SpaceFox

Geste de savoir

La « configuration » par langage dédié (DSL), une invention de Satan

22 septembre 2022

Table des matières

Introduction

Avez-vous remarqué comme les systèmes de configurations complexes, en particulier des systèmes de *build* et de dépendances, sont un musée de choix douteux et pénibles à l'usage?

Je ne parle pas de la pénibilité intrinsèque à la configuration de choses complexe: celle-ci est «normale» si le système à configurer est lui-même complexe. Non, je parle de cette surcouche de complexité apportée par l'outil, qui fait que des opérations *fonctionnellement simples* se retrouvent à être inutilement compliquées quand on tente de les réaliser.

1. Les problèmes avec les langages déclaratifs

Souvent, on trouve des systèmes de configuration complexes (de frameworks entiers par exemple), de build ou de gestion des dépendances qui utilisent des langages déclaratifs. En vrac:

— Des `properties`, qui sont (très) verbeuses et historiquement ne géraient pas Unicode ;
— Du `JSON`, qui est plutôt lisible mais ne gère pas les commentaires, ce qui est vite un problème majeur dans ce genre de contexte ;
— Du `YAML`, qui est lisible mais pénible à écrire avec ses détails de syntaxe et sa tendance à exploser en vol à la moindre type ;
— Du `XML`, qui est très carré mais ultraverbeux et très vite illisible.

2. Un remède pire que le mal : la configuration par langage dédié

Face à ces problématiques, certains ont imaginé que l'on pourrait écrire un langage dédié (ou DSL, abréviation de *Domain-Specific Language*) pour gérer ces configurations. En première approche, ça a deux énormes avantages:

1. Ça donne des fichiers qui sont faciles à lire[1];

2. On peut y mettre du code arbitraire, donc faire des tâches spécifiques complexes sans avoir à *«développer un plugin»* ou quelque chose de ce genre.

Je pense en particulier à deux d'entre eux : Gradle, et les fichiers de configuration de Jenkins (appelés Jenkinsfiles). Les deux sont basés sur le langage Groovy.

Alors, oui, les deux points ci-dessus sont bien remplis. Constatez à quel point il est plus lisible de déclarer une dépendance comme ceci:

```
```

Que comme ceci :

```
```

Surtout qu'en général, on a pas *une seule* dépendance.

Sauf que tout ça, c'est très bien sur de touts petits exemples pour de touts petits projets, sans particularités un peu complexes.

En pratique, **ces DSL ne sont qu'un écran de fumée qui tente de faire croire que des processus complexes sont simples**. Ce qui, *par conception*, ne peut pas fonctionner.

Les points qui posent le plus problème sont les suivants:

— Toute la complexité est masquée derrière une syntaxe et des «outils» qui se veulent simples.
— La partie masquée de la complexité est rendue artificiellement inaccessible, parce que les API mises à disposition sont trop partielles et n'ont pas de points d'extension:
  — Opérations possibles sur des éléments uniques mais pas sur des collections ;
  — Opérations possibles sur seulement un sous-type des possibilités ;
  — Opérations tout simplement impossibles parce que le DSL n'a pas jugé bon de les gérer ;
  — etc.
— Même configurés à fond de leurs capacités, les meilleurs IDE n'ont pas les moyens de donner des conseils pertinents.
— On doit donc se reposer entièrement sur la documentation, qui est souvent mauvaise.
— La notion même de DSL rends complexe le lien entre une déclaration et le code exécuté – ce qui empêche l'exploration pour savoir ce qui va être fait *exactement*.
— On peut mettre du code arbitraire ou presque, mais le *debug* est impossible (notamment les points d'arrêt).
— Rien n'est typé et la syntaxe est trop libérale, on découvre toutes les typo au *runtime*.
— Comme le DSL masque la complexité, plus personne ne comprends ce qui se passe réellement[2]. Conséquence : les plugins sont d'une qualité catastrophique et conçus pour ne gérer que le cas précis de la personne qui les a développés – or ces plugins sont indispensables.
— Pire: les documentations pour développer (ou aider à la maintenance) des plugins sont elles-mêmes mauvaises, ce qui empire le phénomène.

1. J'ai bien écrit «lire» et pas «comprendre», et ce sont deux choses différentes.

— Et j'en passe.

Bref, la configuration par langage dédié, c'est l'archétype de la fausse bonne idée: ça a l'air bien de loin, c'est bien quand on fait juste un test (ce qui permet de *choisir* cette technologie, si tant est qu'on ait le choix)… et en pratique c'est l'enfer dès qu'on a le malheur de sortir des petites cases prévues pour gérer une forme de «standard».

Et c'est comme ça qu'on se retrouve à passer *des heures* à essayer de réaliser des choses pourtant aussi simples fonctionnellement que *«s'identifier sur deux dépôts Docker différents»* ou *«déclencher une intégration continue au push d'un tag Git»*… Avec une dédicace particulière au développeur *officiel* du produit qui refuse l'intégration d'une fonctionnalité ultra-demandée (avec des gens prêts à la développer) avec pour seule réponse *«allez voir cette doc de 2 lignes et utilisez ce plugin pas maintenu depuis 5 ans»*: Satan lui-même n'aurait pas agi autrement.

Conclusion

Est-ce que j'ai écrit ce billet juste pour râler parce que j'ai eu les problèmes sus-cités (entre autres) avec les DSL sus-cités ? Vous n'avez aucune preuve.

N'hésitez pas à ne pas venir expliquer en commentaire que votre langage préféré a fait un meilleur choix que les autres ce qui rends plus beau plus agréable et plus parfait que tous les autres langages, ça n'aurait aucun intérêt. Donnez plutôt vos trucs et astuces pour moins galérer avec ces outils pétés avec lesquels on est coincés, ça, ce serait utile.

Icône : Le génie du mal par Guillaume Geefs, photographie CC BY-SA 3.0 Luc Viatour retaillée.

2. Jeu : demandez à un développeur (qui utilise ces outils) de vous explique le fonctionnement de base de Gradle, ou de webpack (et de ses «versions simplifiées» comme webpack-chain ou webpack-merge). Un conseil, munissez-vous d'un paquet de popcorn.