Rapport d’architecture

SOA

TEAM A
Younes Abdennadher – Thomas Colomban –
Alexis Lefebvre – Alexandre Longordo

Table des matières

I) Choix faits par rapport aux stories

A) Interactions des acteurs avec le système

1) Le sondage avant le lancement

Le sondage créé avant la mission est typiquement une action que l'utilisateur doit effectuer sur le système. Le choix du lancement du sondage dépend de la décision du chef de mission et ne peut pas être décidé automatiquement.

Pour ce qui est du vote dans le sondage, il était possible de le rendre automatique en fonction du statut de la météo et du statut de la fusée. Mais l'approbation de la mission passe par des chefs de départements différents qui peuvent avoir à prendre en compte des données externes au système pour approuver la mission (le département météo doit pouvoir interpréter les données météo venant d'un service externe par exemple). C'est pour cela que les votes dans le sondage doivent être faits par les utilisateurs et non automatiquement.

2) Lancement de la fusée

Nous avons estimé que le lancement de la fusée est une tâche importante qui peut dépendre de données externes au système. Elle ne peut donc être automatisée. Une action du chef du département de la fusée est donc requise pour lancer la fusée. Cette action peut prendre place lorsque tous les résultats du sondage sont OK et que d'autres données venant potentiellement de l'extérieur sont propices.

3) Destruction du matériel

De même, pour la destruction de la fusée et du module de propulsion, cette action peut être lourde de conséquence et peut possiblement mettre des vies en jeu. C'est donc une action dont la responsabilité sera déléguée à l'utilisateur. Nous avons choisi de pouvoir détruire indépendamment le module de propulsion et la fusée, car un problème peut arriver sur l'un des deux seulement, même lorsqu'ils sont séparés et que le module de propulsion est en phase d'atterrissage. Il serait alors peu pertinent de devoir détruire l'ensemble des appareils volants pour un problème survenant uniquement sur l'un des deux.

B) Actions automatiques du système

1) Séparation du booster

Le booster est un service embarqué sur la fusée, il connaît donc son propre état sans aucun délai. Afin d'optimiser la séparation et de gagner du temps par rapport à un déclenchement manuel, nous avons décidé de l'automatiser. Dès que des critères donnés sont remplis (en l'occurrence quand le booster n'a plus d'essence, la séparation est enclenchée de manière automatique.

2) Livraison de la charge

La charge appartient à une entreprise différente de celle de la fusée, c'est pourquoi c'est cette entreprise qui devrait décider quand est-ce qu'elle livre son module/satellite. De plus il n'y a pas d'urgence quant au déclenchement de la séparation du satellite avec la fusée puisqu'une fois que les données orbitales de la fusée sont stables, elles ne changeront pas. Le chef du département de la charge peut donc livrer la charge quand il le souhaite.

3) Atterrissage du booster

Le booster atterrit de manière automatique. Aucune action humaine n'est nécessaire puisque le booster est le plus à même de calculer le déroulement de l'atterrissage. En effet, les données du booster sont directement accessibles depuis le module sans aucun délai, il est donc plus sûr que le booster calcule seul sa trajectoire à l'atterrissage.

4) Ralentissement de la fusée

Le service de la fusée connaît la pression actuelle qu'il subit. Une automatisation est ici obligatoire car une intervention humaine pourrait, tout comme pour l'atterrissage du booster, intervenir trop tard et donc faire échouer l'opération. La fusée va donc réduire la vitesse de ces moteurs une fois Max Q atteint.

5) Visualisation des données télémétriques

Les données télémétriques sont envoyées automatiquement. Elles sont envoyées régulièrement ou à chaque changement de statut des modules de vol (fusée, booster et satellite) au département de télémétrie qui reçoit donc les informations avec un certain délai. Le système ne base aucune action critiques (séparation du module ou atterrissage du booster) sur les données télémétriques à cause de ce délai. Elles ne sont là que pour que la base au sol soit mise au courant de manière régulière des paramètres de vol.

II) Description de l'architecture

A) Diagramme d'architecture

La figure 1 représente l'architecture actuelle de notre système ainsi que les interactions entre utilisateurs et services.

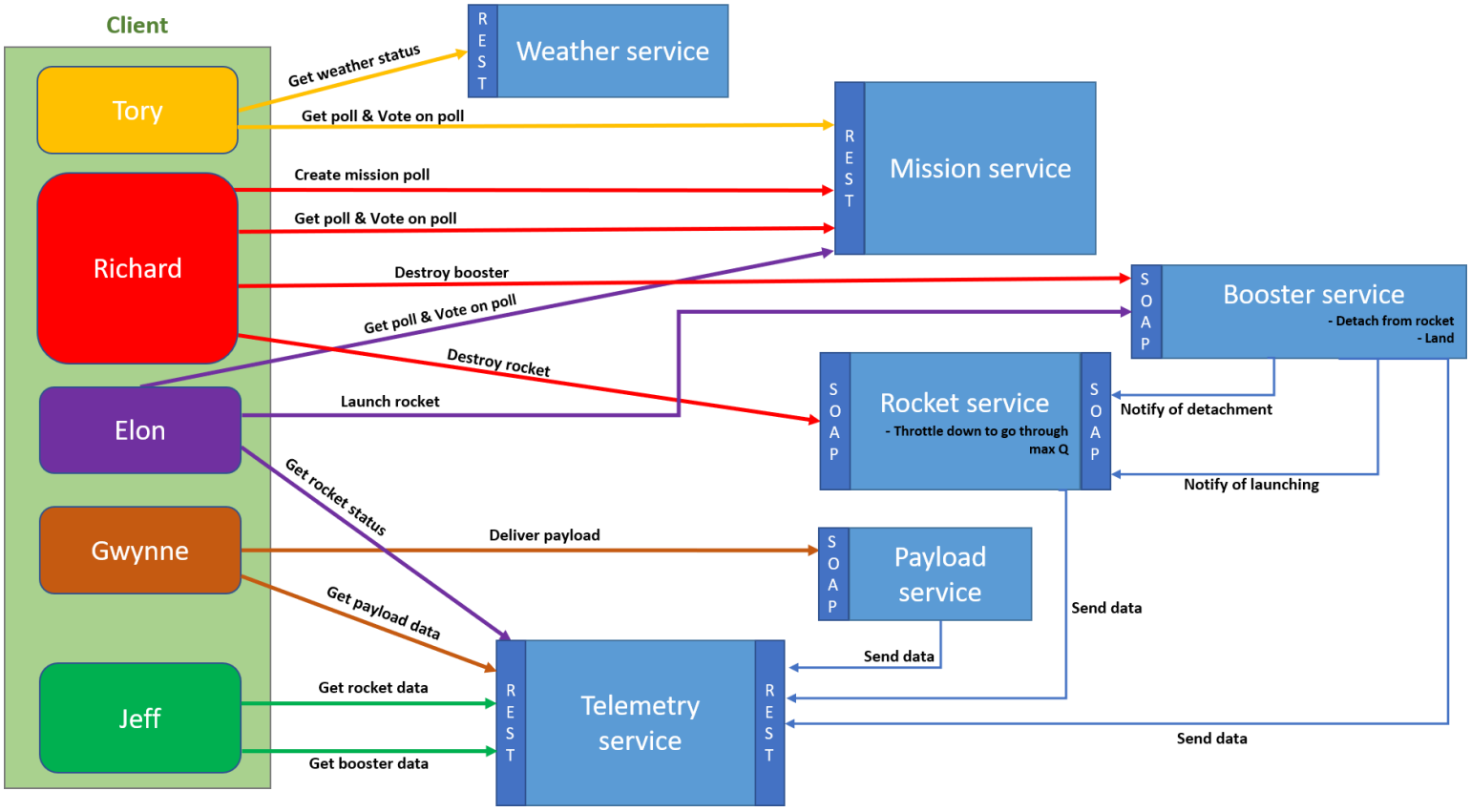

Figure 1: Diagramme d'architecture

Sur ce schéma, on peut voir que notre système possède un client grâce auquel les différents acteurs du peuvent interagir avec le système. Nous avons représenté chaque acteur du sujet dans le client et chaque requête qu'il est capable d'effectuer.

Nous allons détailler les choix faits vis-à-vis de cette architecture.

1) Weather

Weather est un service mocké. Il représente un service externe auquel le chef du département météo fait appel pour avoir des informations sur la météo. Ce service ne renvoie pour l'instant qu'un statut pour simuler le retour d'informations dont aurait besoin l'acteur. Il interprète ensuite le retour et peut voter dans le sondage.

Le service expose une API REST. Nous utilisons ce protocole car l'accès au service n'a pas besoin d'être très codifié. Si les appels aux routes ou si les routes changent, cela ne pose pas de problème vraiment critique puisqu'il suffit de corriger les erreurs sur l'appel à la route avant de refaire un appel.

2) Mission

Le service mission est responsable de la création et de la modification du sondage. Pour cela il contient un objet sondage qu'il peut modifier en fonction des requêtes des utilisateurs. Ce service est donc Stateful pour le moment, il devra être adapté par la suite pour qu'il puisse stocker ses données dans une base de données, et ainsi devenir Stateless. Ce service met à disposition une API REST car il gère des ressources et qu'il n'a pas besoin d'un contrat fort entre lui et le client, au vu du faible niveau de criticité des données qu'il gère.

3) Rocket

Le service Rocket a pour responsabilité de gérer le plan de vol de la fusée et ses données, comme par exemple sa vitesse, son altitude ou son niveau de carburant. Ces données permettent au service d'avoir un aperçu constant et immédiat de tous les paramètres de vol afin que la fusée puisse effectuer des actions automatiquement et sans délai. Elle peut actuellement décider elle-même quand réduire sa vitesse lorsqu'elle subit une certaine pression. La fusée gère donc tout son cycle de vie (sauf lors de la destruction et du décollage).

On peut remarquer que pour accéder aux méthodes de la fusée, un web service SOAP a été mis en place. Les routes vers la fusée étant toutes critiques pour le bon déroulement de la mission, nous avons fait le choix d'instaurer un contrat fort avec les services appelants. Ce contrat permet notamment de s'assurer que les services appelants envoient les bons paramètres à l'appel des routes de Rocket et qu'ils ne peuvent pas appeler une méthode tant qu'elle ne correspond pas exactement au prototype exposé par Rocket.

Lors de la destruction, on ne peut pas se permettre qu'une route ne fonctionne pas à cause d'une erreur dans le nom par exemple. De même, lors du détachement du booster, si le booster ne peut pas notifier la fusée au bon moment à cause d'un mauvais appel pour qu'elle puisse allumer ses moteurs à son tour, la mission échouerait.

4) Payload

Le service Payload à pour responsabilité de gérer le détachement et la mise en orbite de la charge. Payload connaît son altitude ainsi que sa vitesse il est donc autonome pour sa mise en orbite.

Comme expliqué précédemment nous avons décidé de laisser l'humain déclencher le détachement nous avons également choisi d'installer un contrat fort en utilisant SOAP pour le détachement afin d'assurer la cohérence en service et client.

Le service Payload contient actuellement ses propres données (altitude, vitesse) qui sont envoyées au service de télémétrie via un protocole REST.

5) Booster

Le service Booster est responsable du cycle de vie du propulseur de la fusée. Ce service, tout comme Rocket possède ses propres données qui lui permettent d'avoirr un aperçu rapide des paramètres de vol que subit le booster.

La mission reposant en grande partie sur le bon fonctionnement du booster (le décollage, et l'atterrissage), nous avons décidé de mettre en place un web service SOAP dans ce service. Encore une fois, le contrat fort permet de s'assurer que le service appelant ne peut pas être déployé sans que ses appels correspondent exactement aux routes exposées par le service Booster.

Nous avons essayé de limiter au maximum les appels du client vers Booster et de déplacer la logique du plan de mission dans ce service pour minimiser l'erreur humaine. C'est pour cela que les seules interactions du client avec ce service sont des actions qu'il est absolument nécessaire de faire de par une action humaine. Le reste de la logique de vol du propulseur est gérée par le service Booster directement qui est le plus à même d'analyser rapidement ses propres données.

6) Telemetry

Telemetry est un service permettant de stocker et de récupérer les données télémétriques de Rocket, Booster et Payload. Ce service est typiquement le service ayant besoin de persister des données. Actuellement, ce service stocke dans ses entités chaque donnée télémétrique. Il a donc une liste de données pour chaque module de la fusée. Ce service est ainsi potentiellement très vite chargé en données. Nous en parlons plus en détails dans la partie II.C (Gestion des données).

Ce service est donc à modifier en priorité et à rendre *Stateless* une fois que la couche de persistance aura été implémentée.

Une autre limite de ce service réside dans la manière dont les autres services accèdent aux données qu'il stocke

Le service expose une API

B) Choix technologiques

1) Client frontend

Tôt dans le projet, nous avons fait le choix de réaliser un client frontend afin de simuler de manière fiable les actions faites par les utilisateurs. Au départ, il s'agissait d'un client en lignes de commandes, puisque les premières User Stories ne demandaient que des quêtes et des retours utilisateurs simples.

Faire un CLI était certes plus proche du métier puisque le client a une interface concrète avec laquelle il peut interagir de manière intuitive avec des commandes métier, mais nous aurions également pu utiliser des outils tels que Postman ou SoapUI pour simuler une interface. L'utilisation de ces outils pour le client est moins orientée métier puisqu'il faut savoir saisir des routes mais cela permet de gagner du temps sur le développement, en particulier lorsque l'on cherche à produire un produit minimal. Avec du recul, la création du CLI aurait pu être évitée.

En revanche, la décision de représenter le client sous la forme d'une interface graphique est plus discutable et nous allons voir que ce choix est justifié.

Nous avons, dès la lecture des secondes User Stories, décidé de représenter les retours du système sur une interface graphique. En effet, le système se complexifiait, et des interactions entre services commençaient à apparaître. Ces interactions entre services ne sont pas visualisables depuis des outils

tels que Postman ou SoapUI puisqu'ils ne permettent que d'afficher le retour final et sont très peu lisibles via un CLI. De plus, l'introduction de données télémétriques à afficher et dont le suivi doit être fait en temps réel rendait le choix d'une représentation graphique assez logique. Il n'est pas possible de représenter facilement et lisiblement des données télémétriques évolutives et de visualiser le bon déroulement du détachement du booster par exemple sans avoir une interface affichant le déroulement de la mission en temps réel.

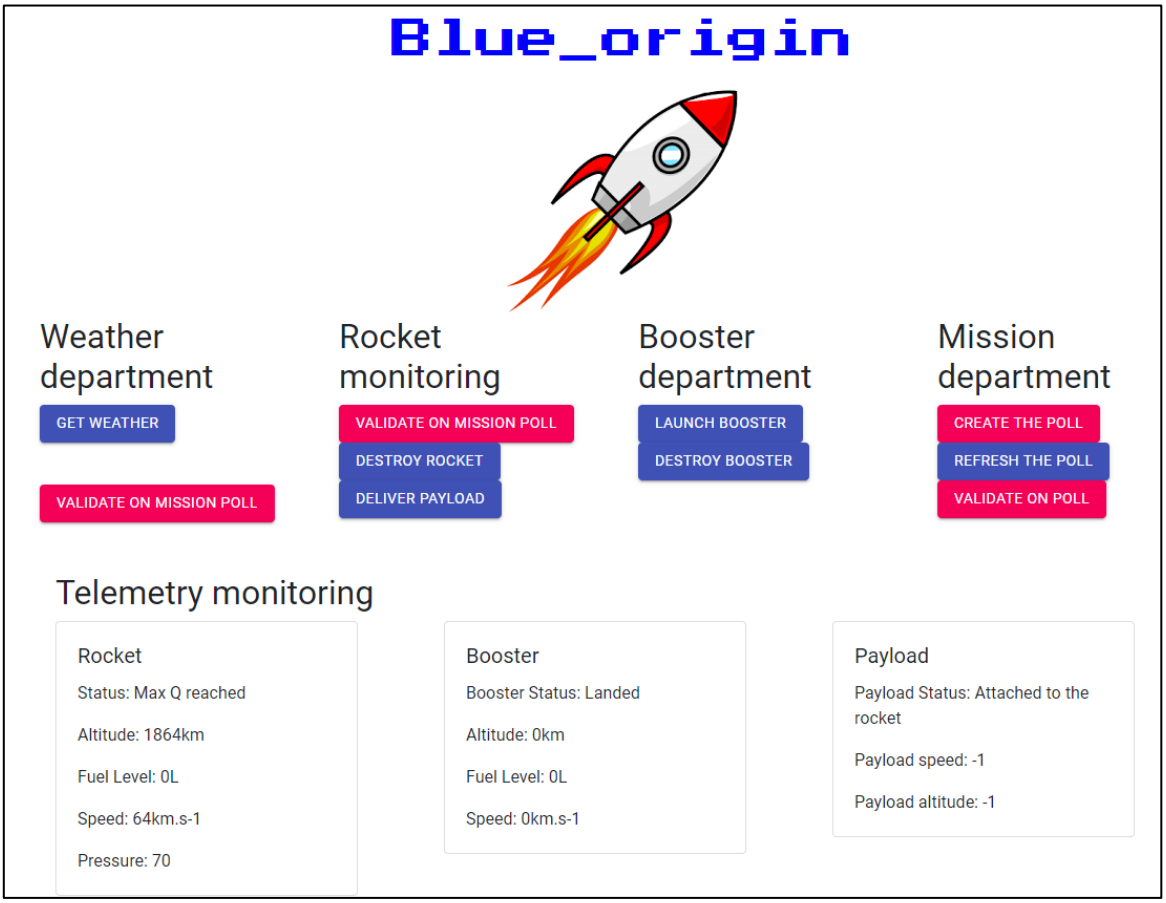

Figure 2 : Interface graphique du client

Sur la figure 2, on peut voir que les utilisateurs ont accès à une interface affichant toutes les données télémétriques en temps réel et leur permettant d'interagir avec le système en fonction de leur métier.

D'autre part, le projet étant voué à être de plus en plus complexifié, il était important, pour le client comme pour nous, d'obtenir des retours des différents service au travers de cette interface graphique.

En ce qui concerne le choix de la technologie, nous avons décidé de choisir une technologie avec laquelle tous les membres sont à l'aise pour ne pas perdre de temps à monter en compétence et avec laquelle on ne perdrait pas de temps à faire des affichages simples. Trois choix s'offraient donc à nous :

- Angular, qui est long à mettre en place et qui est destiné à des projets lourds
- VueJS destiné à des projets légers et facile à mettre en place
- React, qui permet de créer des projets légers rapidement et également facile à mettre en place.

Le choix s'est porté sur React et sur Vue JS qui possèdent des propriétés similaires. Nous avons opté pour React.

2) Backend

a) Nos besoins pour le backend

Nous avons choisi d'implémenter le backend à l'aide de NodeJS et Express.

Certaines contraintes qui nous ont poussés à faire ce choix. Tout d'abord nous avions besoin d'une technologie maitrisée par tous. En effet, du fait de la charge de travail induite par la valeur constante qu'il faut ajouter à chaque sprint, nous ne pouvions pas nous permettre d'utiliser une technologie non maitrisée.

De plus, nous avions besoin d'une technologie permettant la configuration rapide de routes et de micro-services.

Il était également nécessaire d'avoir une technologie similaire entre le frontend et le backend afin de lier facilement les entités du frontend et du backend.

Enfin, la technologie utilisée doit permettre la création de composants, de micro-services et d'entités facilement. Il fallait également avoir en vue l'implémentation d'une couche de persistance.

Les deux premières semaines, le temps gagné par le fait que nous n'ayons pas à apprendre la technologie, a permis de gérer la partie devops du projet : mettre en place des containers Docker, les faire communiquer entre eux et mettre en place l'intégration continue.

b) Technologies étudiées

Voici les technologies qui prennent en compte les contraintes énoncées précédemment :

Java/Spring

Java est le langage que nous maitrisons le plus. De plus Java impose une rigueur dans le code qui permet d'éviter beaucoup d'erreurs et de gagner ce temps.

Ce choix n'a pas été gardé car la manque de connaissance de Spring aurait pu nous ralentir et le temps de création de nouveaux micro-service est plus long. Mais le choix a surtout été fait à cause du temps de mise en place des web-services en Spring.

PHP

PHP souffre d'un manque de cohérence dans les bibliothèques qu'il possède et cela a pour conséquence qu'il existe beaucoup de bibliothèques permettant de faire la même chose de manière différente. Cela demande donc une rigueur au sein du groupe à chaque nouvel ajout afin de se concerter pour définir les bibliothèques à utiliser pour tout le projet.

PHP reste bon lorsqu'il est utilisé par des Framework tels que Laravel ou Symfony car ils abstraient les bibliothèques natives et forcent une structure du code. Cependant ces frameworks sont trop complexes pour notre projet.

Python/Django

Mettre en œuvre un service Django est très rapide et Python est un langage simple. Cela permet de produire beaucoup de fonctionnalités en très peu de lignes de code.

Python demande aussi une rigueur qu'il n'impose pas. Écrire en très peu de lignes de code peut paraitre une bonne source de gain de temps mais le langage reste peu lisible.

Enfin Django n'est maitrisé par personne dans le groupe, ce qui aurait dû entrainer un long apprentissage de la technologie.

Node/express

Le choix Node/express a été retenu. En effet, c'est la technologie la mieux maitrisée de tous après Java et qui rencontre toutes les conditions que nous avons défini.

Il est facile et rapide de lancer un nouveau projet Node pour créer un nouveau service. Node permet aussi d'avoir la même technologie que notre frontend. On peut donc rapidement récupérer les entités du backend vers le frontend.

Node possède également le plus grand magasin de package grâce à NPM. On peut donc installer beaucoup de morceaux de code qui ont déjà été faits.

S'il est nécessaire d'implémenter des systèmes de bus et RPC lors de prochaines User Stories, nous savons que l'on trouvera un module NPM pour le faire. De plus, ce même module sera réutilisable dans la partie frontend.

TypeScript

Le défaut principal d'une solution NodeJS/Express est le manque de rigueur de JavaScript. TypeScript ajoute une couche de rigueur qui permet de vérifier la cohérence de notre code et éviter les mauvais passages de types entre les fonctions.

Aussi, la structure objet de TypeScript ajoute une couche de sécurité et de responsabilité car les notions de privé/public n'existent pas dans les objets JS.

Enfin, TypeScript permet la compilation à la volée dans les IDE et évite les erreurs à l'exécution (comme l'accès à une ressource d'un objet inexistant).

C) Gestion des données

Les user stories nécessitent que certaines données (notamment les données télémétriques) soient stockées. La question de la manière de stocker ces données s'est donc posée. Pour l'heure, tous nos services sauf le service Weather se comportent comme des services *Stateful* et nous ne stockons pas nos données dans une couche de persistance comme une base de données. Dans le cadre de la mise en place d'un produit minimal, nous avons décidé d'omettre la couche de persistance qui peut être pour le moment simulée par des services S*tateful* qui enregistrent leur propre état dans leurs entités.

Nous avons identifié la valeur comme étant dans le bon déroulement du scénario correspondant à l'ensemble des user stories. La couche de persistance était donc moins prioritaire que la couche de présentation dans le client graphique puisque cette dernière permet au client d'avoir un retour direct et intuitif de ce qu'il se passe au sein du système.

Cependant, ne pas avoir de couche de persistance présente des limites non négligeables, et c'est pour cela qu'il sera nécessaire de l'implémenter dans les prochains sprints. Actuellement, de très nombreuses données de vol sont stockées dans les entités du service Telemetry. Si un problème survenait dans ce service, toutes les données télémétriques seraient perdues et la base au sol n'aurait plus aucun retour du matériel en vol.

Lorsque nous implémenterons la couche de persistance, les seuls services qui demeureront *Stateful* sont les services critiques de la mission : Rocket et Booster. Ces deux services ont besoin d'accéder rapidement à leurs dernières données afin de calculer leur trajectoire et d'agir en conséquence.