problèmes de compatibilité quelque part entre certains types de clés, de mot de passe et le plugin de signature – j'ai pas cherché les détails.
- OSSHR supporte 3 serveurs de clé pour vérifier la clé publique…
- L'un accepte de recevoir les clés, mais l'interface web ne les trouve qu'après un bon quart d'heure.
- Un autre a les 2/3 des pages qui tombent en erreur 502 ou 503.
- Le troisième a une interface digne des années 1990 absolument incompréhensible si tu ne sais pas *exactement* ce que tu fais (j'imagine que si tu es expert, elle est très pratique, tu as tout sous la main)
- Les comportements par défaut de GPG sont douteux (pas d'export des clés)
- Les outils semblent demander aléatoirement des identifiants de clés longs ou courts.
- Les messages d'erreur sont incompréhensibles.

Sans déconner, avant de dire que tout le monde devrait utiliser ça pour sa vie privée, il faudrait peut-être commencer par rendre l'outil compréhensible et utilisable pour le commun des mortels !

6. Conclusion

Et donc, en conclusion : c'était intéressant, comme exercice… mais pas très rassurant sur l'organisation de l'écosystème Java.

Développer une lib et la rendre disponible semble bien plus simple en Python avec Pip, ou en JS avec NPM (certains diront *trop* simple pour ce dernier).

Si vous avez des retours d'expérience sur le développement de code libre et réutilisable dans les standards de votre langage préféré, c'est le moment d'user votre clavier !