

SOA INTÉGRATION DE SERVICE

Projet Blue Origin

Groupe J

Paul-Marie DJEKINNOU

Paul KOFFI

Florian AINADOU

Djotiham NABAGOU

11 Octobre 2020 | Département des Sciences Informatiques (AL) 5A

PLAN

Présentation du projet	4
Hypothèses	4
La météo	4
Les fusées	4
Le vote pour le lancement de la fusée	4
Le lancement de la fusée	5
Les données télémétriques	5
Les détachement de la fusée et du satellite	5
La séparation de la fusée en deux parties	5
La livraison de charges	5
Passer le MAX Q	5
Choix de conception et évolution d'architecture	6
Scope 1	6
Description	6
Weather status	7
Rocket status	7
Poll creator et Launch service	7
Scope 2	8
Description	8
Launch service & Rocket system simulation	9
Telemetry service (Socket server) et Delivery service (REST)	10
Scope 3	11
Description	12
Launch service, Rocket system simulation, Payload system simulation et Payload Telemetry Système	12
Conclusion	13

Présentation du projet

BlueOrigin est notre premier projet dans le cadre du cours de SOA Intégration de services. Au travers de ce projet, nous devons produire un système à base de microservices permettant de répondre à un certain nombre de besoins utilisateurs. Aussi, nous avons chaque semaine une nouvelle liste de fonctionnalités à ajouter à notre application existante. Le projet vise à faciliter la gestion de missions spatiales.

I. Hypothèses

1. La météo

Pour la météo, nous avons considéré que l'objet à enregistrer serait le site de lancement de la fusée et que les conditions météorologiques y seraient liées. Les paramètres pris en compte sont notamment le nom du site, la température, la précipitation, l'humidité et la vitesse du vent.

2. Les fusées

Pour ce qui est des fusées, nous conservons comme informations sa disponibilité, son état actuel ainsi que son nom, sa quantité d'essence ou encore sa vitesse.

3. Le vote pour le lancement de la fusée

Nous avons émis comme hypothèse le fait que les étapes de lancement de la fusée soient automatisées. Le vote pour déterminer si une fusée peut décoller ou pas est fait de façon automatique par une analyse de l'état de la fusée et de la météo. Ce vote n'est donc pas dans l'attente de choix de personnes humaines.



4. Le lancement de la fusée

Dans notre système, le lancement de fusée se fait automatiquement dès que les approbations du service de gestion des fusées et du service météorologique sont reçues.

5. Les données télémétriques

Nous sommes partis du principe que les données télémétriques se résument à l'altitude de la fusée dans le temps et que la mission de la fusée dure environs une minute.

6. Les détachement de la fusée et du satellite

Cette étape est également automatisée et se base sur la position de la fusée pour déterminer à quel moment cette étape devrait être entamée.

7. La séparation de la fusée en deux parties

Cette action a elle aussi été automatisée. Elle est pour le moment faite de façon statique en se basant sur le fait qu'arrivée à la moitié de son trajet, les deux parties de la fusée sont détachées. Nous sommes partis du principe qu'une action de cette envergure faite de façon automatique permettrait de limiter les risques d'erreur.

8. La livraison de charges

Ici, la livraison se résume à la correspondance entre la valeur finale transférée au système télémétrique et une valeur prédéfinie afin de vérifier que la fusée est arrivée à la destination escomptée.

9. Passer le MAX Q

Le MAX Q est une étape du lancement de la fusée où nous considérons qu'il faut réduire la vitesse de la fusée afin de faciliter son évolution dans l'atmosphère. Cette tâche est elle aussi automatisée et pour le moment effectuée à un moment spécifique fixé de manière statique.

II. Choix de conception et évolution d'architecture

Nous présentons dans cette partie nos choix de conception et les justifications qui nous ont mené progressivement à l'architecture finale.

1. Scope 1

La première version du projet nous a mené à l'architecture suivante :

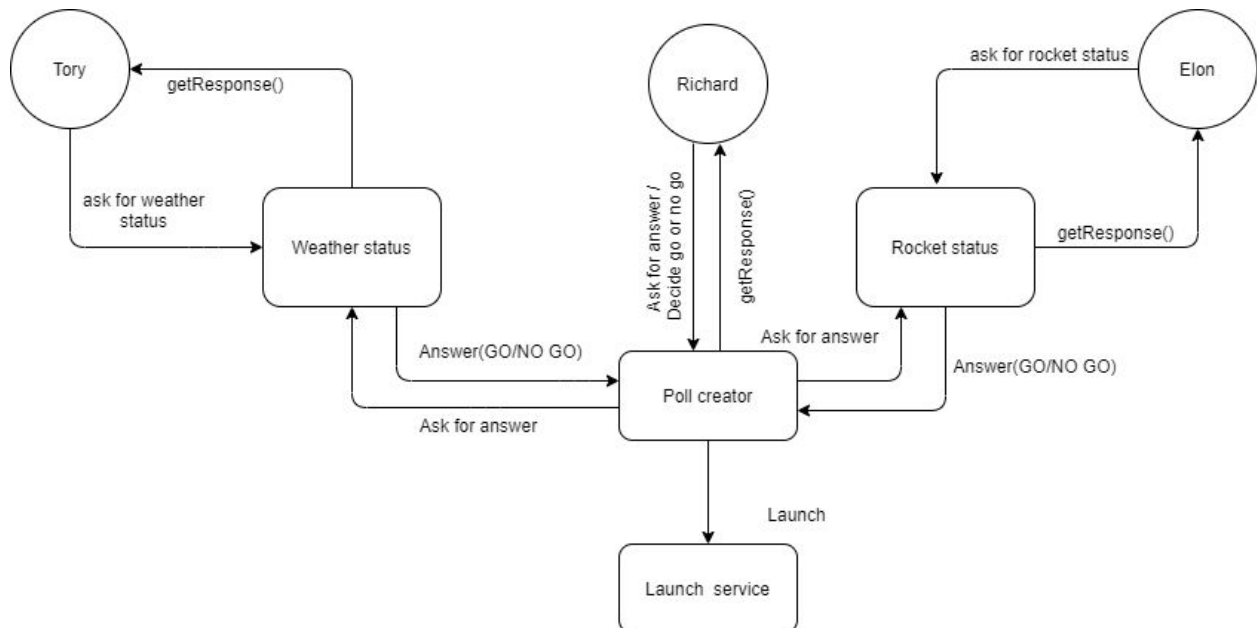



Figure 1 : Architecture du scope 1

(PS : Image également consultable sur

https://github.com/pns-si5-soa/box-20-21-team-f/blob/master/docs/archi_scope_1.png)

a. Description

Dans cette version, les acteurs du système sont **Elon**, **Tory** et **Richard**. Ils sont représentés par des cercles. Dans notre architecture, ces cercles représentent les points d'entrée de notre système. Pour ce MVP, ce sont des CLIs dans lesquelles chaque acteur peut taper des commandes pour effectuer les actions relatives à sa fonction dans le système. Ils interagissent ainsi avec différents micro-services dédiés à leurs fonctions respectives et représentés par des rectangles.



Pour ce qui est des services, nous avons à choisir quelle implémentation leur correspondrait le mieux en fonction des fonctionnalités qu'ils couvraient.

b. Weather status

Tory étant en charge du contrôle des conditions météorologiques, nous lui avons dédié un service “**Weather status**” implémenté en **REST** avec des données météorologiques basiques (nom du site, température, précipitation, humidité et vitesse du vent) et mockés. Nous l'avons implémenté ainsi car nous estimons que notre système n'est pas en charge de mesurer ou produire les données météorologiques mais uniquement de les consulter au besoin. Ce service constitue donc une API où nous consultons nos données météorologiques (**ask for weather status**) en fournissant en entrée les paramètres requis et nous obtenons en retour un résultat (**get response**).

c. Rocket status

À l'image de Tory, Elon se charge plutôt du contrôle de la fusée et de son lancement. Il doit être en mesure de dire si la fusée est opérationnelle quand on le lui demande. Il s'adresse donc au service “**Rocket status**” pour obtenir ces informations (nom de la fusée, état, essence, vitesse) toujours mockés. Elon peut donc consulter les données de toutes les fusées ou d'une fusée en particulier (**ask for rocket status**) et recevoir une réponse (**get response**).

d. Poll creator et Launch service

Pour finir, Richard est le chef d'orchestre du scénario et dispose du service “**Poll creator**” d'architecture **RPC** qui se charge une fois le vote enclenché par Richard (**Ask for answer**), d'interroger les services “**Rocket status**” et “**Weather status**” afin de s'assurer des conditions de lancement de la fusée (**Answer go/ no go**). Après cette étape, le service “**Poll creator**” décide directement de la réponse finale à envoyer au service du lancement de la fusée “**launch service**”. Ce système a l'avantage d'être automatique et de pas avoir à attendre que chaque utilisateur soit devant son interface pour répondre (même si c'est tout à fait possible car nous l'avons implémenté).

2. Scope 2

La deuxième version du sujet nous a mené à l'architecture suivante :

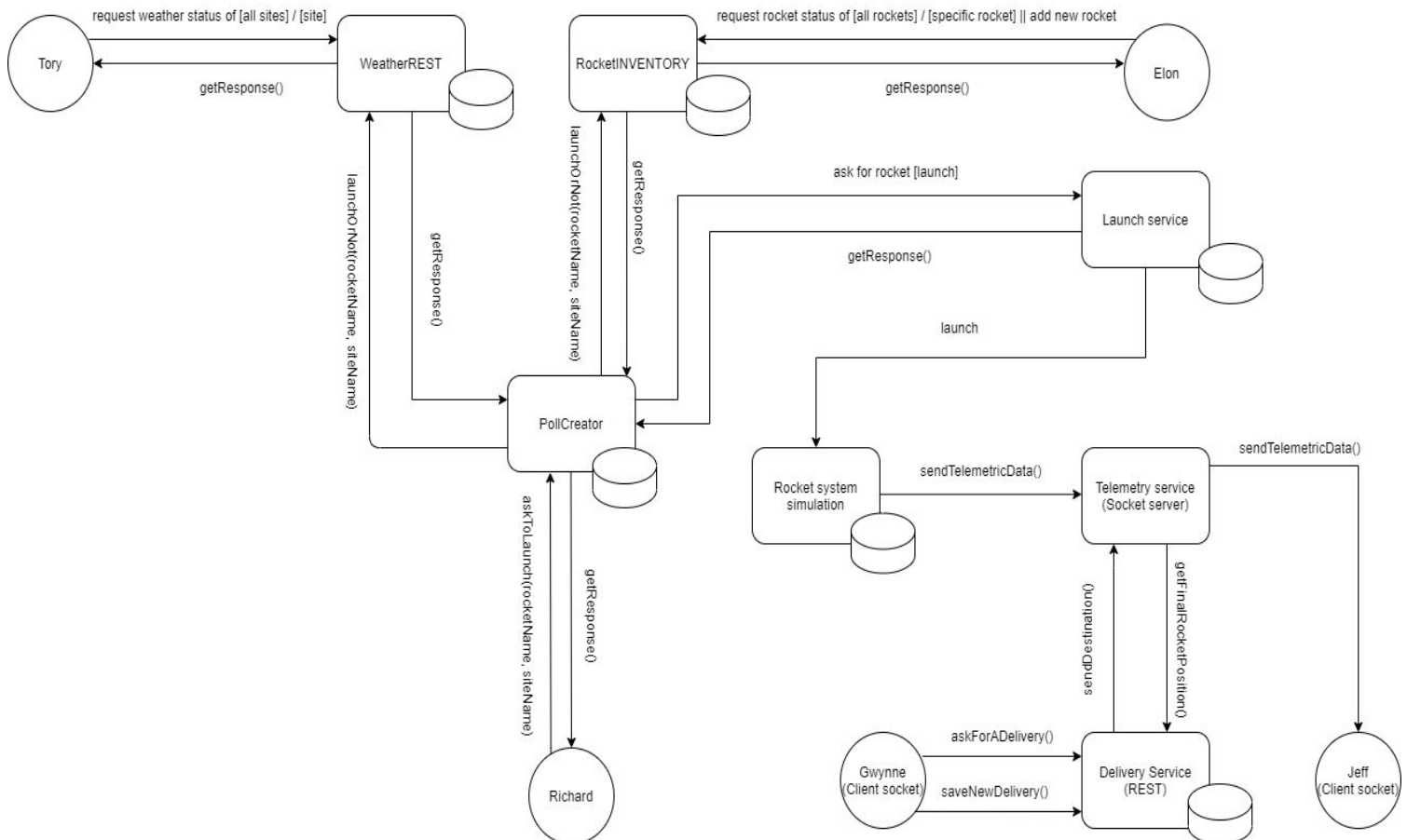



Figure 2 : Architecture du scope 2

(PS : Image également consultable sur

https://github.com/pns-si5-soa/box-20-21-team-f/blob/master/docs/archi_scope_2.png)

a. Description

Dans cette version, les nouveaux acteurs du système sont **Jeff** et **Gwynne** en interaction avec leurs services respectifs. Cette architecture se démarque tout de suite de la précédente avec l'icône de base de données rajoutée au niveau de chaque service. Le premier scope était relativement simple et ne nécessitait pas de base de données puisque les données étaient mockées. Mais le nombre de données à mocker et le traitement manuel de ces données compliquaient le développement des fonctionnalités.



Nous avons donc créé une base de données architecturée à l'image des services que nous possédons déjà. L'icône de base de données placée à côté de chaque service symbolise que ce service a accès à une table spécifique et dédiée à ses fonctionnalités. Nous utilisons de ce fait une base données **mongoDB sur le cloud**. Cela nous permet d'avoir une base de données à laquelle accèdent toutes les instances de notre application.


En outre, nous avons effectué des renommages de services parce que soit leurs fonctionnalités ne correspondaient pas tout à fait à leurs noms, soit parce que les noms étaient trop similaires et montraient un couplage de services. Le but étant de rendre chaque service le plus indépendant possible et parfois, d'identifier directement si c'est implémenté en REST, RPC ou SOCKET.

Renommages effectués :

- Weather status => **WeatherREST** : Mêmes fonctionnalités qu'avant, le besoin ici étant d'identifier l'implémentation REST et la démarquer des autres services réellement internes du système puisque nous ne produisons pas les données météorologiques.
- Rocket status => **RocketInventory** : Mêmes fonctionnalités qu'avant, le besoin ici étant d'éviter un conflit / couplage avec un autre service **Rocket system simulation** qui lui se charge d'envoyer les données télémétriques de la fusée en temps réel.

b. Launch service et Rocket system simulation

En partant du scope précédent, l'automatisation déjà existante nous permettait de dire juste si la fusée était lancée ou pas. Cette fois-ci, nous avons réellement besoin de faire apparaître d'une manière ou d'une autre la fusée dans notre système car nous devons récupérer ses données télémétriques une fois qu'elle était lancée. Le signal de lancement arrivant donc au **launch service**, ce dernier doit se charger de faire décoller la fusée. Pour ce faire nous avons créé un service **Rocket system simulation** qui **simule** une réelle fusée et donc reçoit le signal de lancement (**launch**). Une fois le signal reçu, la fusée a réellement décollé et qu'elle transmet immédiatement et en temps réel ses données télémétriques au service dédié : **Telemetry service (socket server)**. C'est



pendant cette simulation que la fusée est séparée en deux et c'est toujours ce service qui en est responsable. Il est important pour nous de préciser qu'à cette étape, tout se fait en socket.

c. Telemetry service (Socket server) et Delivery service (REST)

Une fois les données télémétriques reçues depuis la fusée simulée, elles sont immédiatement transmises à **Jeff** en temps réel toujours via transmission socket. Une fois que la dernière donnée télémétrique est reçue (c'est à dire que la fusée ne change plus de position), cette donnée est comparée avec la position donnée par le client (celui qui a organisé le lancement de la fusée) lors de l'enregistrement de la mission. C'est ici qu'intervient **Gwynne** car il interagit avec **Delivery Service** pour enregistrer une nouvelle mission (**saveNewDelivery**) à laquelle il affecte une fusée disponible. La mission contenant la destination finale, le service Telemetry demande les coordonnées de cette destination (**getFinalRocketPosition**) afin de la comparer à la dernière donnée télémétrique reçue de la fusée. Le résultat de la comparaison est ensuite sauvegardé dans le service Delivery où Gwynne pourra le consulter (**askForADelivery**).

3. Scope 3

La version finale du sujet nous a mené à l'architecture suivante :

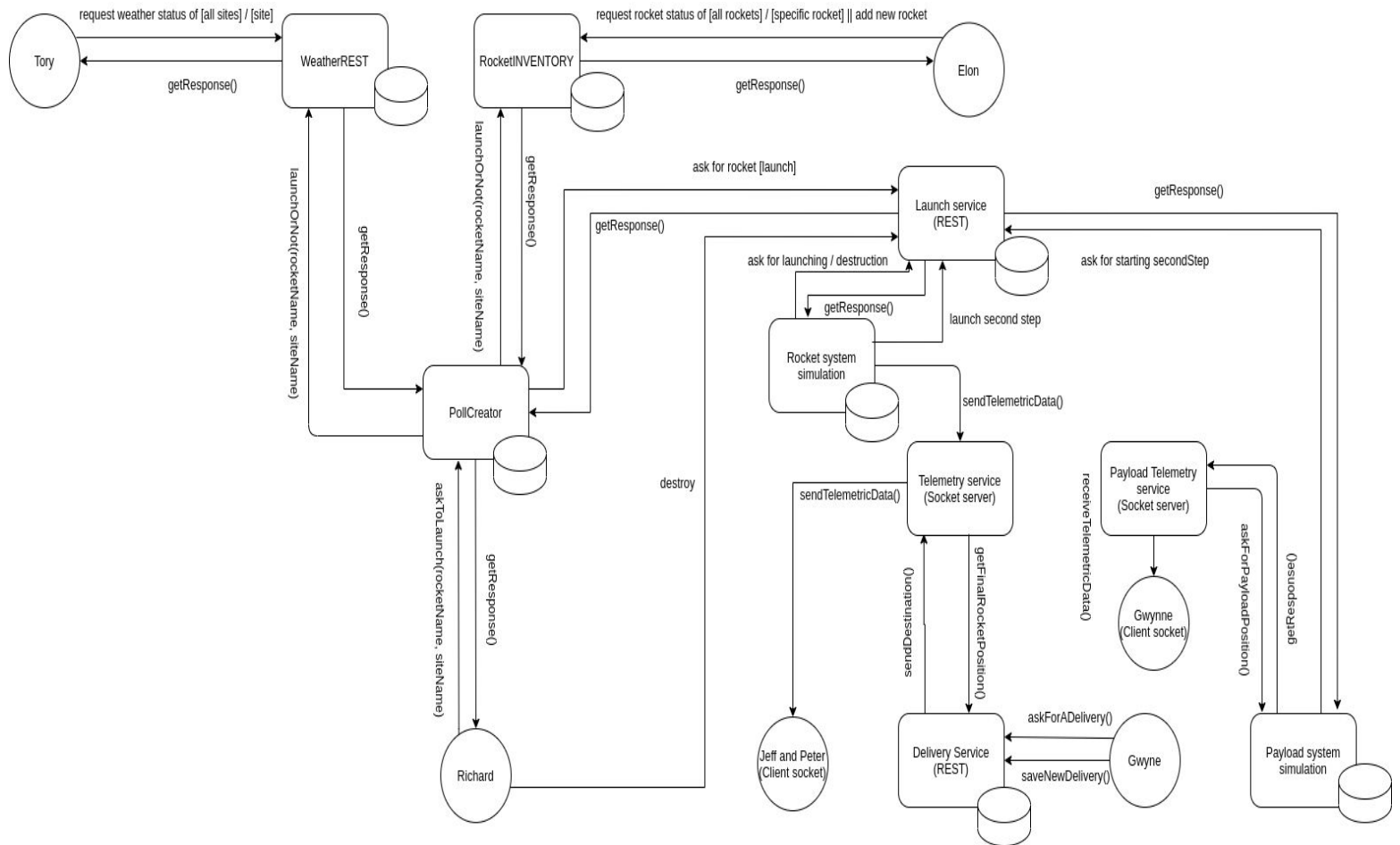


Figure 3 : Architecture du scope 3(Consultable sur le git)

(PS : Image également consultable sur https://github.com/pns-si5-soa/box-20-21-team-f/blob/master/docs/archi_scope_3.png)

a. Description

Dans cette version, le nouvel acteur du système est **Peter**. Mais nous ne créerons pas une nouvelle CLI pour ce dernier. On partira du principe que la première partie de la fusée est celle qui envoie des données à Jeff depuis le début de la simulation. De ce fait, Peter utilisera la même CLI que Jeff. En outre, les autres fonctionnalités à ajouter, seront rajoutées sur les CLI des utilisateurs correspondants. En outre, **Richard** effectue une nouvelle action sur **Launch service**. Il a la possibilité de détruire la fusée s'il se rend compte d'une anomalie quelconque.

b. Launch service, Rocket system simulation, Payload system simulation et Payload Telemetry Système


Dans ce set de scénarios, la séparation de la fusée et du chargement devient de plus en plus nécessaire à expliciter. En effet, il fallait désormais que le chargement envoie ses données télémétriques à Gwynne. De ce fait, il fallut créer un nouveau service qui soit chargé de la transmission de ces données télémétriques. De ce fait, nous avons conçu ce service de la même façon que le **“Rocket system Simulation”** car leur comportement était à peu près le même. Il sera donc aussi implémenté avec un système de socket avec **“Payload Telemetry Système”** qui transmet à son tour ces informations à Gwynne.

Mais une fois à ce niveau, nous avons eu un problème: la communication entre la première partie de la fusée et le chargement. Nous avons entrevu deux solutions:

- **c1** : la communication par socket en partant du principe que les deux services sont sur la même fusée
- **c2** : demander continuellement au **“Launch service”** si la fusée doit passer à la deuxième étape (séparation) et de même pour la destruction.

Pour notre architecture, nous avons choisi l'option **c2** car elle permet de garder une trace de la phase e lancement de la fusée à laquelle on est. Ce choix reste néanmoins discutable.

Quand le **“Rocket system Simulation”** décide que le moment le plus adapté à la séparation des composants de la fusée est arrivé, il met à jour l'état de la variable que le **“Payload system Simulation”** observe continuellement et ce dernier démarre l'étape de la séparation. Le **“Payload system Simulation”** commence alors à envoyer les données



téléométriques du chargement. Ces deux services sont dissociés car leurs comportements sont similaires mais indépendants.

Bilan

Pendant cette première partie de projet, nous avons par nous mêmes, constaté comment l'évolution de contraintes peuvent affecter l'évolution de l'architecture établie. Pour commencer, les choix du type de services. Sur les première et deuxième versions de projet, nous avons prévu que le **“Launcher service”** soit fait en architecture RPC. Mais le dernier set de US ainsi que le choix que nous avons fait, nous a conduit vers une modification de ce choix. Ensuite, la création de nouveaux services: la création du **“Payload system Simulation”** était due à la dissociation des comportements du chargement et de la première partie de la fusée. Enfin, certains services sont restés inchangés car leurs fonctionnalités n'ont pas changé dans le temps notamment **“Weather status”** et **“Rocket status”**. Certaines questions nous traversent néanmoins l'esprit:

- Comment éviter le fort couplage entre les services qui doivent interagir? (notamment pour connaître avec quels services interagir)
- Peut-on mieux gérer la gestion des services et des scénarios?