SpaceFox

8 mois avec Javascript (ES6) et vue.js

21 décembre 2018

Table des matières

1. Introduction

Il y a environ huit mois, mon patron m'a dit : « SpaceFox, on est en dèche de développeurs front, y'a un projet à faire là, est-ce que tu veux le faire ? ».

Comme c'était un projet à démarrer depuis rien, que je ne connaissais pas le javascript moderne[1], et que je ne serais pas coincé que sur cette technologie, eh bien j'ai accepté. Alors, huit mois plus tard, qu'est-ce que je peux en dire ?

> **!** Tout ce qui suit n'est que mon avis personnel et ne prétends ni à l'exhaustivité, ni à l'objectivité !

1. Parce que malgré mes années de développement web, à cause des projets j'ai toujours été coincé sur du bricolage avec jQuery – au mieux – niveau dev front…

2. L'environnement technique

Le but du jeu était de développer des applications en une page (*single-page application*) avec un framework moderne, et ça devait supporter IE11 en mode dégradé, et les navigateurs modernes en fonctionnement nominal. C'est de l'application assez classique, avec des appels à des API REST et une poignée de traitements côté client.

On s'est posés la question de TypeScript, mais ajouter un système de type à un langage qui n'en a pas, c'est pratique tant que tu maitrise ton code de bout en bout. Mais dès que tu appelles une lib externe, tu te retrouves avec des types à rallonge et/ou incompréhensibles. Vues les avancées de ES6, on s'est dit qu'on pouvait se passer de TypeScript. Y'a des jours où je me dis que c'était une bonne idée, d'autres où je pense que c'était une connerie… la vérité c'est qu'à mon avis je me serait fait chier autant mais pas sur les mêmes trucs.

Alors on a monté un projet avec :

— ES6 (et même plus, parce qu'on utilise `async`/`await`)
— Babel pour la compatibilité
— Webpack pour lier tout ça
— Vue.js comme framework avec vue-router pour le routage et vuex pour la gestion d'états type flux (et donc vue-cli 3 pour gérer une bonne partie des aspects)
— Axios pour la communication réseau
— Jest pour les tests unitaires
— Gitlab CI pour l'intégration continue
— eslint en mode paranoïaque pour les bonnes pratiques et le formatage du code (c'est fou ce qu'on peut apprendre en lisant les erreurs de lint)
— Et un gros tas de libs

3. Les bons côtés

3.1. La souplesse de Javascript

Le fait que JS permette de faire un peu tout est n'importe quoi a un côté pratique : souvent, on peut aller au plus court et ça fonctionne. Par exemple, on peut faire des objets parfaitement arbitraires sans jamais déclarer leur structure nulle part, et mine de rien ça fait économiser plein de code.

Je ne vais pas m'étendre là-dessus : JS est très souple, c'est très pratique.

3.2. La programmation réactive

Y'a un article Wikipédia à ce sujet , mais pour résumer : en programmation classique (MVC), on va avoir des structures de données, on va définir des vues à partir de ces données, et programmer les moyens de mise à jour des données et comment les vues sont mises à jour à partir de ces données.

En programmation réactive, on va avoir des données *réactives* et construire des représentations de ces données. Mais pas besoin de programmer la mise à jour des vues à partir des données : le framework réactif garantit que les vues sont à jour.

Concrètement, ça a une courbe d'apprentissage bien raide et demande un découpage du travail très différent d'un framework réactif (définition des données, et dans le cas de Vue.js des composants). Mais une fois le concept maîtrisé, c'est **excellent**, surtout pour les applications qui consistent principalement en de la présentation et de l'édition de données.

3.3. Le pattern flux

Des explications ici avec des liens . Ça rajoute un peu de code par rapport à une utilisation du framework « nue », mais pour moi dès que l'application n'est plus triviale c'est tout simplement indispensable.

Pourquoi ? Parce que la combo programmation réactive + flux force un découplage tel que des évolutions qui seraient casse-pieds sur un framework « classique » deviennent complètement triviales.

3.4. Une communauté énorme et réactive

Il y a une telle communauté JS (forcément, c'est le seul langage disponible en front… et on l'utilise en back aussi maintenant) que si on a besoin d'un coup de patte ou d'une lib, c'est à peu près certain que ça existe.

3.5. `async` / `await`

Plus de callback ! Plus de promesses ! Tu écris ton code asynchrone, et il fonctionne !

3.6. Les performances en *runtime*

Une très bonne surprise : moi qui m'inquiétais de ça, en fait je peux faire afficher des centaines de composants, utiliser des select avec des milliers d'items (et une recherche) ou faire traiter des tableaux de dizaines de millions de cellules avec des performances correctes sans y passer spécialement du temps.

4. Les mauvais côtés

4.1. L'incohérence et l'aspect cryptique du langage

Là non plus je ne m'étendrai pas, il y a des sites entiers sur le sujet, mais en vrac : `===`, la non-commutativité de beaucoup d'opérateurs, la gestion incompréhensible des casts automatiques, `null` et `undefined`…

Plus vicieux : ES6 a ajouté énormément d'éléments très pratiques, mais vient aussi avec son lot de détails incompréhensibles. Exemple : si les variables `a` et `b` sont déjà déclarées et que j'ai un objet `o = { a: 'quelque chose', b: 'autre chose' }` que j'essaie de déstructurer, je ne peux pas faire `{ a, b } = o;` mais je dois faire `({ a, b } = o);`… (je vous laisse trouver la différence).

4.2. Les webworkers

J'en ai besoin parce que j'ai des traitements qui sont gourmands en CPU et que je veux garder une interface réactive (et que JS est mono-thread). Eh ben, c'est le pire système de parallélisme que je connaisse. Surtout couplé à l'absence de types explicites, qui fait que tu découvres seulement au runtime que ton worker ne fonctionne pas parce qu'il y a une fonction dans l'objet que tu essaie de lui passer.

4.3. La souplesse de Javascript

C'est bien, la souplesse. Mais dès qu'on bosse à plus de 1, ou plus d'une semaine sur un même projet, sans un linter paranoïaque, c'est un coup à se retrouver avec des techniques de code très différentes partout dans le projet. Ou du code illisible. Ou les deux.

4.4. La folie furieuse de la communauté

Sans déconner : après huit mois d'observation, ma conclusion la plus logique est que 95 % de la communauté Javascript est composée de macaques sous cocaïne. Je ne vois pas d'autres explication logique à l'état général de l'environnement.

Tout détailler serait beaucoup trop long, mais en vrac :

— Tout change tout le temps sans te laisser le temps de te retourner (Vn+1 en bêta ? On déprécie la Vn !)
— Conséquence : beaucoup de codes cassés, et pratiquement toutes les docs sont obsolètes (quel que soit le sujet).
— Y'a le choix pour tout… tout existe au moins en double, sans que rien de devienne un standard de fait (et ça concerne jusqu'aux règles de formatage de code, ou aux éléments du framework de test – j'ignorais qu'on pouvait découper un framework de tests unitaires en plusieurs sous-blocs)
— Le découpage en femtomodules jusqu'au ridicule infini. Il existe un package pour savoir si un nombre est pair qui se contente de dépendre d'un autre package pour savoir si un nombre est impair et fait un « not » dessus . La première fois qu'on me l'a dit je ne l'ai pas cru. Pire : ce package a connu 4 (quatre) versions, a été téléchargé 19 020 (dix-neuf-mille-vingt) fois la semaine passé et 7 (sept) autres packages en dépendent .
— Les empilements de solutions absconses : ça fonctionne dans le cas nominal et t'explose à la tronche dès que tu essaies de sortir des clous. Exemple ? Les fichiers webpack sont un tel foutoir que même avec webpack-merge on ne s'y retrouve pas ? Pas grave, on va créer un troisième outil nommé webpack-chain qui va garantir que dans le cas nominal c'est plus simple, et que dans absolument tous les autres cas c'est encore moins

compréhensible, parce qu'on doit écrire du code illisible pour générer un fichier de conf illisible (et t'as pas le choix, tout ça t'es imposé par un quatrième outil…)
— Ce genre de réponse .
— Le choix d'un format de fichier (JSON) qui ne gère pas les commentaires pour le fichier de définition des dépendances du projet. Juste ça.
— Les chaines de « Projet A ne me plait pas alors je crée B qui fait la même chose mais en mieux mais depuis A s'est amélioré du coup A et B cohabitent et/ou mergent et du coup c'est le foutoir » (coucou npm et yarn !)

4.5. Les outils

C'est un peu une conséquence de tout ce qui précède. Même avec le meilleur IDE du monde, le langage et l'écosystème est tellement pourri jusqu'à la moelle que des tas de trucs qui devraient fonctionner ne peuvent pas fonctionner. Et donc tu perds un max de temps et d'énergie là où tu ne devrait pas. Exemples ? En vrac avec les IDE JetBrains :

— Impossible d'avoir une autocomplétion correcte : dès que le scope de ta variable n'est plus absolument évident (et encore), l'outil te propose n'importe quoi. La refactorisation peut t'exploser à la tronche pour les mêmes raisons.
— Impossible de trouver une dépendance à cause d'une typo ? C'est probablement pas grave, c'est sans doute un truc résolu dynamiquement, c'est donc un weak warning non signalé et presque invisible. Mais ça plantera au runtime.
— Les versions installés des packages ne sont pas cohérentes avec celles demandées ? Pas grave, personne ne préviens tant qu'on essaie pas d'ouvrir le fichier package.json…
— Le package.json qui ne fixe pas des versions précises, tâche déléguée au package-lock.json… qui ne les fixe pas non plus.
— npm qui fait des modifications random dans ce dernier fichier (« et si je changeais la moitié des https en http ? »).
— Les surprises à la MàJ de Node .
— Les formats différents de packages JS. Le pire étant une lib qui n'est distribuée qu'en plugin jQuery.

4.6. Les performances en *développement*

Bordel mais que font les outils pour que la compilation soit aussi lente et nécessite autant d'entrées/sorties disque ? Le développement JS sans SSD, c'est devenu virtuellement impossible.

5. Conclusion

6. En conclusion…

J'ai découvert des tas de trucs ces huit derniers mois. J'ai appris des frameworks et des technologies très intéressantes, qui ont objectivement fait de moi un programmeur meilleur

et plus complet. J'ai aussi une bien meilleure compréhension du développement web front moderne.

Je sais aussi maintenant que, de mon point de vue, on peut enfin faire des projets intéressants et s'amuser avec du JS en front (ce qui était strictement impossible avant pour moi). Vue.js, c'est un framework que j'aime bien (et y'a un an, je ne pensais pas dire ça un jour d'un framework JS).

Par contre, c'est clairement **malgré** Javascript et **malgré** 90 % de l'écosystème, qui sont des plaies purulentes qui ne mériteraient que la purification par le feu. Et ça fait que maintenant je sais que je ne toucherai pas à du JS côté serveur, même avec une pelleteuse de mine.