

Document d'architecture technique

Table of Contents

Gestionnaire de versions	3
Introduction	4
Q1: En partant des pré-requis, que proposez-vous comme séparation en quartiers fonctionnels de votre application et en termes de micro-services?	5
Vocabulaire métier et user stories	5
Découpage en quartier fonctionnels	11
Justification de l'architecture	12
Q2: Comment gérer le système d'authentification avec un OIDC?	13
Mise en place de Keycloak	13
Compte vanilla	15
LDAP	16
Compte Google	17
Q3: Communication Inter micro-services	19
Q4: Comment tracer les logs et les requêtes?	22
Q5: Comment ajouter un moteur de recherche?	24
Moteur de recherche	24
Intégration dans le frontend	26
Q6: Comment mettre en place l'architecture de runner	29
Définition du runner	29
Architecture	29
Enregistrement des runners	30
Q7: Comment gérer les données?	31
Organisation des bases de données	31
SGBD utilisés	32
Schemas de données	36
CQRS et Event Sourcing	37
Q8: Comment ajouter une application mobile dans le système	39
Fonctionnalités et mockup visuels	39
Flows de fonctionnement	43
API	44
Authentification	44
Q9: Comment gérer les problématiques de sécurité	46
Sécurisation du réseau	46
Gestion des secrets	48
Protection contre les attaques DDoS	50
Q10: Comment intégrer les applications UI?	52

Bibliographie/Webographie/Liens	54
---------------------------------------	----

Gestionnaire de versions

Version	Date	Auteur	Commentaire
v1.0	22/01/2023	Yann POMIE	Rendu initial
v1.1	07/05/2023	Yann POMIE	Ajout Post-Polystore: changements architecturaux dans la question 1, usage de RabbitMQ dans la question 3 (ce qui était auparavant interdit pour la V1), évocation de lambdo dans la question 6 et rajout d'une section sur l'Event Sourcing et CQRS dans la question 7

Introduction

Polycode est un projet scolaire créé en 2022 par la promotion DevOps 2021-2022 de Polytech Montpellier, il s'agit d'une plateforme en ligne permettant à ses utilisateurs de consulter des lessons de programmation, d'écrire du code qui sera exécuté et d'être évalué sur ses compétences lors de campagnes de tests.

La première itération de cette application suivait une conception monolithique, c'est à dire que toutes les fonctionnalités du *back end* de l'application résidaient toutes en un seul binaire, le problème de ce genre de conception est que toutes les fonctionnalités ne sont pas toutes sollicitées de la même manière ce qui entraîne donc du gâchis de ressources. Il nous a donc été demandé de revoir notre conception et de la découper en micro-services pour répondre à ce problème, mais aussi de rajouter d'autres fonctionnalités à l'application tout en prenant en compte notre nouvelle architecture. Nous allons donc présenter dans ce document, notre réponse à une dizaine de questions soulevant ces problématiques.

Dans un premier temps nous allons proposer une séparation de Polycode en quartiers fonctionnels, de nouveaux mécanismes d'authentification ainsi que les processus de communication inter-services utilisés. Viendront ensuite l'ajout de deux fonctionnalités supplémentaires: le traçage des logs de l'application et un moteur de recherche. Nous terminerons par divers aspects de l'organisation de notre application avec notamment l'architecture de nos runner de code, l'organisation de nos bases de données, l'ajout d'une application mobile, la sécurisation de notre application et enfin nous finirons sur l'intégration de nos applications UI.

Q1: En partant des pré-requis, que proposez-vous comme séparation en quartiers fonctionnels de votre application et en termes de micro-services?

Vocabulaire métier et user stories

Suite à de nombreuses incompréhensions entre les différents acteurs du projet, nous avons redéfini une grande partie du vocabulaire métier, nous en avons aussi profité pour redéfinir nos user stories.

Voici les définitions du vocabulaire métier que vous serez susceptible de trouver dans nos user stories et dans le reste du document.

- **Invité** : Client n'ayant pas de compte sur la plateforme
- **Utilisateur** : Client ayant un compte sur la plateforme
- **Administrateur** : Utilisateur privilégié ayant des droits CRUD sur toutes les ressources de la plateforme
- **Équipe** : Regroupement d'utilisateurs composé d'un capitaine et de membres
- **Évaluation** : Une évaluation destinée à des utilisateurs
- **Test** : Il s'agit d'un groupe ordonné de contenus, noté, conçu pour évaluer les utilisateurs. Il est créé par un créateur d'évaluation.
- **Candidat** : Utilisateur participant à une évaluation ou à un test
- **Campagne de tests** : exposition d'un groupe utilisateurs à un test
- **Composant** : Élément affiché à l'utilisateur, il peut s'agir de markdown, d'un éditeur de code etc...
- **Contenu** : Ensemble cohérent de composants
- **Module** : Un groupement de contenus, peut également contenir d'autres modules appelés sous-modules
- **PolyPoints** : Points gagnés par les utilisateurs lorsqu'ils réussissent des défis de composant
- **Validateur** : Élément caché de l'utilisateur, associé à un composant, il permet de vérifier la validité des réponses de l'utilisateur et récompense les bonnes réponses avec des PolyPoints.

En tant que...	Je veux...	Afin de...
Invité	Créer un compte	D'utiliser l'application
Invité	Créer un compte avec un compte Google/Polytech	D'utiliser l'application sans avoir à créer un mot de passe et sans avoir à saisir mes informations personnelles

En tant que...	Je veux...	Afin de...
Utilisateur	Mettre à jour mes adresses mails	Pour récupérer mon compte en cas de perte de mot de passe ou en cas de problèmes d'email
Utilisateur	Recevoir un mail lors de bienvenue après inscription	Être averti que mon inscription à bien été faite
Utilisateur	Changer mon nom, mon mot de passe, mon language de programmation et ma bio	Mes informations personnelles soient à jourwzJB7BBo4W9uUjm+rBMS3H4ZoDTE2dM2A+JbfE515doK
Utilisateur	Delete my account	Ne plus avoir de trace sur l'application
Utilisateur	Me connecter grâce une adresse mail et un mot de passe/grâce à un compte Google ou Polytech	Je puisse accéder à toutes les fonctionnalités de l'application
Utilisateur	Me déconnecter de mon compte	Éviter les accès non autorisés depuis le navigateur utilisé.
Utilisateur	Récupérer mon mot de passe via un mail	Je puisse récupérer mon compte en cas de perte de mon mot de passe
Utilisateur	Créer une nouvelle équipe	Réunir un groupe d'utilisateurs et participer au leaderboard des équipes (somme des points de chaque utilisateur).
Capitaine	Invite d'autres utilisateurs à être membre de mon équipe	Mon équipe s'agrandisse
Capitaine	Exclure un membre de mon équipe	Enlever un membre problématique / non actif / qui n'est plus en adéquation avec le groupe.
Capitaine	Donner le rôle de capitaine à un autre membre de mon équipe	Je suis disposé de ces fonctions
Capitaine	Supprimer mon équipe	Ne plus avoir de traces de cette équipe, pour quelque raison.
Capitaine	Changer le nom et la description de mon équipe	Les informations de l'équipe reste à jour
Utilisateur	Accepter ou décliner une invitation à une équipe	Ajoute les PolyPoints (précédents) et gagnés de l'utilisateur à l'équipe
Utilisateur	Pouvoir quitter une équipe	Je ne suis plus associé à un groupe d'utilisateur
Utilisateur	Voir les points de mon équipe	Constatier l'avancement des membres de l'équipe
Utilisateur	Voir le classement des équipes	Je vois le placement de mon équipe vis-à-vis des autres

En tant que...	Je veux...	Afin de...
Utilisateur	Voir le classement interne des membres de l'équipe	Voir qui a participé le plus dans l'équipe, concurrence interne
Utilisateur	Voir la liste des exercices disponibles	Je puisse choisir un exercice à faire
Utilisateur	Voir la liste des modules disponibles	Je puisse choisir un module à faire
Utilisateur	Voir les sous-modules et les exercices d'un module	Trouver les étapes à faire pour compléter le module
Utilisateur	Voir la liste des évaluations disponibles	Je puisse choisir une évaluation à passer
Utilisateur	Voir les derniers exercices / modules mis en ligne	Voir le nouveau contenu
Utilisateur	Voir les informations d'un exercice	M'informer sur le sujet d'un exercice
Utilisateur	Voir les informations d'un module	M'informer sur le sujet du module, l'objectif
Utilisateur	Voir les informations d'une évaluation	M'informer sur le sujet de l'évaluation, l'objectif
Utilisateur	Voir l'énoncé d'un exercice	D'apprendre une nouvelle notion, connaître le problème à résoudre, question à répondre pour valider la notion
Utilisateur	Proposer une solution à l'exercice	Gagner des PolyPoints et avancer dans le module associé
Utilisateur	Dans le cas d'un code à écrire, exécuter un validateur intermédiaire	Vérifier si mon code est correct pour le validateur en question
Utilisateur	Revoir la dernière solution qui a passée le plus de validateurs	Reprendre le code depuis un appareil différent, à un autre moment, pour l'améliorer
Utilisateur	Écrire (et modifier) sa solution de code dans un éditeur intégré à la page de l'exercice	Proposer une solution à l'exercice
Utilisateur	Ajouter des fichiers dans l'éditeur intégré à la page d'exercice	Organiser la solution en plusieurs fichiers
Utilisateur	Supprimer des fichiers dans l'éditeur	Organiser la solution en plusieurs fichiers
Utilisateur	Afficher les données de validateur (entrée et sortie) en échange de avec des PolyPoints	Comprendre mieux comment résoudre l'exercice
Utilisateur	Suivre ma progression dans chacun des modules	Voir ce qui est complété / à faire

En tant que...	Je veux...	Afin de...
Utilisateur	Voir le classement global des utilisateurs (par polypoints)	Nous motiver à atteindre le sommet (principe de concurrence)
Utilisateur	Passer une évaluation	Obtenir une certification
Utilisateur	Lire le contenu d'un cours	Monter en compétence sur un sujet
Créateur de contenu	Créer un exercice	Proposer l'apprentissage d'une nouvelle notion, faire vérifier la connaissance de cette notion par une question/ un code à écrire
Créateur de contenu	Créer un module	Organiser les exercices par notion majeure / thématique
Créateur d'évaluation	Créer une évaluation	Vérifier les compétence d'un utilisateur sur un contenu
Créateur de contenu	Ajouter ses exercices à un module qu'il a créé	Remplir le contenu d'un module en ensemble d'élément cohérent
Créateur de contenu	Ajouter des modules dans un module, et ce avec des modules qu'il a créé (sous-module)	Remplir le contenu d'un module en ensemble d'élément cohérent
Créateur de contenu	Modifier le nom, la description, le nombre de PolyPoints de récompense, les tags, le contenu (exercices et sous-module) de ses modules	Garder à jour un module
Créateur de contenu	Modifier le titre, la description, le contenu, récompense en polypoints, les validateurs, les tags d'un exercice	Garder à jour un exercice
Créateur de contenu	Modifier le titre, la description, le contenu d'une évaluation	Garder à jour une évaluation
Créateur de contenu	Supprimer un exercice qu'il a créé	Réparer une erreur / ne plus vouloir la présence de ce contenu
Créateur de contenu	Supprimer un module qu'il a créé	Réparer une erreur / ne plus vouloir la présence de ce contenu
Créateur de contenu	Supprimer une évaluation qu'il a créé	Réparer une erreur / ne plus vouloir la présence de ce contenu
Créateur de contenu	Voir le résultat des utilisateurs sur une évaluation qu'il a créé	Pour que le recruteur / professeur voie le résultat des élèves pour attribuer une note / recruter
Administrateur	Promouvoir un utilisateur en rédacteur	Qu'un utilisateur ai les droits d'un "redacteur"
Administrateur	Promouvoir un utilisateur en Administrateur	Qu'un utilisateur ai les droits d'un "Administrateur"

En tant que...	Je veux...	Afin de...
Administrateur	Créer un utilisateur	Utiliser l'application avec un autre compte
Administrateur	Récupérer les données d'un utilisateur	Voir les informations confidentielles d'un compte utilisateur
Administrateur	Mettre à jour les données d'un utilisateur	Mettre à jour les informations personnelles afin qu'elles soient cohérentes
Administrateur	Supprimer un utilisateur	Ne plus donner accès à la plateforme pour un compte utilisateur
Administrateur	Créer un exercice	Proposer l'apprentissage d'une nouvelle notion, faire vérifier la connaissance de cette notion par une question/ un code à écrire
Administrateur	Modifier le titre, la description, le contenu, récompense en polypoints, les validateurs, les tags d'un exercice	Garder à jour un exercice
Administrateur	Supprimer un exercice	Réparer une erreur / ne plus vouloir la présence de ce contenu
Administrateur	Créer un module	Créer un module afin de regrouper des contenus
Administrateur	Récupérer les données d'un module	Voir les informations et les contenus associés à ce module
Administrateur	Mettre à jour les données d'un module	Garde le module à jour
Administrateur	Supprimer un module	Effacer les traces du module sur la plateforme
Administrateur	Créer une évaluation	Vérifier les compétence d'un utilisateur sur un contenu
Administrateur	Récupérer les données d'une évaluation	Voir les différentes données en lien avec une évaluation
Administrateur	Mettre à jour les données d'une évaluation	Ajouter des utilisateurs ou modifier des données relatives à une évaluation
Administrateur	Supprimer une évaluation	Enlever une évaluation de la plateforme
Administrateur	Créer une team	Rassembler des utilisateurs dans une équipe
Administrateur	Ajouter un membre dans mon équipe	Proposer à un utilisateur de rejoindre mon équipe
Administrateur	Supprimer un membre d'une team	Enlever un utilisateur de mon équipe pour une quelconque raison

En tant que...	Je veux...	Afin de...
Administrateur	Supprimer une team	Supprimer une team qui ne valide pas les conditions d'utilisation
Administrateur	Modifier la description d'une équipe	Avoir une description à jour de l'équipe
Créateur d'évaluation	Créer une campagne de test	Evaluer le niveau des utilisateurs
Créateur d'évaluation	Ajouter des utilisateurs à ma campagne via une interface web	Faire participer les candidats
Créateur d'évaluation	Supprimer des utilisateurs à ma campagne via une interface web	Enlever un candidat des participants
Créateur d'évaluation	Ajouter des utilisateurs à ma campagne via des appels API	Faire participer les candidats
Créateur d'évaluation	Supprimer des utilisateurs à ma campagne via des appels API	Enlever un candidat des participants
Créateur d'évaluation	Ajouter des utilisateurs à ma campagne via l'importation de fichiers csv	Faire participer les candidats
Créateur d'évaluation	Voir les résultats et statistiques sur la campagne que j'ai créé	Me rendre compte du niveau des candidats testés
Créateur d'évaluation	Ajouter des tags à mes candidats	Grouper les candidats
Créateur d'évaluation	Définir une date limite pour ma campagne	Clôturer ma campagne à une date fixe
Candidat	Revenir sur un test et reprendre là où j'en était	Finir mon test si jamais je quitte l'application
Créateur d'évaluation	Définir un temps limite pour chaque question de ma campagne	Les candidats répondent dans un temps limité
Créateur d'évaluation	Définir un nb de points pour chaque question	Avoir un score par candidat et voir leur différence de score à la fin de la campagne
Candidat	Recevoir un mail me permettant de participer à une campagne de tests	Avoir un lien pour participer à une campagne
Candidat	Accepter de participer à une campagne	Tester ses compétences à travers une campagne
Candidat	Refuser de participer à une campagne	Avoir la possibilité de refuser une campagne et que le créateur en soit informé
Créateur d'évaluation	Éditer ma campagne, les tests liés	Modifier une campagne précédemment créée

En tant que...	Je veux...	Afin de...
Créateur d'évaluation	Définir une date de début de ma campagne	Définir une date pour les candidats, ainsi qu'un temps imparti pour finaliser la campagne
Créateur d'évaluation	Envoyer des liens de ma campagne manuellement à mes candidats	S'assurer que les candidats reçoivent bien le lien pour participer à une campagne
Candidat	Recevoir un mail de confirmation contenant des stats quand j'ai soumis mon test	Notifier l'utilisateur que sa participation et ses réponses ont bien été enregistrées pour une campagne
Créateur d'évaluation	Voir le nombre de points totaux par candidats	Comparer les points des candidats ayant participé à la campagne
Créateur d'évaluation	Visualiser un graphique/un excel par tags de content et par candidats	Voir graphiquement les différents résultats
Créateur d'évaluation	Exporter les résultats synthétisés dans un pdf	Sauvegarder les résultats des candidats et avoir une vue synthétique
Créateur d'évaluation	Exporter les résultats détaillés dans un pdf	Sauvegarder les résultats des candidats et y avoir accès sans passer par l'application
Créateur d'évaluation	Avoir une vue comparative des candidats sous la forme d'un tableau excel	Comparer les scores des candidats à travers un tableau
Créateur d'évaluation	Trier la liste des candidats par tags, résultats	Comparer les résultats des candidats en fonction de données précises
Créateur d'évaluation	Télécharger les scores des candidats	Afin de garder les stats en local

Découpage en quartier fonctionnels

En considérant ces user stories on peut en déduire ces quartiers fonctionnels :

1. Authentification/Authorisation : permet à l'utilisateur de s'inscrire et de s'identifier sur la plateforme. Vérifie les droits de l'utilisateur sur une ressource.
2. Gestion des utilisateurs : permet la gestion des utilisateurs.
3. Edition de modules : donne la possibilité d'éditer et de visualiser des modules ainsi que leurs contenus et composants.
4. Envoi de mail : envoie des mail aux utilisateurs.
5. Gestion de campagne : donne la possibilité de la gestion des campagnes de tests.
6. Gestion des runners : permet de lancer des runners afin d'exécuter du code.

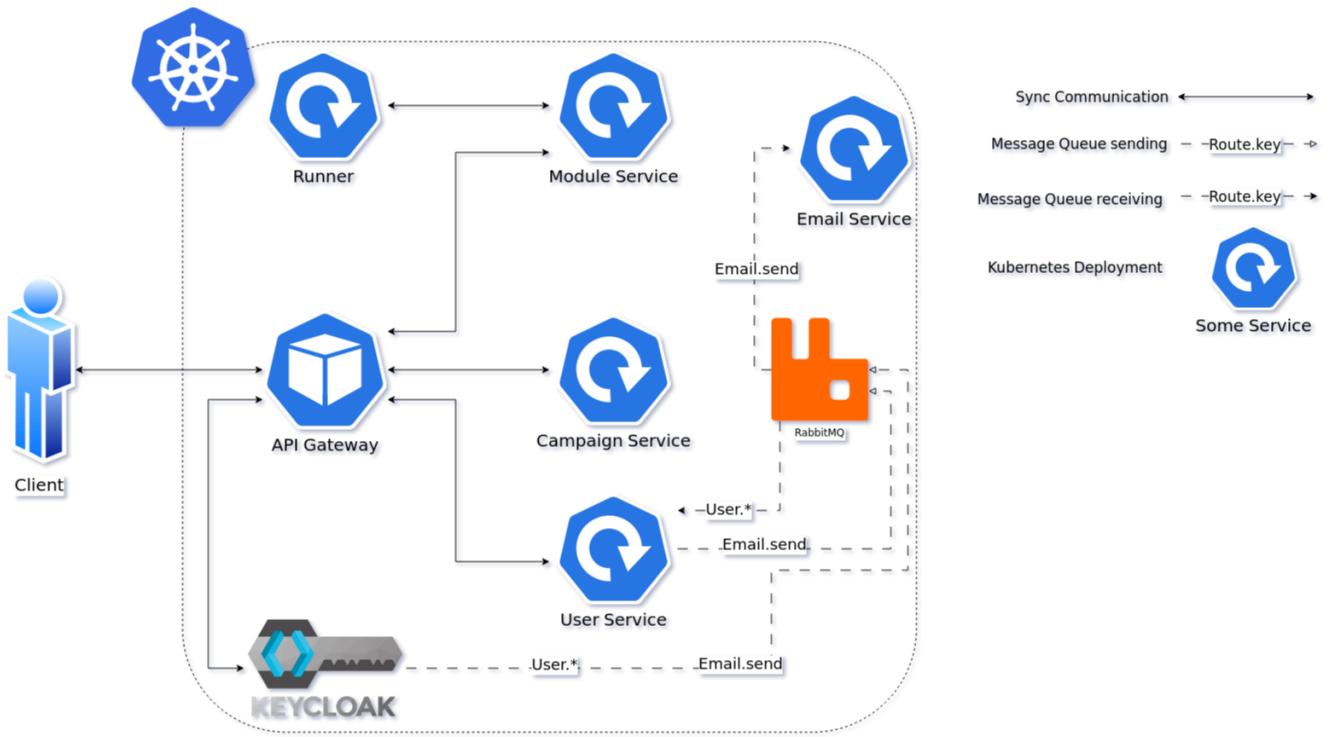


Figure 1. Architecture en microservices proposée

Justification de l'architecture

Le but premier de cette organisation est de réduire au maximum les dépendances entre chaque service notamment au niveau des canneaux de communications, en effet on peut remarquer que peu de services communiquent entre eux. Le fait de limiter le nombre de canaux de communications permet de réduire les risques de défaillance générale et de faciliter la maintenance.

L'API Gateway est le point d'entrée de l'application, elle se chargera de rediriger les requêtes vers les services concernés. Elle se chargera aussi de vérifier les droits de l'utilisateur sur une ressource en interrogeant Keycloak.

Vu que nous nous plaçons dans une architecture de microservices il faut partir du principe que nos services auront des réplicas qui vont cracher et d'autres qui vont démarrer. Pour assurer un bon routage de nos requêtes, nous allons utiliser un service registry afin de garder en mémoire nos services encore vivants ainsi que leur adresse IP (cf. [Question 9 - Enregistrement des runners](#)).

Il a aussi été envisagé de faire un service *Équipe* et *Contenu* séparés mais cela aurait impliqué de faire des appels API supplémentaires et donc d'encore augmenter le nombre de canaux de communications et donc d'augmenter le temps de latence. De plus ça n'aurait aucun sens car les notions d'utilisateur et d'équipe sont interdépendantes et qu'un module n'a au final d'intérêt que les contenus qu'il contient.

Q2: Comment gérer le système d'authentification avec un OIDC?

Mise en place de Keycloak

Afin de gérer l'authentification dans notre application nous allons mettre en place des instances Keycloak. Keycloak est un *identity provider* OpenID Connect et SAML 2.0. Il permet de gérer les utilisateurs, les droits d'accès, les sessions, les tokens d'accès et d'autres. Dans notre cas nous allons nous contenter de l'authentification de nos utilisateurs et de leur connection.

Pour ce faire nous allons utiliser les pages de connection et d'inscription et de connexion fournies par Keycloak, quand l'utilisateur voudra faire l'une de ces deux actions il sera donc redirigé vers une instance Keycloak, cette dernière s'occupant de stocker ses identifiants à la place du service d'utilisateur et donc sans passer par ce dernier.

Au delà du fait que Keycloak nous est imposé dans le cadre de la prochaine itération de Polycode, le choix cette solution est justifiée par sa polyvalence: Keycloak fonctionne avec de nombreux SGBD, il peut être manipulé via de nombreux langages grâce à une API HTTP REST, il supporte de base une dizaine d'*identity providers* allant de Paypal à Twitter et il permet de fédérer les utilisateurs de différentes manières. Si on ajoute à cela les nombreuses autres fonctionnalités de Keycloak comme la confirmation de mail, des droits d'accès, des event listeners, etc. on peut dire que c'est une solution qui est adaptée à notre cas d'utilisation.

Keycloak sera déployé sur notre cluster Kubernetes avec une règle ingress permettant de rediriger vers les pages de connection et d'inscription de Keycloak. L'intérêt de cette manœuvre est de permettre à Keycloak de déclencher des évènements ce qui permettra à notre application de réagir à ces derniers, comme par exemple l'enregistrement via nos services, des informations de l'utilisateur tel que ses rôles, son nombre de Polypoints, etc. Keycloak met à disposition une librairie Java permettant d'écrire ses propres *event listeners* afin de définir une réaction spécifique à des évènements, au vue de l'omniprésence de communication par API REST dans notre application (cf. [Question 3: Communication Inter micro-services](#)).



Figure 2. Illustration de la communication entre Keycloak et Polycode

Dans un premier temps un système de webhook a été envisagé. Les Webhooks sont des serveurs auquel des clients vont s'abonner afin d'être notifiés d'un évènement, à l'inverse du http polling l'initiative vient du serveur et non du client. Il était envisagé de faire un event listener qui consisterait en un serveur webhook qui notifie ses abonnés quand un utilisateur se connecte ou s'inscrit, le problème de cette manière de faire est que les services ayant besoin de ses notifications

doivent s'inscrire auprès du serveur. Serveur qui doit garder en mémoire l'adresse de ses abonnés, hors dans un système de microservices on part du principe que nos services vont continuellement s'arrêter et démarrer, cela peut donc entraîner une fuite de mémoire au niveau du webhook, sa liste d'adresse ne cessant d'augmenter.

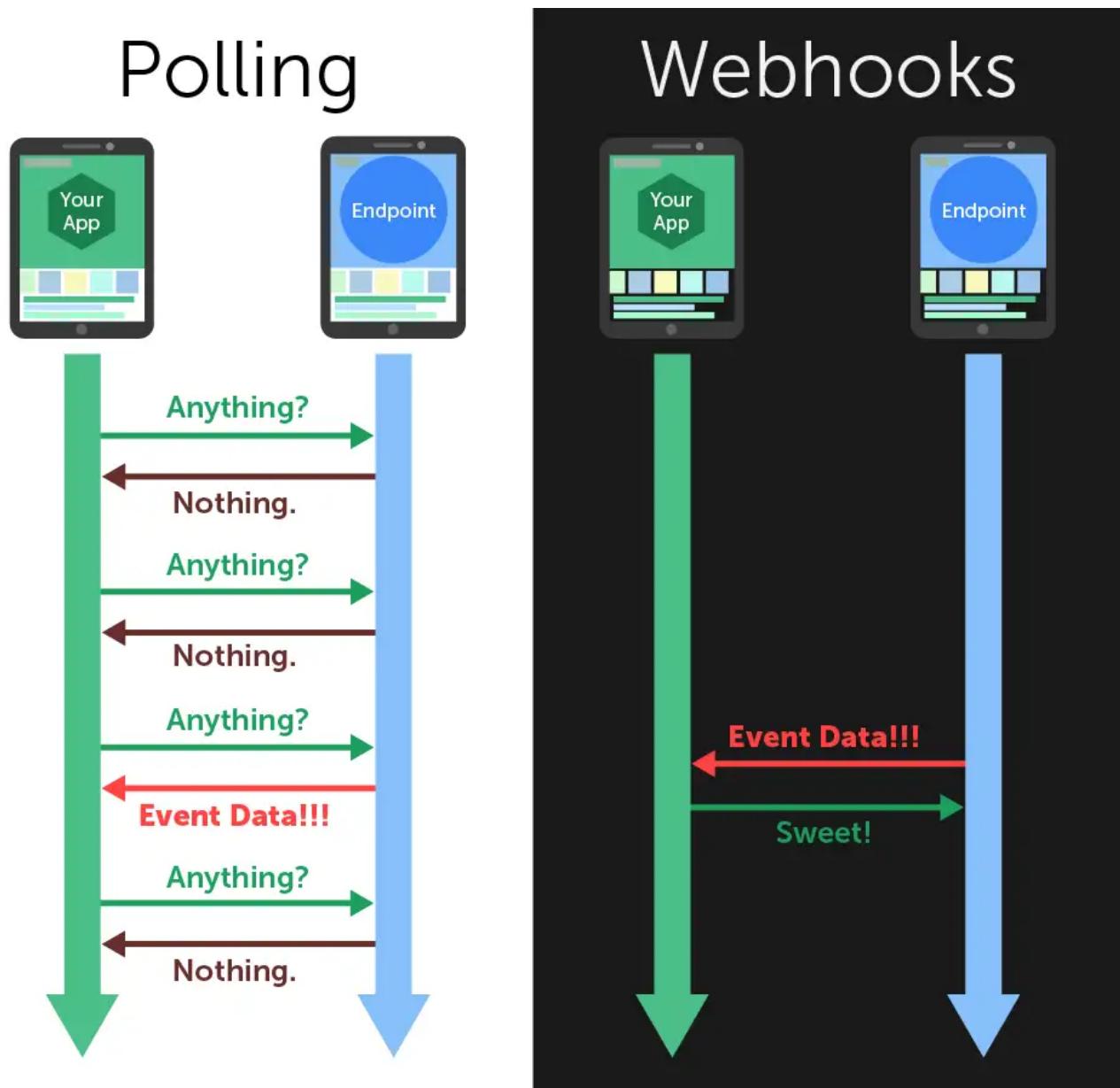


Figure 3. Illustration de webhook

Pour palier à ce problème notre event listener va simplement envoyer des messages AMQP à un serveur RabbitMQ avec des clés de routage de la forme `<domaine>.<événement>` (par exemple `Email.send`, `User.register` etc.) que les services pourront recevoir et interpréter, ce protocole de communication est adapté car Keycloak va émettre des évènements dont il n'attend rien en retour, ce qui permet d'éviter l'usage de HTTP et ses désavantages (cf. [Question 3: Communication Inter micro-services](#)).

Vous pouvez remarquer que suite à des problèmes techniques et à un manque de temps, le *proof of concept [poc2]* est incomplet et pas déployé. Veuillez nous excuser pour ce manque.

Compte vanilla

La gestion de compte vanilla (compte stocké dans la base de données de Polycode) se fait via une page de connexion et d'inscription fournie par une instance Keycloak. Instance qui se chargera de la vérification des identifiants, de l'absence de doublons, de l'envoi de mail de confirmation, etc.

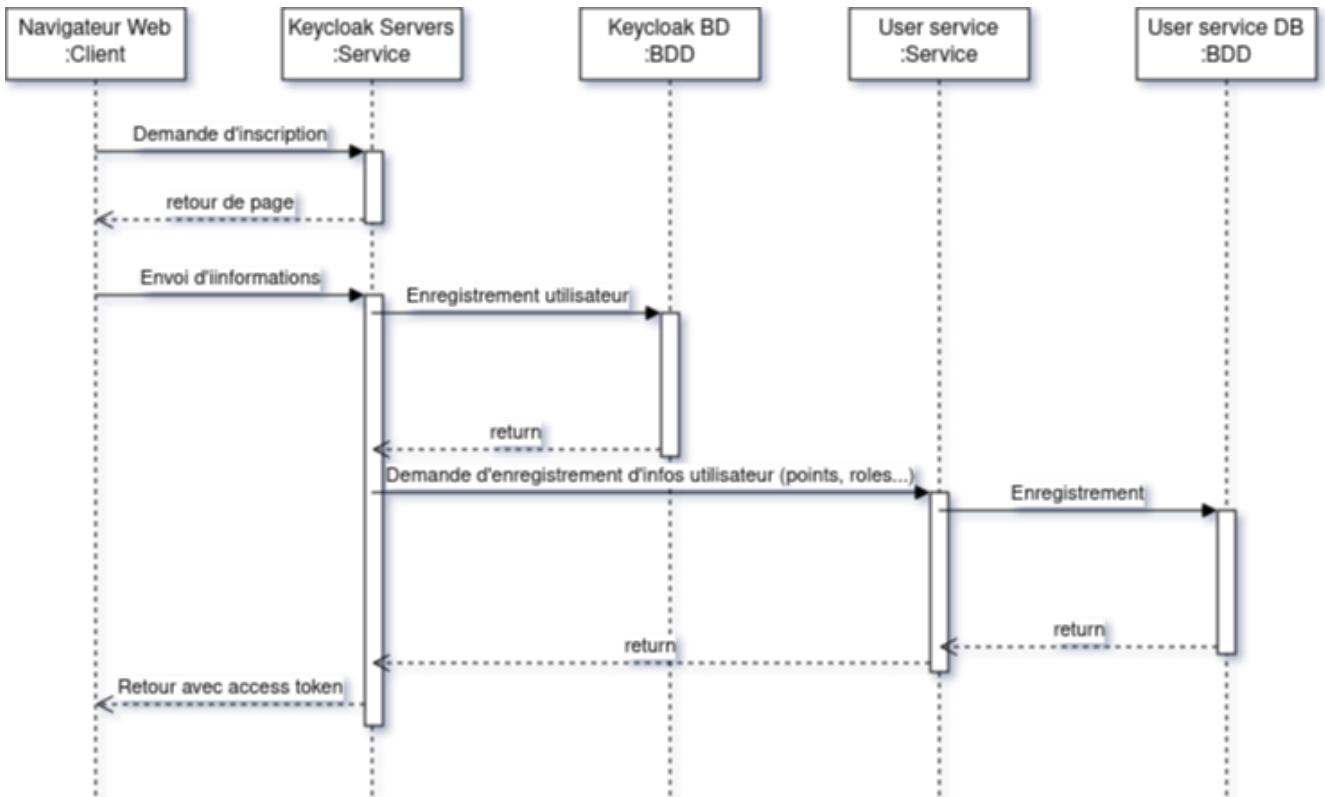


Figure 4. Inscription avec un compte vanilla

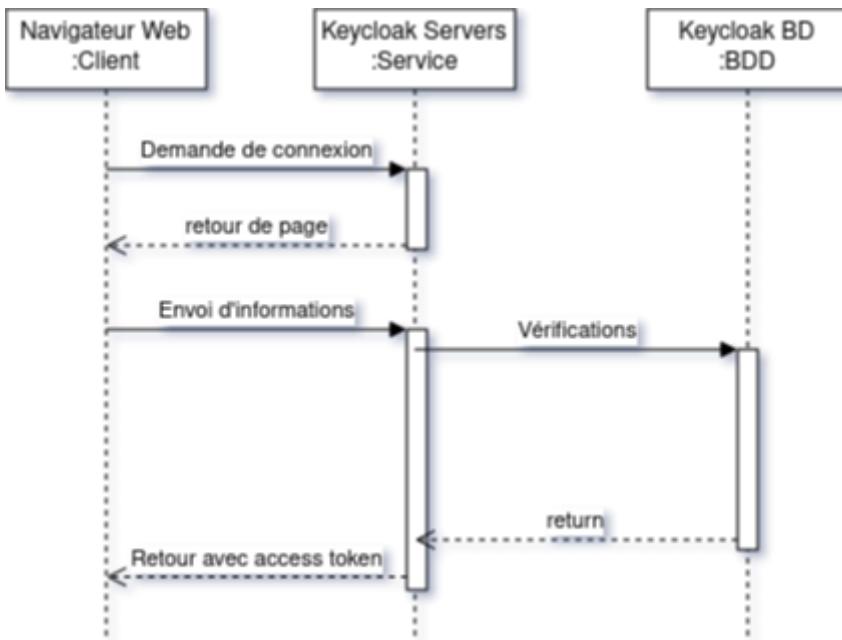


Figure 5. Connexion avec un compte vanilla

LDAP

Keycloak propose une option d'importation des utilisateurs, cette option permet si elle est activée de sauvegarder à intervalles régulier les utilisateurs du système LDAP dans Keycloak. Nous utiliserons cette option, car Keycloak assumera la lecture et écriture de ses données de son côté, ce qui permet de solliciter le serveur LDAP uniquement pour la vérification de mot de passe. Envoyer le moins de requêtes possibles au serveur LDAP de Polytech est important car il ne faudrait pas surcharger ce serveur utilisé par environ 1500 étudiants.

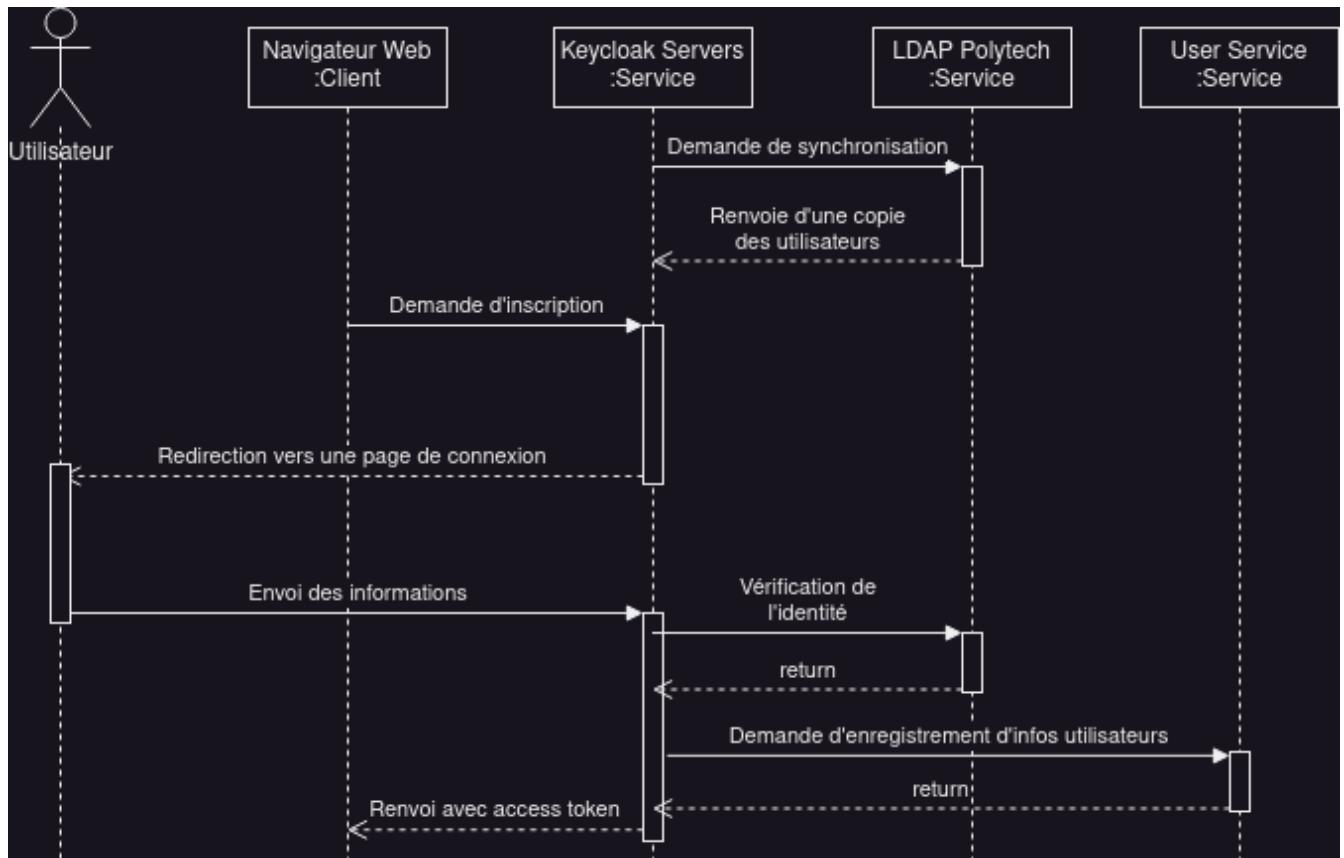


Figure 6. Inscription avec un compte LDAP

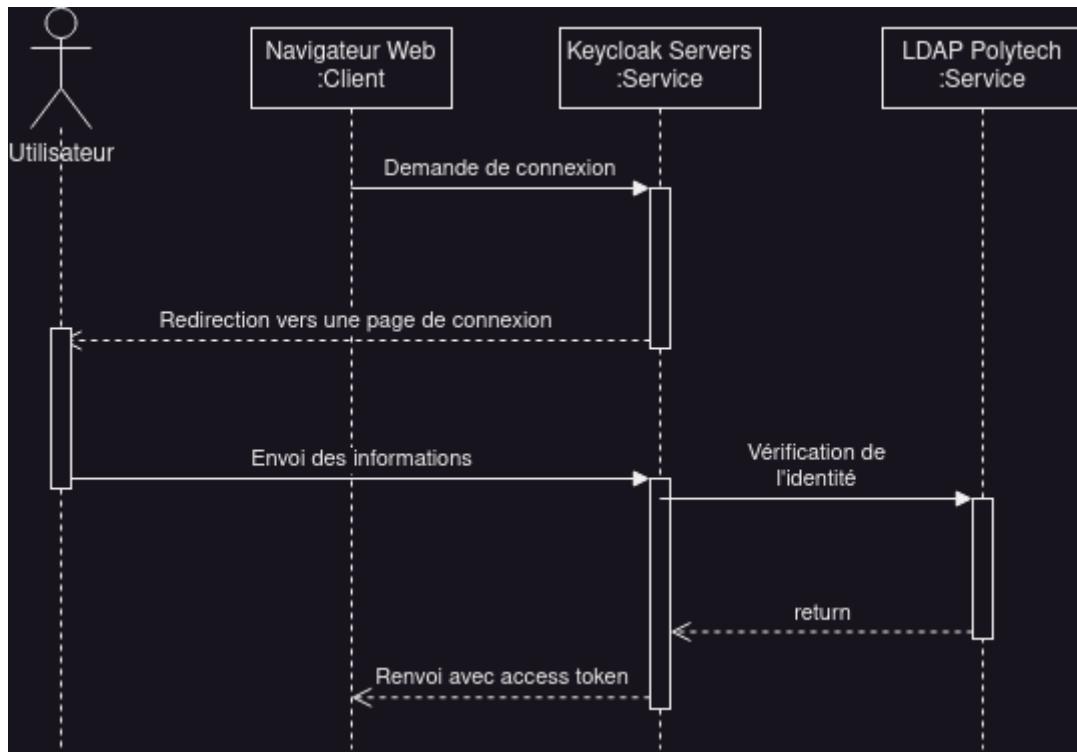


Figure 7. Connexion avec un compte LDAP

Compte Google

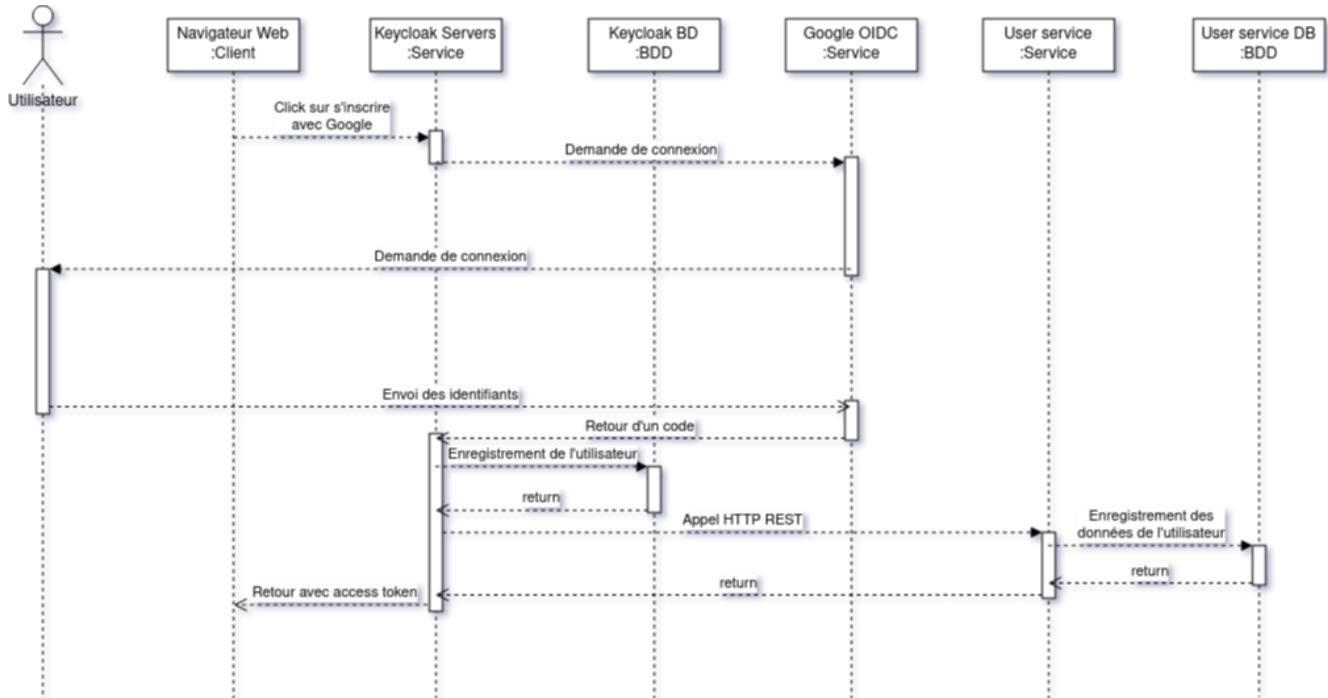


Figure 8. Incription avec un compte Google

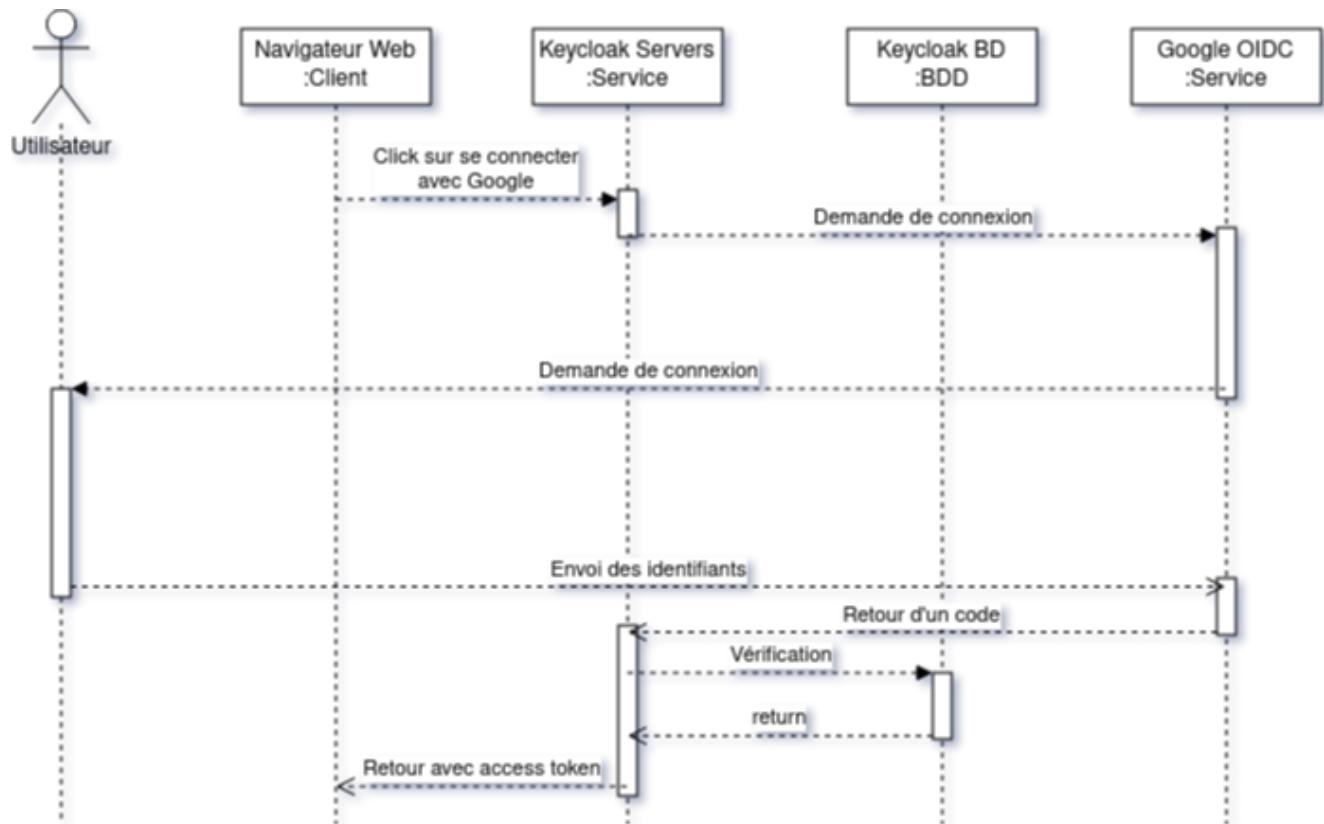


Figure 9. Connexion avec un compte Google

Q3: Communication Inter micro-services

Quand nous regardons les microservices que nous avons défini plus tôt nous pouvons remarquer que grand nombre d'entre eux devront implémenter un CRUD tel que le microservice des utilisateurs, des modules etc., il est donc nécessaire d'utiliser un protocole de communication synchrone afin de faire remonter une erreur en cas de problème au niveau des bases de données. Pour ce faire ces services vont utiliser le protocole HTTP avec des endpoint REST.

Pour les microservices qui ne sont pas des CRUD comme le microservice d'envoi de mail, il est plus intéressant d'utiliser un protocole de communication asynchrone. En effet, si nous prenons toujours l'exemple de l'envoi de mail, il est inutile de faire remonter une erreur au niveau de l'API gateway si le mail n'a pas pu être envoyé, ce dernier pouvant être renvoyé, il est préférable de simplement logger l'erreur et de continuer le traitement.

Dans un premier temps il a été envisagé d'utiliser la technique du http long polling. Cette technique consiste à faire de longues requêtes HTTP en boucle jusqu'à ce que le serveur renvoie une réponse, ce qui permet de simuler une communication asynchrone. Dans notre cas c'est le microservice d'envoi de mail qui va faire les requêtes HTTP jusqu'à ce qu'une réponse soit reçue. Cette approche est très simple conceptuellement, dépendant elle présente beaucoup trop de désavantages pour être utilisée. En effet un *proof of concept* [[poc_q3_lp](#)] ainsi que le document RFC6202 "Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP" [[rfc6202](#)] publié en avril 2011, mettent en évidence de nombreux problèmes liés à cette technique dont les plus importants sont:

- Une consommation importante de ressources côté serveur, en effet le serveur doit maintenir une connection ouverte avec le client et doit traiter les requêtes HTTP même si aucune réponse n'est attendue.
- Une certaine latence, même si l'envoi de mail est un processus long, il est préférable de réduire la latence de nos processus au maximum.
- Les timeout pouvant poser problème dans la mesure où les proxys peuvent fermer la connexion avant que le serveur ne renvoie une réponse.

Suite à un deuxième *proof of concept* [[poc_q3_ws](#)], une autre solution est d'utiliser des websockets pour communiquer avec ces services, en effet les websockets permettent des communications asynchrones entre nos services qui peuvent passer à travers un proxy. Hélas cette solution admet deux gros problèmes :

- Il est impossible pour l'un des deux parties de savoir si l'autre est indisponible ce qui pose problème dans un environnement en picroservice où tout service peut s'arrêter à tout moment. Une implémentation pour relancer le socket s'impose donc.
- Les connections étant persistentes, un scaling horizontal est tout bonnement impossible, les deux parties devant absolument être les mêmes.

Une solution plus simple et posant le moins de soucis reste des appels HTTP vers une API REST que notre fil d'exécution principal n'attendra pas, même si cette solution est moins *fault tolerant* qu'une message queue, elle a le mérite d'être très simple à impémenter (cf. [[poc_q3_http](#)]), là où du RPC nécessiterait des fichiers de définition. De plus cela permet d'éviter de faire de nos microservice des

hybrides API REST/Serveur long polling ou API REST/Serveur websocket comme vous pouvez le voir dans les premiers *proof of concept* [poc_q3_ws] [poc_q3_lp]. Cependant cette approche pose quelques problèmes, premièrement les requêtes HTTP embarquent des Headers qui peuvent s'avérer lourds et qui peuvent donc surcharger le réseau pour rien. La seconde raison est que les fonctions asynchrones sont traitées différemment en mémoire par les langages, par exemple NodeJS va les placer dans une file d'attente, Java va les exécuter dans un autre thread etc. Faire ainsi s'avère risqué si les requêtes sont nombreuses car cela va surcharger la mémoire et le processeur.

En voyant les désavantages majeurs de ces deux solutions il est clair que le protocole de communication en lui-même doit être asynchrone et léger, ce que n'est pas HTTP. C'est donc pour cela que nous allons nous tourner vers les message brokers. Il s'agit de serveurs utilisant un protocole qui fonctionne comme la poste: un expéditeur va envoyer un message à un message broker, ce dernier va le stocker et le transmettre à un ou plusieurs destinataires. Ici nous n'avons aucune fonction asynchrone à gérer, le message broker s'occupe de tout, de plus, les messages envoyés ne contiennent pas ou très peu d'erreurs ce qui les allège.

Les deux message brokers les plus répandus sont RabbitMQ et apache Kafka. Bien que les deux semblent similaires sur le papier ils sont en réalité très différents.

Premièrement leur manière d'envoyer les messages est différente, Kafka repose sur un simple système de *Publisher/subscriber* alors que RabbitMQ lui dispose de différents types d'émetteurs appellés *Exchanges* qui vont envoyer les messages à des *Queues* en fonction de clés de routages, ce qui permet d'avoir plusieurs options à notre disposition. Pour ce qui est du stockage des messages Kafka va les stocker pour une certaine durée de temps, alors que RabbitMQ va les stocker jusqu'à ce qu'ils soient consommés, ce qui permet d'économiser de l'espace de stockage.

Avec ces deux considérations ainsi que d'autres, il est plus communément acquis que Kafka sera plus adapté pour streamer des données en temps réel alors que RabbitMQ est plus adapté pour l'envoi de messages tel que des événements. C'est donc pour cela que nous choisirons RabbitMQ pour assurer la communication entre nos services.

"Hello, world" example routing

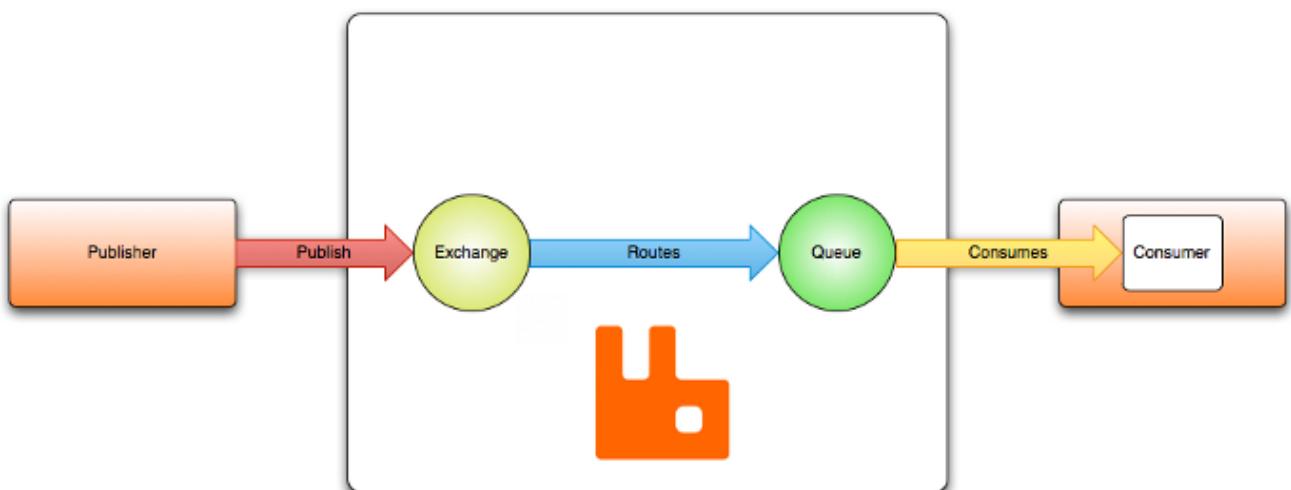


Figure 10. Schématisation du message broking avec RabbitMQ

Pour illustrer notre solution faisons un diagramme de séquence en prenant l'exemple de quelqu'un

s'inscrivant à Polycode:

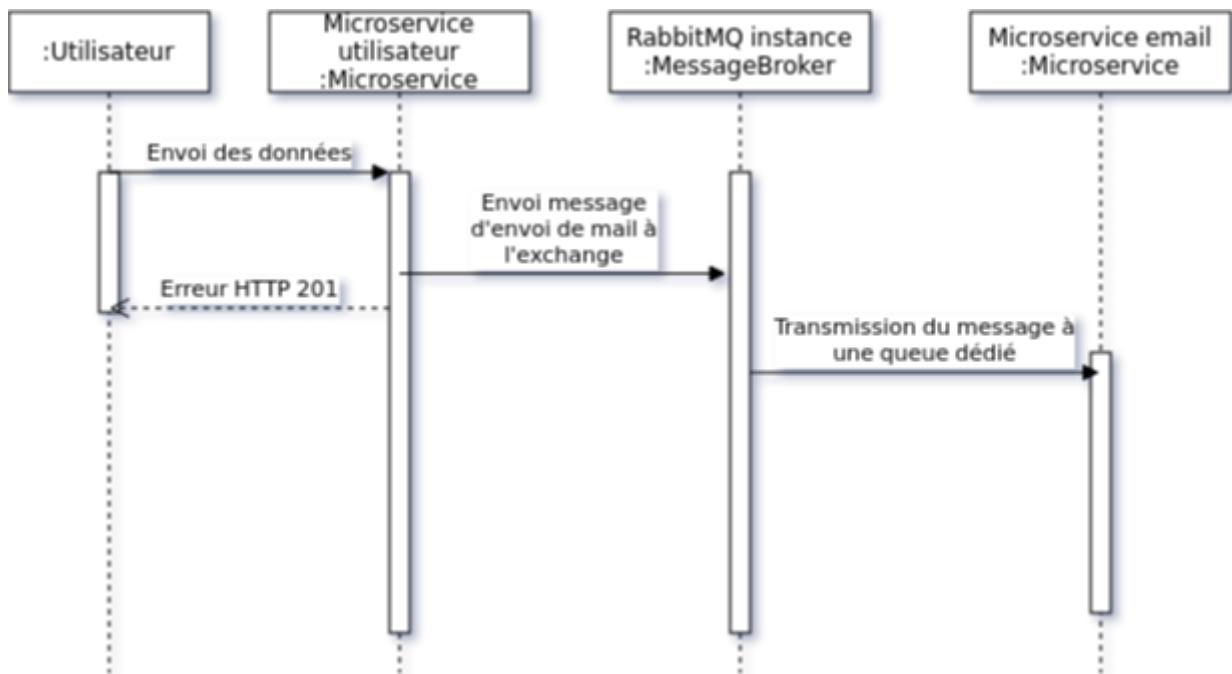


Figure 11. Diagramme de séquence d'une inscription

Q4: Comment tracer les logs et les requêtes?

Le traçage distribué est une méthode employée pour suivre le parcours d'une requête dans un système distribué comme dans le cas d'une architecture en microservices. Il permet de suivre les requêtes et les réponses entre les différents services.

Une solution de traçage distribué va marquer une requête de l'utilisateur avec un identifiant unique et la transmettre à chaque service qui la reçoit. Chaque traitement effectué sur la requête va ajouter des informations tel que le nom du service, le temps de traitement, etc. Zipkin est une solution qui va collecter les informations de traçage et de présenter les données de façon compréhensible. Zipkin est composé de 3 composants :

- Un serveur qui va collecter les informations de traçage et les stocker dans une base de données.
- Un client qui va ajouter des informations de traçage à chaque requête.
- Une interface web qui va permettre de visualiser les informations de traçage.

On peut associer une base de données au serveur Zipkin pour stocker les informations de traçage.

Voici l'architecture proposée pour l'intégration de Zipkin dans notre application :

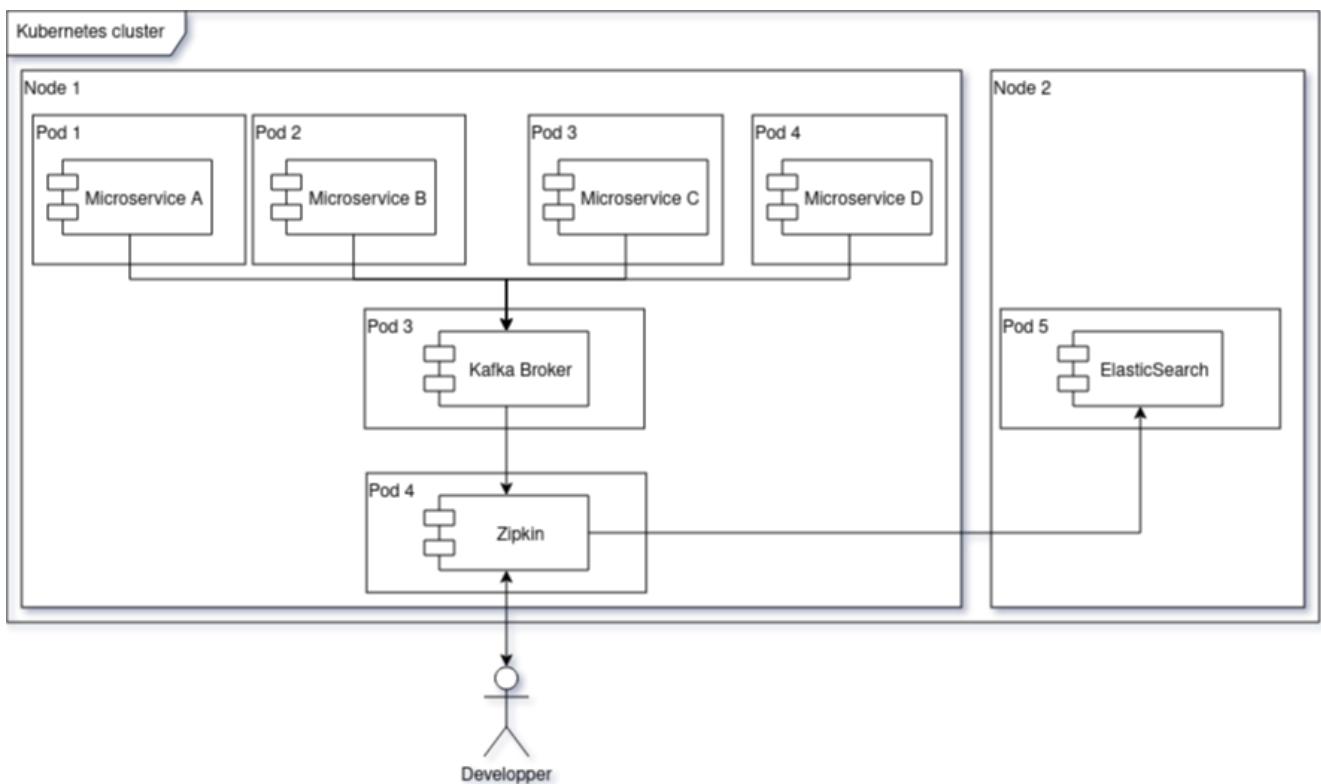


Figure 12. Proposition d'architecture pour l'intégration de Zipkin

Nous aurions pu utiliser Jaeger à la place de Zipkin, cependant Zipkin supporte plus de langages.

Les logs seront stockés dans une base de données Elasticsearch, ses performances permettent le stockage et la lecture de nombreux logs. Malgré son efficacité nous prendrons bien soin de la placer sur un autre noeud afin que ses opérations ne perturbent pas d'autres pods.

Voici un diagramme de séquence qui illustre le fonctionnement de Zipkin et d'Elasticsearch lors

d'une création de compte :

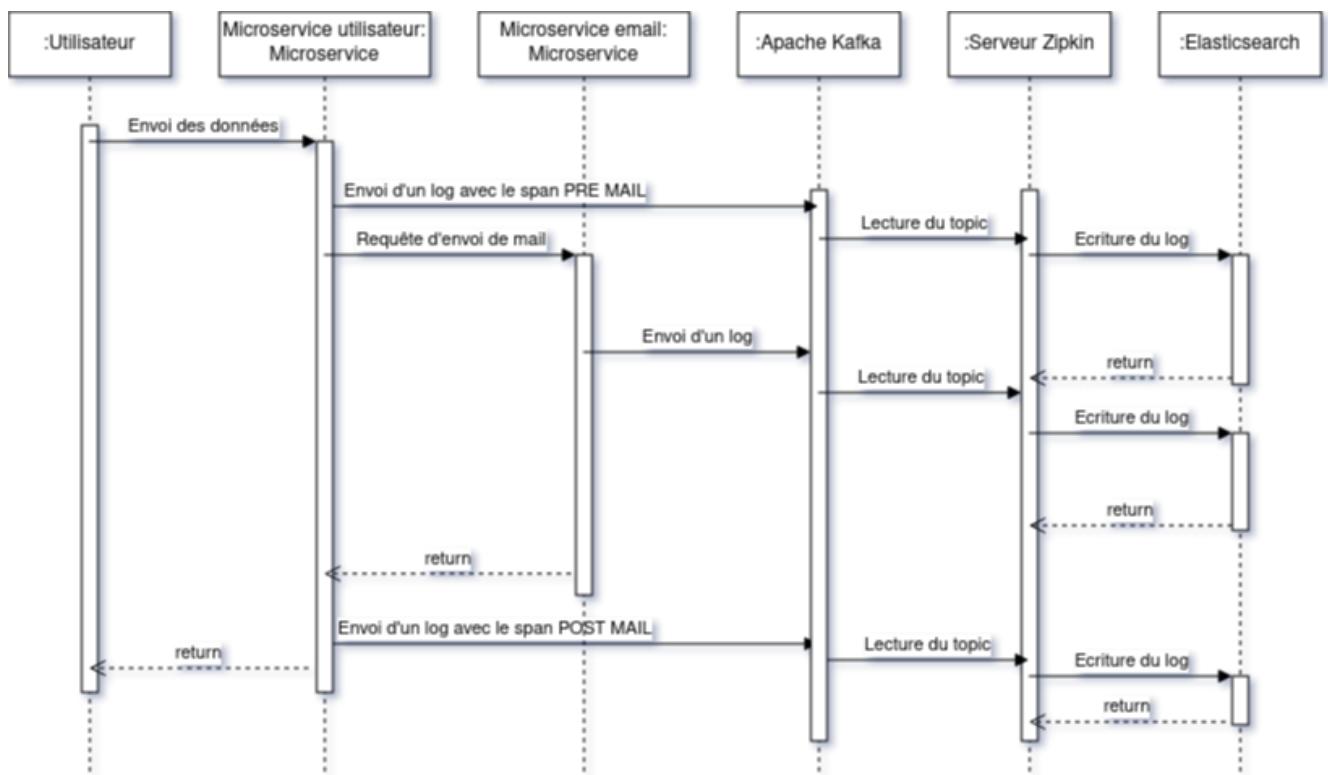


Figure 13. Diagramme de séquence décrivant la création de compte

Apache Kafka est utilisé dans cette configuration pour envoyer les logs à Zipkin, le but de cette utilisation est d'envoyer des requêtes de manière asynchrone afin de ne pas ralentir les microservices utilisateur et d'envoi de mail, ainsi une inscription sera rapide pour l'utilisateur.

Q5: Comment ajouter un moteur de recherche?

Moteur de recherche

Afin de pouvoir rechercher du texte grâce à une barre de recherche, nous devons d'abord identifier les données pouvant être sujets à une recherche. À l'heure actuelle nous souhaitons pouvoir retrouver un exercice ou un cours à partir du texte qu'il contient, à l'heure actuelle les cours de Polycode sont organisés avec cette structure.

Classes impliquées dans la recherche

```
class Module {  
    id: uuid;  
    name: string; // Champ sujet à recherche  
    description: string; // Champ sujet à recherche  
    type: 'challenge' | 'practice' | 'certification' | 'submodule' | ...;  
    // ...  
    contents: Content[];  
    modules: Module[];  
    tags: string[]; // Champ sujet à recherche  
}  
  
// ...  
  
class Content {  
    id: uuid;  
    name: string; // Champ sujet à recherche  
    description: string; // Champ sujet à recherche  
    type: 'exercise' | 'lesson' | ...;  
    rootComponent: Component;  
    // ...  
}  
  
// ...  
  
class Component {  
    id: uuid;  
    type: 'container' | 'editor' | 'quizz' | 'markdown';  
    // ...  
    components: Component[];  
    markdown: string; // Champ sujet à recherche  
}
```

Les champs name, description, tags et markdown sont des champs sur lesquels l'utilisateur pourra effectuer des recherches full text search. Cependant pour éviter d'avoir à indexer plusieurs champs d'un même document, le microservice module se chargera d'insérer un document dans une autre

base de données contenant la concaténation de ces champs et de l'identifiant de l'objet en question. Ce même microservice se chargera aussi d'effectuer les recherches.

Structure du document dans la base de données de recherche

```
type IndexedDocument = {
    id: uuid;
    text: string; // le texte concaténé

    // ces champs ne seront jamais utilisés pour une recherche
    data : {
        type: 'module' | 'content' | 'component';
        name: string;
        description: string;
    };
};
```

La base de données en question sera Elasticsearch, cette dernière utilisant le moteur apache Lucene, il nous sera possible de faire des recherches en full text search sur le texte de chaque module, contenu et composant. Tout comme pour les logs nous placrons cette base de données sur une autre machine physique, séparée des autres services pour ne pas interférer avec eux.

En cas de nouvelles attentes concernant la recherche (par exemple la recherche d'utilisateurs), il serait envisageable de créer un microservice dédié à la recherche. Nous ne le créerons pas pour l'instant car seuls les modules, contenus et composants sont concernés par la recherche et que cela violerait le principe YAGNI (You Ain't Gonna Need It)

Nous avons donc ces diagrammes de séquence :

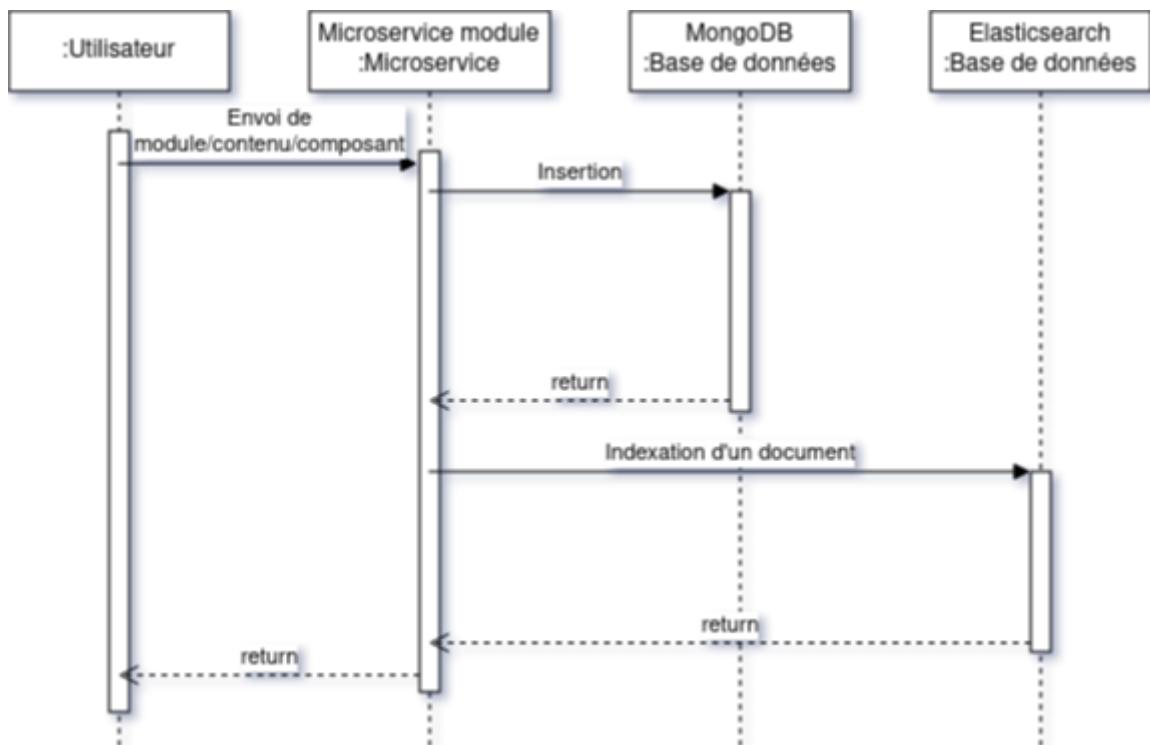


Figure 14. Diagramme de séquence décrivant l'indexation d'un module

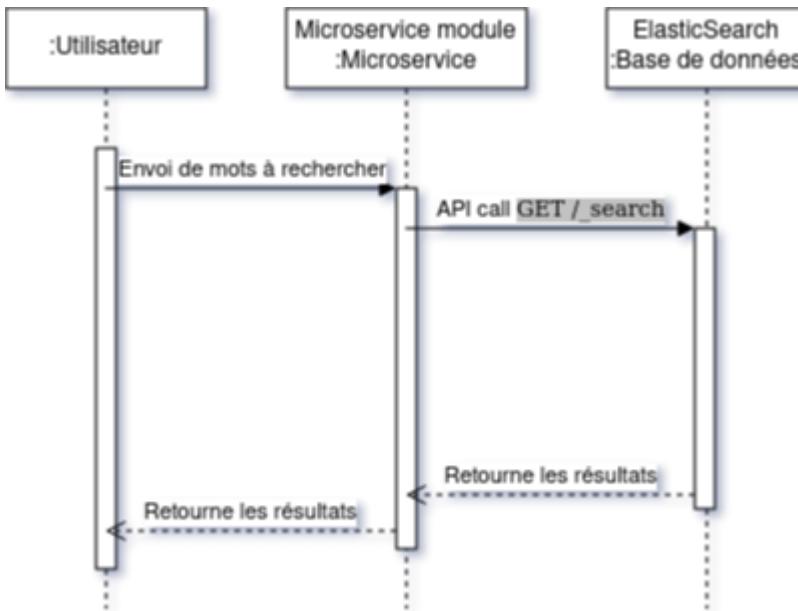


Figure 15. Diagramme de séquence décrivant la recherche par mot clé

Intégration dans le frontend

Concernant le frontend nous pouvons imaginer une barre de recherche dans l'en-tête de la page. Qui une fois clickée ouvrira une fenêtre modale avec un champ de recherche. Cela permettra de pouvoir rechercher sans avoir à se soucier des composants de la page.

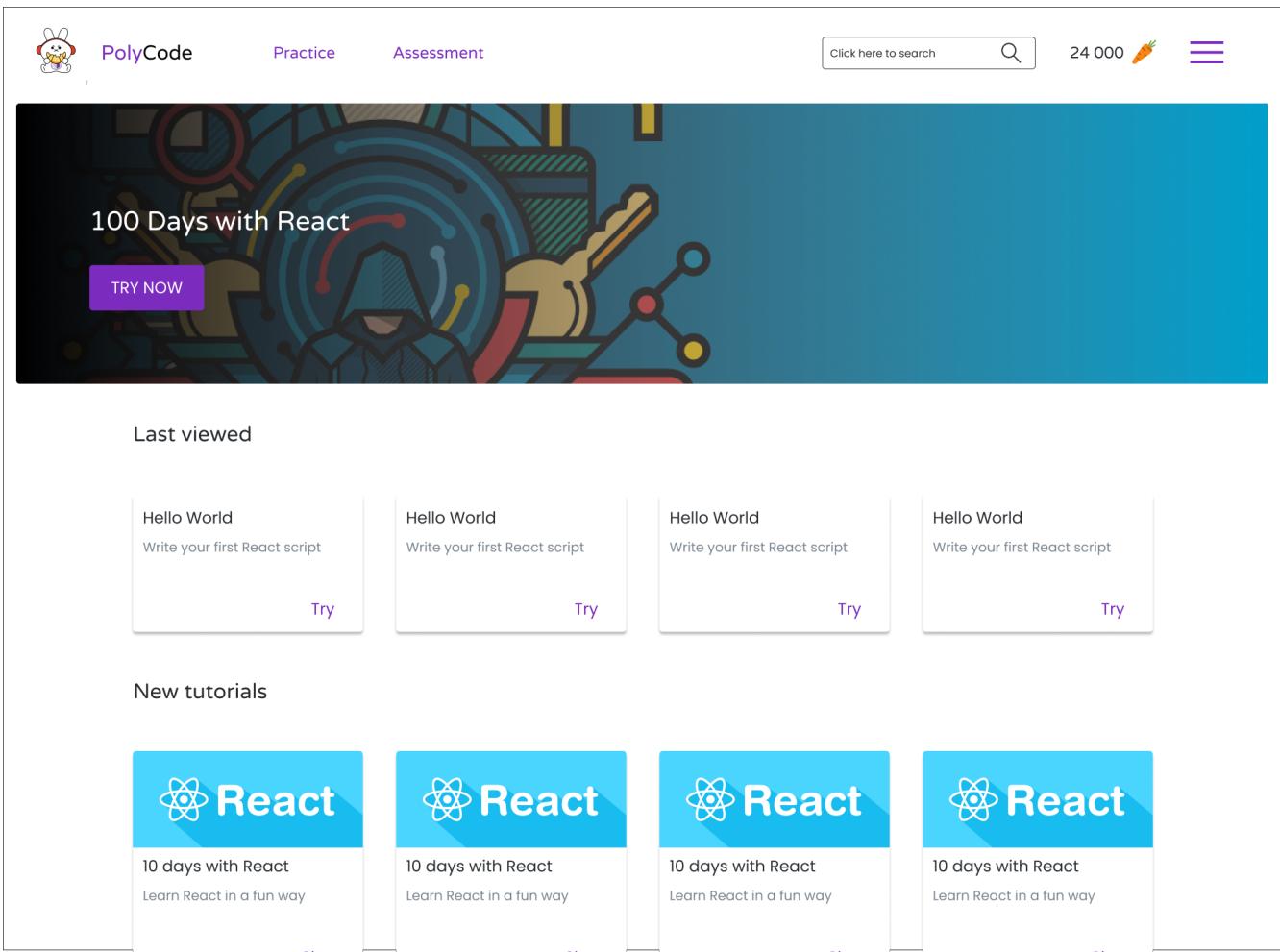


Figure 16. Proposition d'intégration de la barre de recherche

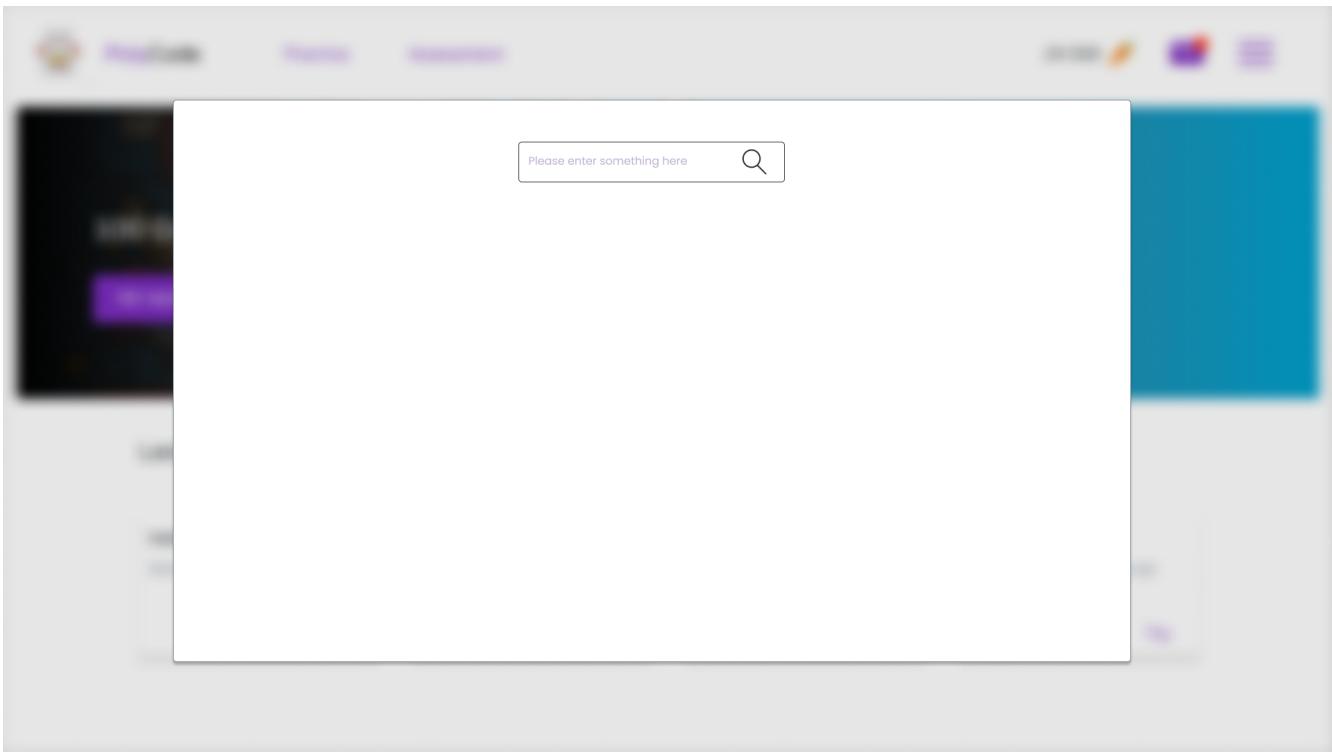


Figure 17. Proposition d'interface de recherche

Quand l'utilisateur rentre du texte une requête est envoyée au microservice de modules qui va renvoyer les résultats. Les données contenues dans le champ `IndexedDocument.data` permettront de

détailler les résultats de la recherche.

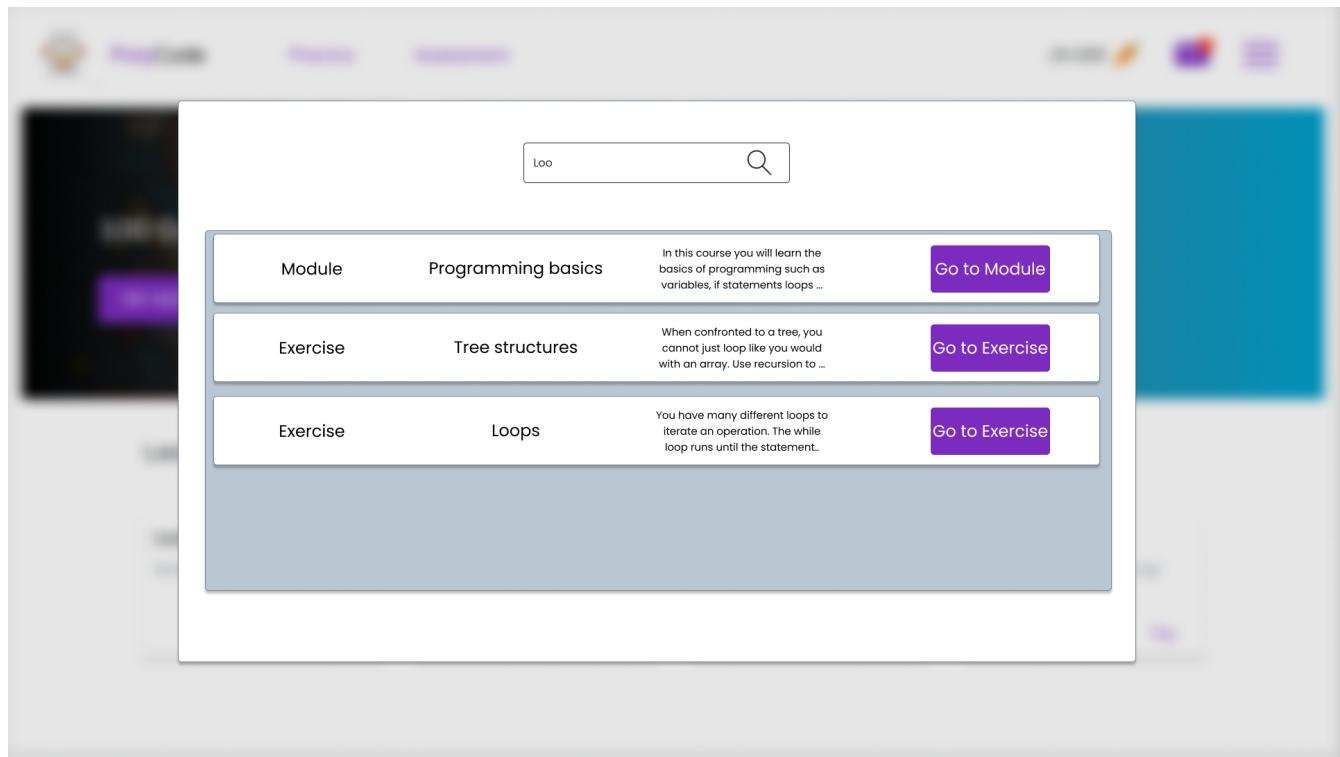


Figure 18. Proposition d'interface de résultats de recherche

Q6: Comment mettre en place l'architecture de runner

Définition du runner

Le runner est un élément central dans le fonctionnement de Polycode. Il s'agit d'une API à laquelle on donne du code, un language et des paramètres à passer par l'entrée standard, elle est chargée de créer des environnements isolés (conteneurs, machines virtuelles etc) qui vont se charger d'exécuter le code en renvoyant le résultat de l'exécution sur leur sortie standard. C'est cette sortie standard qui sera comparée aux données du validateur pour déterminer si l'exercice a été réussi ou non.

Architecture

La première chose à laquelle on peut penser en parlant d'isolation c'est de privilégier la création de machines virtuelles qui, contrairement à ce que l'on pense, peuvent démarrer relativement rapidement. Ce besoin de d'isolation est important car il faut éviter que le code de l'utilisateur puisse interférer avec l'infrastructure avec d'autres processus, isolation que Docker ne peut pas garantir car les conteneurs docker sont des processus isolés avec `iptables` et `cgroup`.

En plus d'une isolation du code, il faut aussi isoler le réseau pour empêcher l'utilisateur d'utiliser du code qui pourrait cibler nos services internes, c'est pour cela que nous empêcherons les runners d'envoyer des paquets réseau via un pare feu.

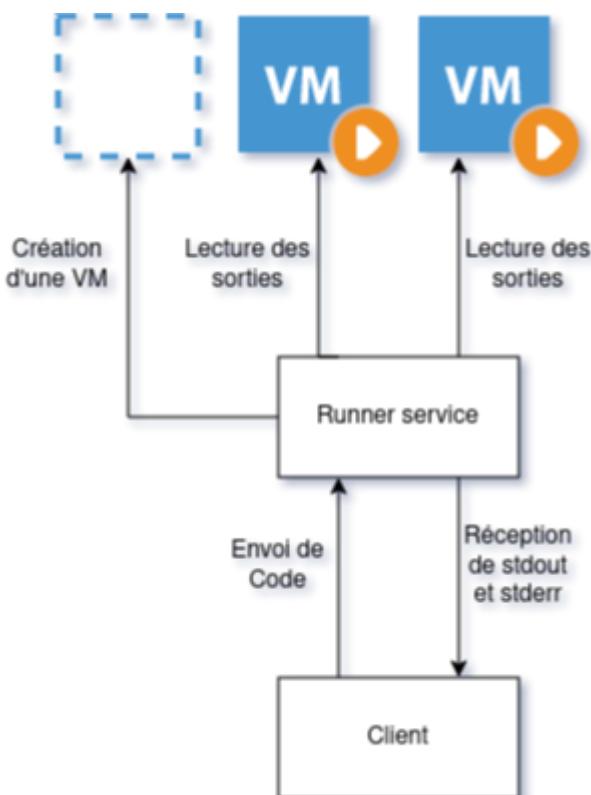


Figure 19. Schématisation du service de runner

L'équipe de Polycode a déjà développé une solution nommée [\[lambdo\]](#), un runtime écrit en rust qui

s'occupe de lancer des machines virtuelles, ces machines sont équipées d'un agent qui va exécuter une séquence de commandes qui lui sont envoyées tel que des commandes pour installer des dépendances, compiler le code de l'utilisateur mais surtout l'exécuter. Cet agent va en plus écouter la sortie standard et la sortie d'erreur produites par le code exécuté et le lui retourner.

Enregistrement des runners

Afin de pouvoir contacter les runners pouvant être sur plusieurs machines virtuelles nous aurons besoin d'un register qui gardera en mémoire l'adresse des différents services. Nous utiliserons Consul par HashiCorp qui permet de faire de la découverte de service, les runners s'enregistreront auprès de consul et consul nous permettra de les récupérer, l'avantage de Consul est que les proxy qu'il génère supportent de base la répartition de charge, il sera donc plus facile de distribuer les services runners sur plusieurs machines. Le même raisonnement s'applique sur nos autres microservices.

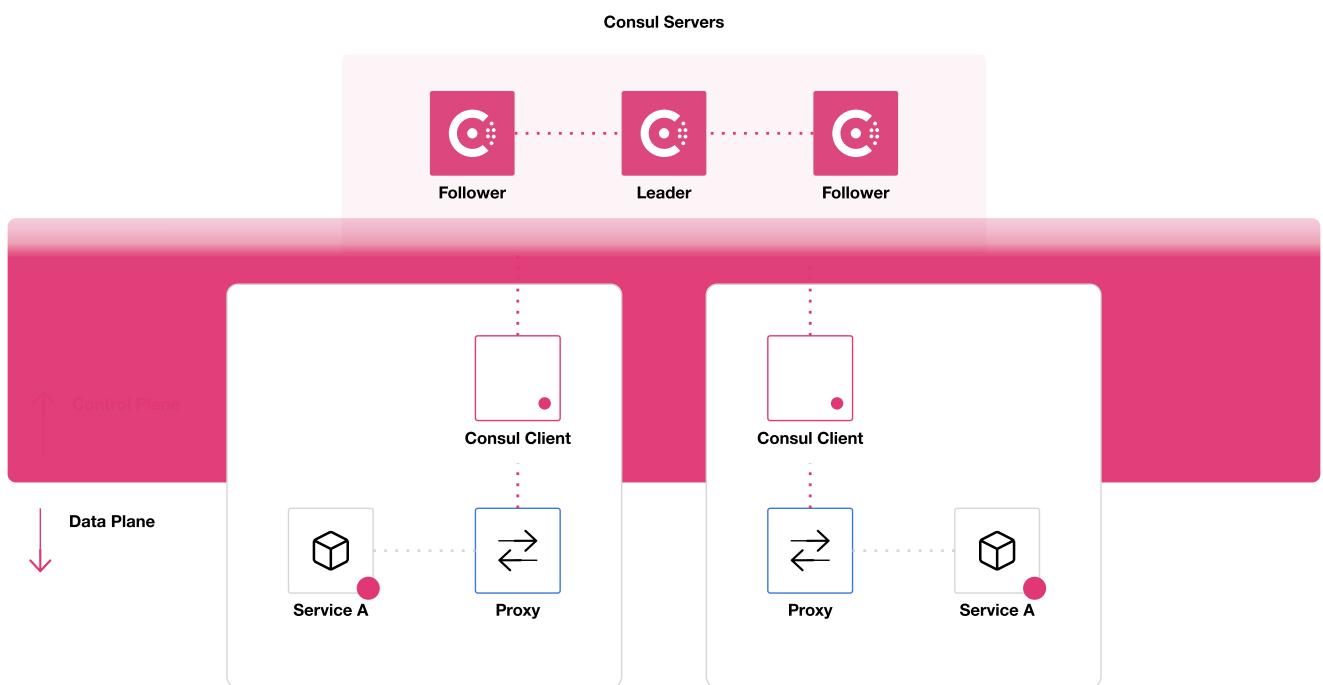


Figure 20. Schéma issu du site internet de Consul

Q7: Comment gérer les données?

Organisation des bases de données

Quand on se pose la question de l'organisation des données la première question que l'on peut se poser est "comment vont être associés les services et les bases de données?", pour répondre à cette question on a deux grandes approches: Le pattern "shared database" et le pattern du "Database per service".

La première aproche consiste à mettre en place une de base de données (ou un cluster) dans laquelle tous les services viendraient écrire et lire. Cette manière d'organiser sa base de donnée permet d'avoir des services qui sont aisément capables de faire des jointures si besoin, la source de donnée étant unique.

Cependant ce choix de design n'est adapté qu'à des architectures monolithiques, il est même considéré par beaucoup comme un anti-pattern. En effet la base de données consistera en un *Single point of failure* si la base de données plante, l'entièreté du système sera paralysé. De plus cela entrave le développement des services car tous les services seront développés pour un seul type de base de données. Ces pour ces raisons que nous rejetons instantanément ce pattern.

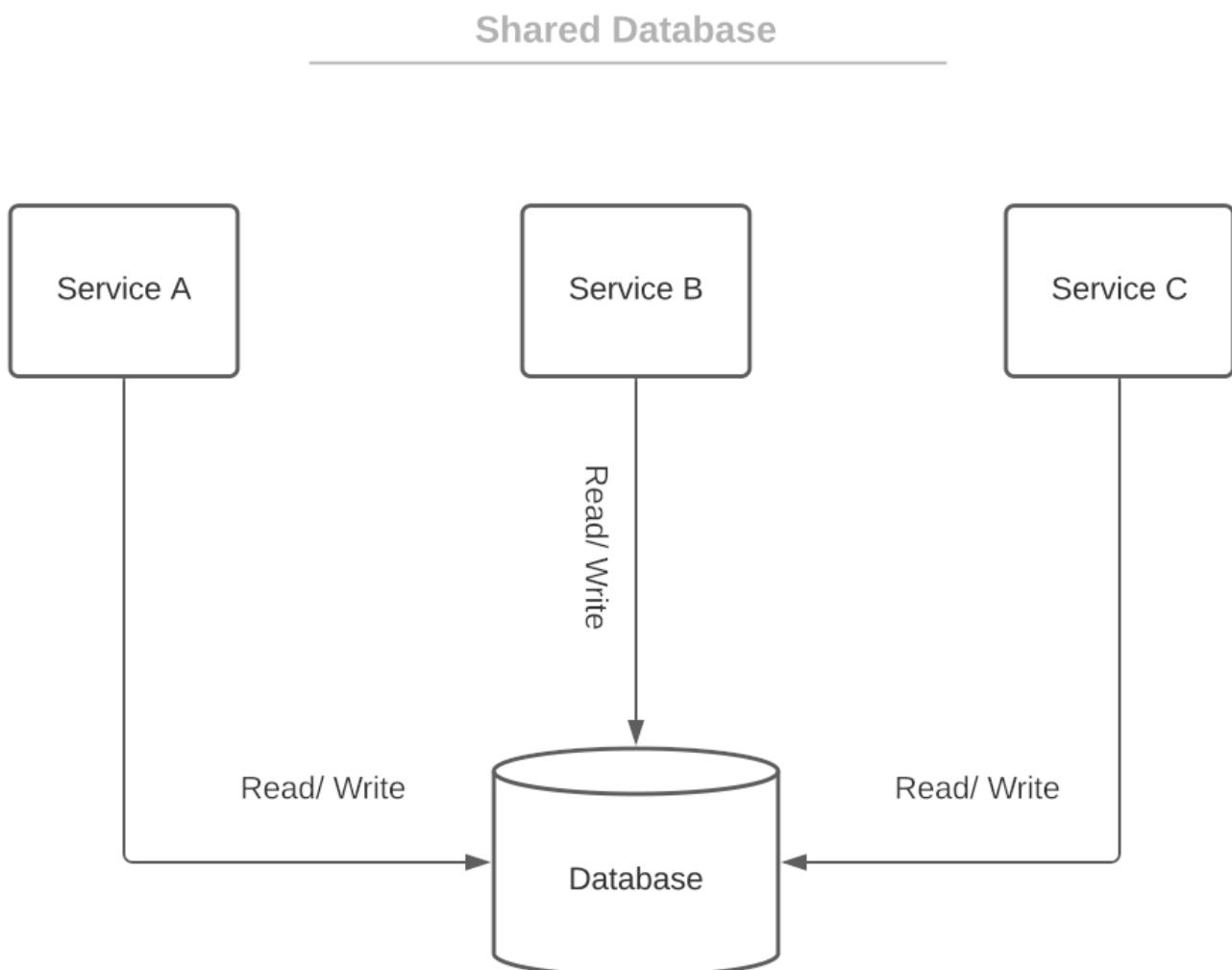


Figure 21. Shared database pattern

À l'exact opposé nous avons le pattern *database per service* dans lequel les instances de chaque service ont à disposition une base de données dédié, inaccessible par d'autres services. Ce découplage entre les services permet d'éviter de surcharger une seule base de données et donc réduit les latences de chaque service. De plus cela permet d'avoir une application Polyglotte, c'est à dire que grâce à cette séparation des services vis à vis de leurs données, nous pourrons avoir des services utilisant différents SGBD en fonction de nos besoins.

Au vu des avantages du pattern *database per service*, nous allons l'implémenter dans notre architecture, à la seule différence que nos services seront reliés à des clusters de base de données, ce qui permettra de répartir la charge et d'avoir des données cohérentes.

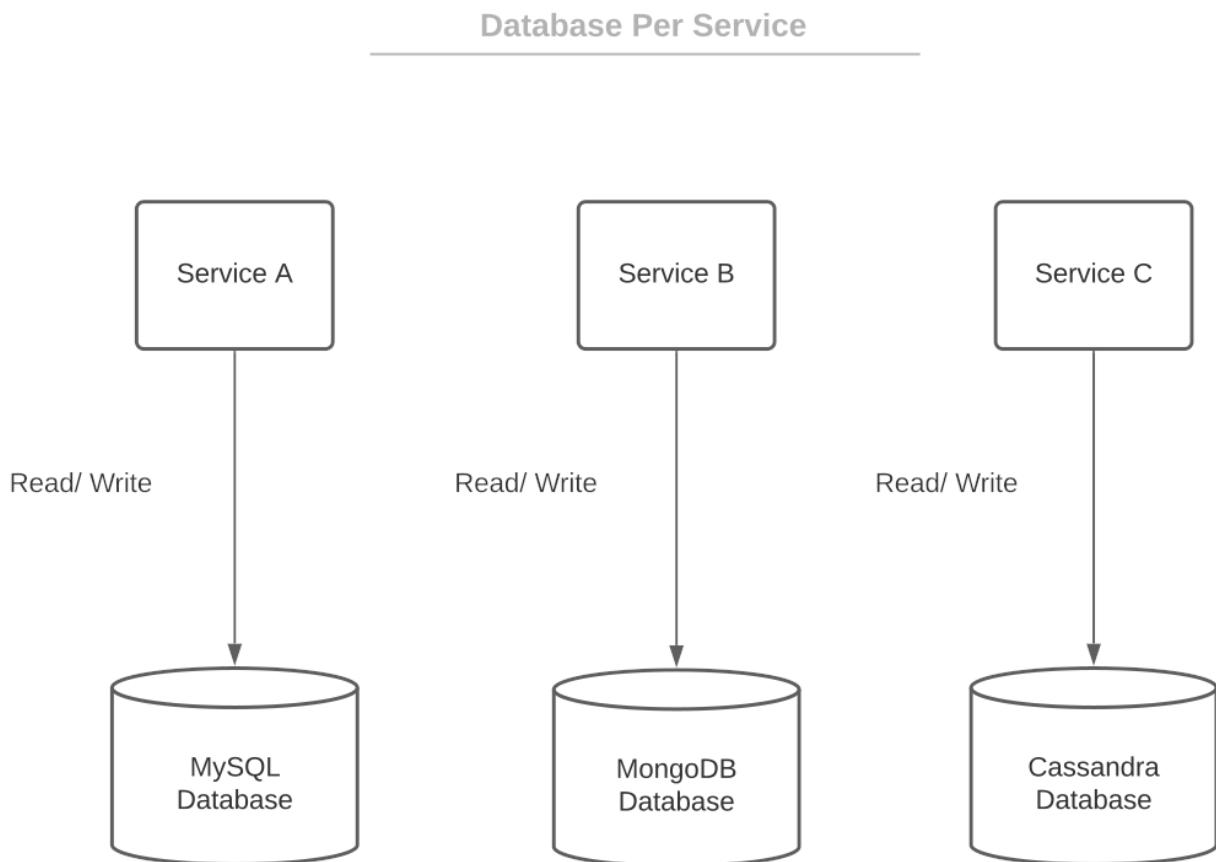


Figure 22. Database per service pattern

SGBD utilisés

Nos services utiliserons différents types de base de données en fonction de nos services:

- Gestion des utilisateurs : Relationnelle. Cette base de données stockera les utilisateurs, leurs adresses email, leurs paramètres ainsi que les transactions effectuées en Polypoints. Nous voulons nous assurer que les données soient ACID car les informations stockées (notamment les adresses mails, les mots de passe ou encore les items achetés) sont assez importantes pour exiger une cohérence forte.
- Édition de modules : Documentaire. Les modules, composants, validateurs et composants ont été désignés pour être des éléments centraux et réutilisables de Polycode remplissant de nombreuses fonctions en effet ils sont utilisés dans le cadre des exercices, des évaluations et

dans les campagnes de test. Cette variété implique nécessairement des schémas qui vont varier. Les bases de données documentaires sont adaptées à ce genre de cas de part l'absence de schéma précis prédefini.

- Gestion de campagne de tests: Relationelle. Les campagnes de tests sont des structures de données auquelles sont rattachées les candidats ainsi que des tags. Nous souhaitons des données les plus cohérentes car il serait malheureux que des utilisateurs ne soient pas reliés à une campagne à cause d'un manque de cohérence.

```
## Campaign
- id: 'uuid'
- name: 'string'
- description: 'string'
- testId: 'uuid'
- creatorId: 'uuid'

# ...

## Candidate
- id: 'uuid'
- campaignId: 'uuid'
- userId: 'uuid'
- status: 'string{Confirmation Pending, Confirmed, Declined, Cancelled, Test Sent, Test Opened, Test In Process, Test Finished}'
- startedAt: 'Date'
- email: 'string'
- linkCode: 'uuid'
- linkExpiration: 'Date'
```

Pour les bases de données relationnelles nous utiliserons PostgreSQL. En effet ce dernier comparé à d'autres SGBD s'est imposé par ses nombreuses fonctionnalités et ses langages supportés malgré le fait qu'elle ne soit pas la plus rapide. Cependant le réel intérêt de postgres c'est sa fiabilité quant à l'exécution des transactions. En effet PostgreSQL tient un Write Ahead LOG (WAL), soit des logs écrits avant les transactions, ce qui permet de les re-exécuter si la transaction s'arrête inopinément comme par exemple dans le cas d'une panne. Ce journal peut être envoyé à d'autres instances pour former un cluster, le transfert peut se faire en streaming ou via FTP pour privilégier soit respectivement pour une réPLICATION asynchrone ou une réPLICATION synchrone.

Dans un cluster postgres nous avons deux types de noeuds: les noeuds maîtres qui se chargent des opérations d'écriture faisant preuve de source de vérité, et les noeuds esclave qui eux se contentent de copier les données des maîtres afin d'être opérationnels pour les opérations de lecture.

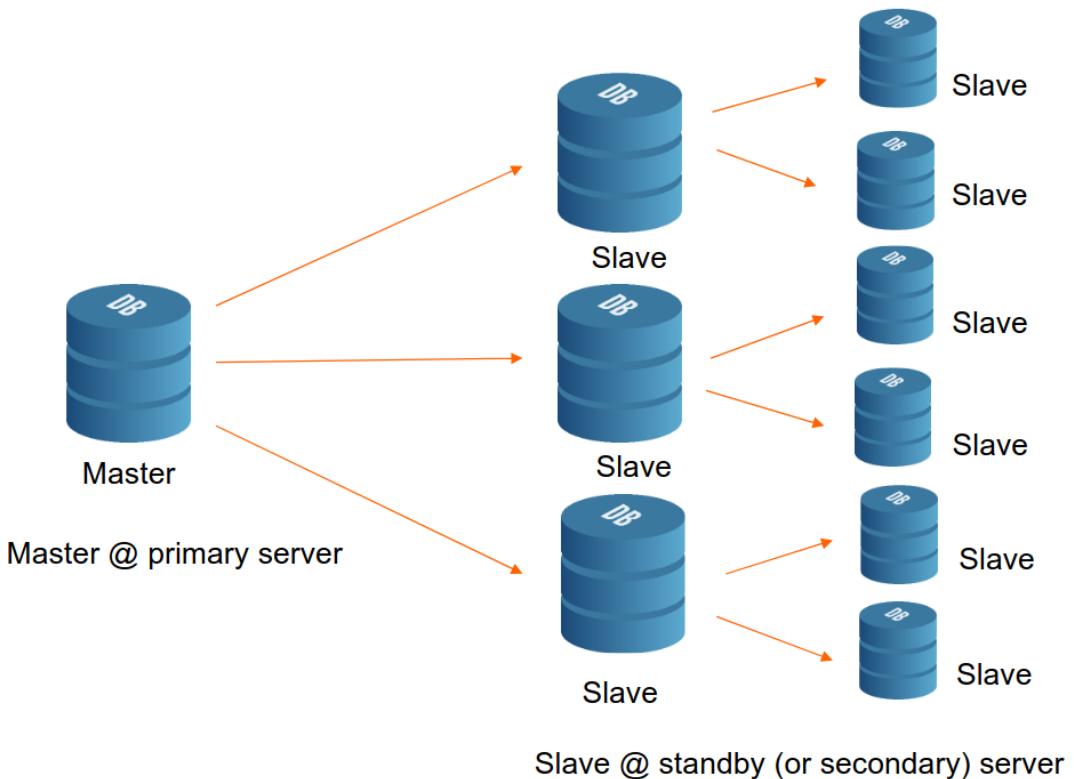


Figure 23. Exemple de cluster Postgres

Pour ce qui est de l'organisation du cluster nous avons deux choix:

- Avoir plusieurs noeud maîtres, ce qui permet une plus grande disponibilité et une bonne tolérance aux pannes quand à l'écriture des données en effet si un des noeuds maîtres tombe en panne, un autre peut prendre la relève. Cependant la présence de multiple sources de vérité peut amener à des conflits qui doivent être résolus.
- Avoir un seul noeud maître. Cette approche propose l'exact inverse de l'approche multi-maître: à savoir une absence de conflits mais une moins bonne disponibilité.

Nous souhaitons à tout prix avoir des bases de données hautement disponibles mais avec les données les plus cohérentes possibles. Nous allons donc répondre au premier besoin avec un cluster multi-maître, et au second point en utilisant le streaming du WAL, afin que les données soient mises à jour le plus fréquemment possible pour éviter des conflits.

Alternativement, il a été envisagé d'utiliser Apache Cassandra, une base de données orientée colonnes, souvent utilisée pour rapidement stocker de grands volumes de données, en effet chaque noeud d'un Ring est capable de contenir jusqu'à plus de 1 Teraoctet de données et faire énormément d'écritures par seconde. Néanmoins, au delà du fait Cassandra n'est pas une base de données *ACID compliant* de part l'absence de transactions et de transactions, il y a aussi une absence de jointure ce qui implique donc un refactoring de nos données mais aussi des données dupliquées pour chacune de nos relations n-aires ce qui peut être compliqué à gérer. Nous n'avons donc pas gardé cette solution.

Pour ce qui est des bases de données orientée documents 3 noms sont ressortis pendant mes recherches: CouchDB, CouchBase et MongoDB. Tous deux sont des bases de données orientées document très similaires mais ont quand même beaucoup de différences.

CouchDB est un projet de la fondation Apache qui vise à fournir une base de données orientée document, vouée à être accessible et manipulable via une API REST mais aussi à être ACID. Malgré cet avantage il y a des inconvénients majeurs comme le fait que les données soient stockées en format JSON (JavaScript Object Notation) qui est un format lourd, le fait que ce ne soit pas stocké en mémoire, l'absence de transactions la latence observée sur les grands sets de données. Nous rejeterons donc ce SGBD pour ces raisons.

Pour pallier à une partie de ces problèmes il existe Couchbase, un SGBD créé par la fusion de CouchOne, une base de données basée la base de données orientée document couchDB, et de membase une base de donnée clé-valeur stockée en mémoire. Elle a la particularité de stocker ses documents sur disque, mais aussi de placer les plus récents dans la mémoire RAM.

La solution pour laquelle nous avions optée dans les précédentes versions de Polycode est MongoDB. Il s'agit d'un SGBD orienté document dont la principale particularité est que le stockage des données est en BSON, qui un format de données binaire qui permet de stocker des données JSON de manière plus compacte.

MongoDB et couchbase ayant beaucoup de points communs nous allons les comparer dans le tableau suivant:

	MongoDB	CouchBase
Indexage	Indexage par arbre binaire. Temps moyen $O(\log n)$, temps dans le pire des cas $O(n)$	Indexage par skip list. Temps moyen $O(\log n)$, temps dans le pire des cas $O(n)$
Performances	Performances constantes peu importe la taille du cluster	Performances limitées dans un petit cluster mais augmentent rapidement avec la taille du cluster
Stockage des données	Stockage sur disque avec le format BSON	Les documents les plus récents ont droit à une copie en mémoire, les documents sont stockés dans des <i>virtuals Buckets</i> qui sont copiés sur plusieurs noeuds d'un cluster

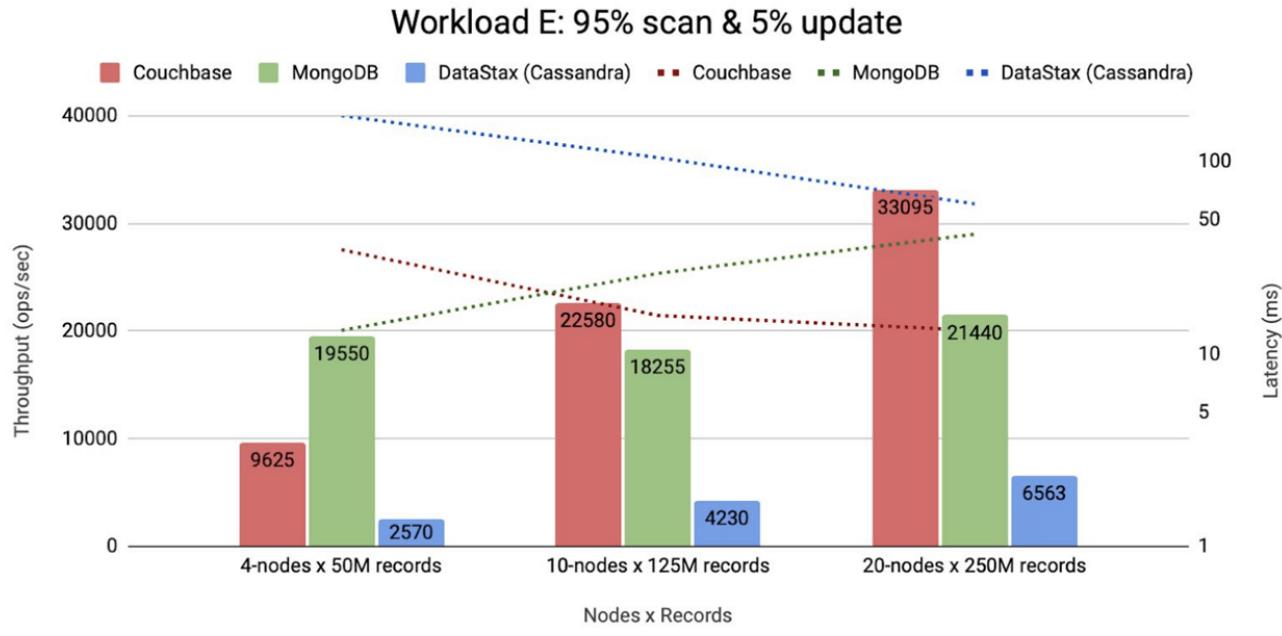


Figure 24. Comparaison entre CouchBase MongoDB et Cassandra

Concrètement couchbase est plus appropriée dans des environnements de grande taille, tandis que MongoDB est plus Polyvalent. Nous avons donc opté pour MongoDB de part cette polyvalence.

Schemas de données

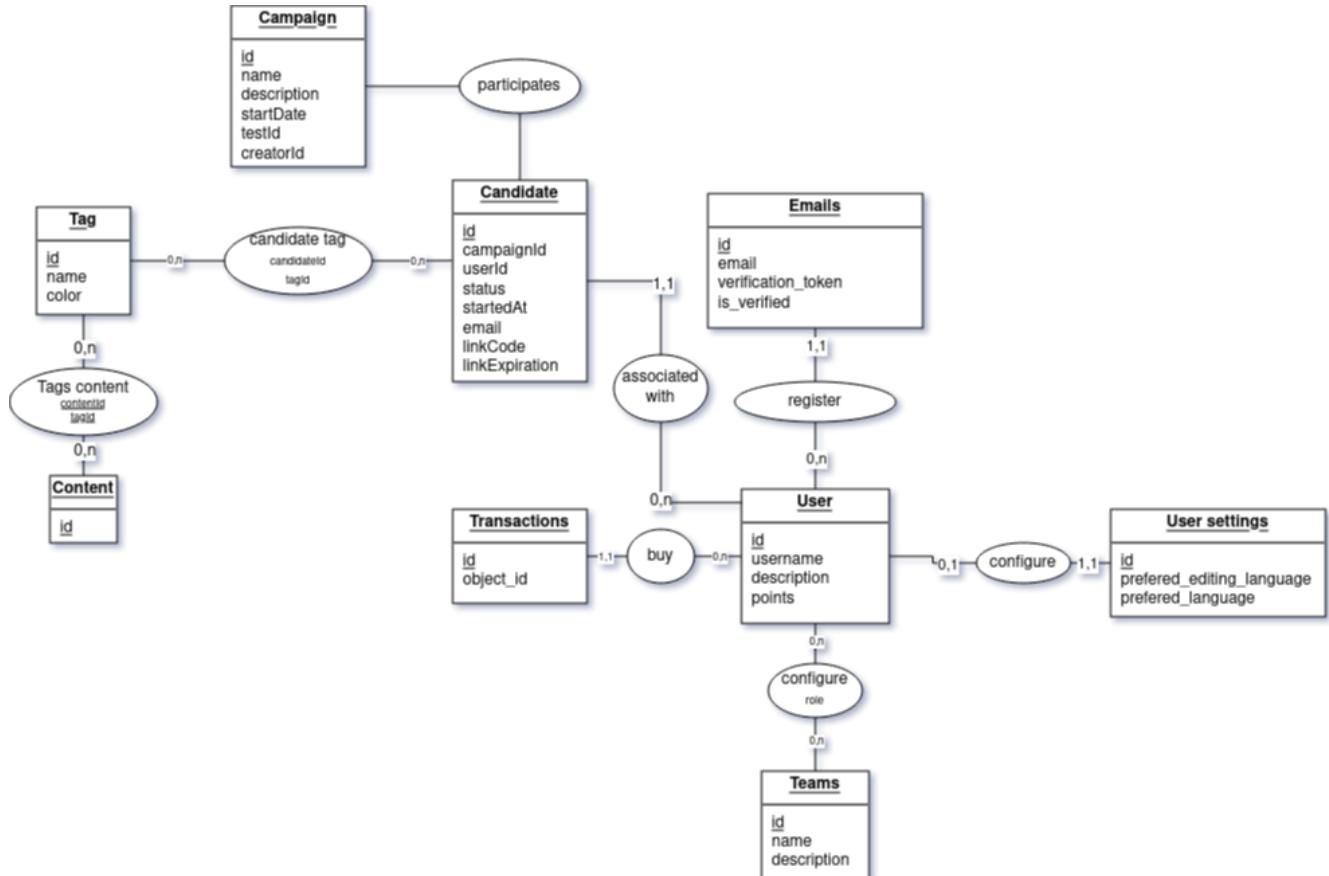


Figure 25. Modèle conceptuel de données

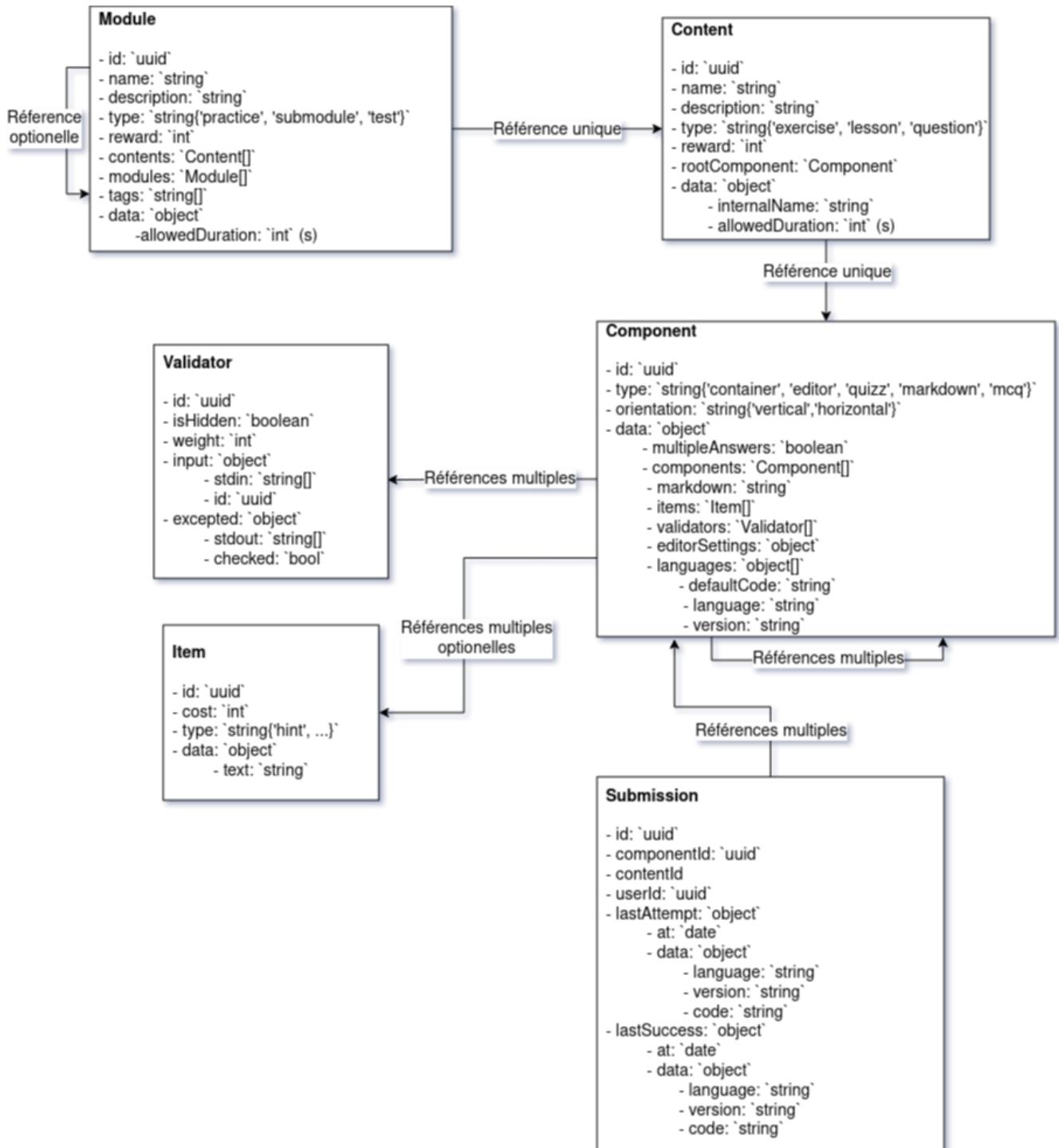


Figure 26. Représentation des données documentaire

A noter que pour éviter de dupliquer les données dans la base documentaire, toutes les sous documents sont stockés dans leur collection correspondants et référencées par leur identifiant unique au lieu d'être inclus directement dans leur document parent.

CQRS et Event Sourcing

Le *Command And Query Requests Separation* est un design pattern qui consiste à séparer les requêtes de lecture et d'écriture dans deux SGBD différents. Cela permet de permettre aux utilisateurs de lire des données sans être inquiétés par des ralentissements causés aux locks du SGBD, dûs à l'écriture.

L'*Event Sourcing* consiste à partir du principe que l'application traverse différents états grâce à des évènements que l'on va stocker, en définitive chaque état de l'application résulte d'une combinaison distincte d'évènements, comme une fonction bijective. l'avantage de ce système est que l'on peut facilement revenir à un état antérieur de l'application en rejouant les évènements qui ont mené à cet état. Cela permet aussi de facilement débugger une application en rejouant les évènements qui ont mené à un bug.

Il est possible de cumuler ces deux patterns en remplaçant la base de données destinée aux commandes par une base de donnée stockant les events et un *Event Handler* qui va les interpréter pour mettre à jour la base de données de lecture. Cela permet de garder le meilleur des deux mondes.

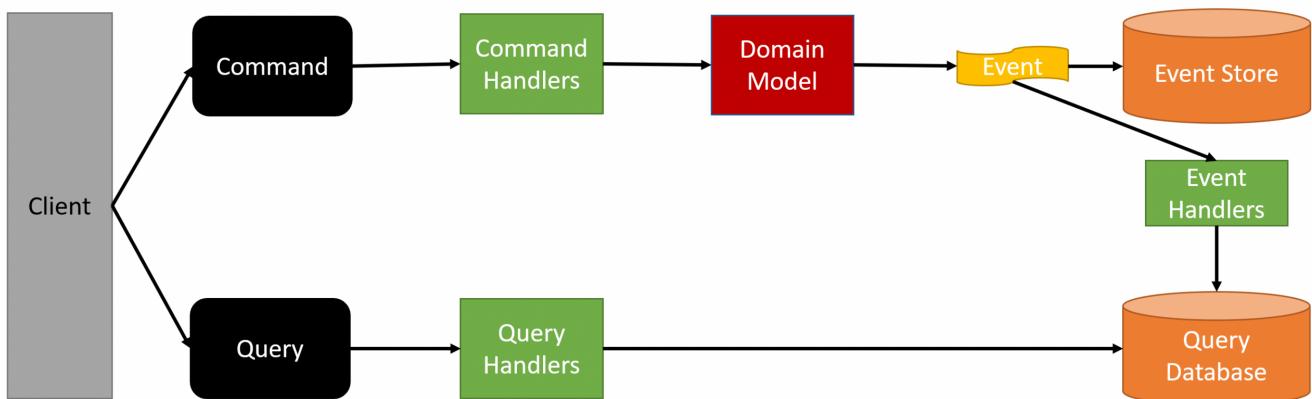


Figure 27. CQRS et Event Sourcing

Nous allons donc utiliser ce système pour notre application afin de garantir un maximum de performance et de fiabilité pour nos services CRUD, à savoir le service des utilisateurs, des modules et des campagnes. Lors de recherches sur les bases de données les plus adaptées pour ce pattern le nom de EventStoreDB [ESDB] est ressorti plusieurs fois, il s'agit d'un SGBD dans laquelle les évènements de l'application sont ajoutés dans des streams, les sdk permettent de lire ces streams et de les interpréter pour mettre à jour la base de données de lecture. Nous allons donc utiliser ce SGBD pour stocker les évènements.

Q8: Comment ajouter une application mobile dans le système

Fonctionnalités et mockup visuels

Dans le cadre d'une application mobile intégrée à Polycode, nous souhaiterions proposer une expérience la plus proche possible que ce que l'utilisateur peut avoir sur un ordinateur, tout en l'adaptant le plus possible au support Android.

C'est autour de cette ligne directrice que nous avons défini les fonctionnalités que nous souhaiterions implémenter dans notre application mobile.

Premièrement des fonctionnalités hors ligne seront proposées:

- Les informations de l'utilisateur notamment le nom d'utilisateur, ses adresses email, son nombre de polypoint, ses équipes et son rang seront stockées en local et synchronisées avec le serveur lorsqu'une connexion internet sera disponible.
- À chaque connexion à internet l'application va garder en cache une partie des modules et des contenus chargés afin de pouvoir les lire et de les compléter hors ligne.
- Le code écrit dans un éditeur de code par l'utilisateur sera stocké en local, afin d'être modifié même hors ligne.
- Un utilisateur pourra soumettre du code et des réponses aux exercices même hors ligne, l'application s'occupera d'envoyer les données lorsqu'une connexion internet sera disponible et notifiera l'utilisateur de ses résultats.

Comme nous entendu ci-dessus nous autorisons l'utilisateur à répondre aux contenus et notamment ceux de type éditeur de code sur téléphone. Pour faciliter la saisie de code, il sera mis en place un éditeur de code avec coloration syntaxique, autocomplétion ainsi qu'une barre horizontale placée au dessus du clavier de l'utilisateur, sur laquelle sera placée des caractères spéciaux fréquemment utilisés dans le langage de programmation contenu dans l'éditeur. En cliquant sur ces caractères spéciaux, ils seront automatiquement ajoutés à la position du curseur dans l'éditeur pour que l'utilisateur n'aie pas à naviguer dans les symboles de son clavier, tout comme le shell pour Android Termux.

Editeur de code sans focus du clavier

[Back](#)

```
class MyClass {
    static fileExtentions = new Map ([

        [".csv", CSVQueryGenerator],
        [".ttl", TurtleQueryGenerator],
        [".turtle", TurtleQueryGenerator],
        [".rdf", RDFXMLQueryGenerator],
        [".rdffxml", RDFXMLQueryGenerator],
        [".owl", RDFFMLQueryGenerator],
        [".jsonld", JSONLDQueryGenerator],
        [".json", JSONLDQueryGenerator]
    ])

    constructor (filePath, fileHandler) {
        const gen = Fill.fileExtentions.get(
            filePath.slice(filePath.lastIndexOf('.')))
        if (!gen) throw new Error('File type not
supported!')
        this.strategy = new gen(fileHandler)
    }
}
```

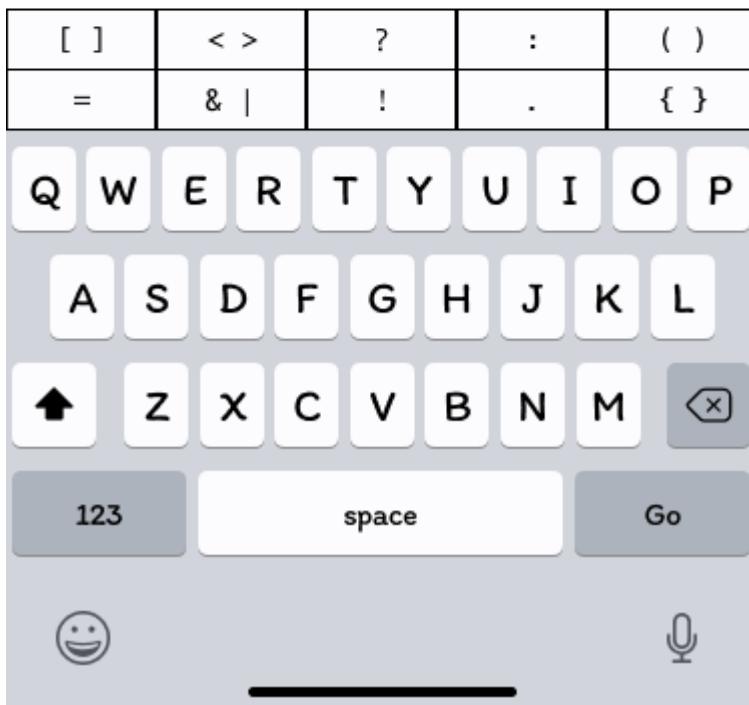
The screenshot shows a mobile application interface for editing and validating code. At the top, there is a dark header bar with a back arrow icon. Below it is a light-colored navigation bar with a back button labeled "Back". The main content area is divided into several sections:

- Code Editor:** A large dark text area containing the provided JavaScript code.
- Language Selection:** A dropdown menu set to "Javascript" with a downward arrow icon.
- Validators:** A section with two items:
 - #1 (click to reveal 1234 🥕) with a "Run" button.
 - #2 (click to reveal 1234 🥕) with a "Failed" button.
- Input/Output:** A section showing "Input: Some Input Here" and "Output: ereH tupnl emoS" with a "Done!" button.
- Hints:** A section with two items:
 - #1 (click to reveal 1234 🥕)
 - #2 Lorem ipsum dolor sit amet, consectetur a ...

Editeur de code avec focus du clavier

[Back](#)

```
static fileExtentions = new Map ([  
    [".csv", CSVQueryGenerator],  
    [".ttl", TurtleQueryGenerator],  
    [".turtle", TurtleQueryGenerator],  
    [".rdf", RDFXMLQueryGenerator],  
    [".rdffxml", RDFXMLQueryGenerator],  
    [".owl", RDFXMLQueryGenerator],  
    [".jsonld", JSONLDQueryGenerator],  
    [".json", JSONLDQueryGenerator]  
])  
  
constructor (filePath, fileHandler) {  
    const gen = Fill.fileExtentions.get(  
        filePath.slice(filePath.lastIndexOf('.')))  
    if (!gen) throw new Error('File type not  
supported!')  
    this.strategy = new gen(fileHandler)  
}  
async sendData(contai  
}  
}
```



Cependant cette fonctionnalité ne sera pas disponible pour des questions de lisibilité et de confort d'utilisation. Elle sera donc désactivée si l'application mobile juge que votre écran est trop petit, cet inconfort pouvant être considéré comme un désavantage comparé à un ordinateur, les utilisateurs ne pourront pas participer aux campagnes de tests via l'application mobile par soucis d'égalité. Ce problème de visibilité nous contraint aussi à restreindre les opérations CRUD de module et de contenu ainsi que d'empêcher l'utilisation des outils d'administration.

Message d'erreur en cas d'écran trop petit

[Back](#)

Unfortunately, your device
is too small to write code.

Message d'erreur en cas d'accès à un test

Back

PolyCode



For equity reasons,
assessments are disabled
on this platform. Go on PC
to do it.

Un mockup UI a été réalisé avec Figma afin de visualiser l'application mobile dans son ensemble.
Un aperçu est disponible à [cette adresse](#), vous pouvez naviguer dans le mockup grâce à [ce lien](#).

Flows de fonctionnement

Il y a 5 flows principaux dans l'application mobile:

- L'authentification et la création de compte

- La gestion de compte
- La navigation dans les modules et les contenus
- La gestion d'équipe
- Recherche de contenus et modules

Nous pourrions définir deux flows supplémentaires, du fait que nous pourrions ouvrir l'application à partir d'un lien:

- Ouverture d'un lien n'existant pas (erreur 404)
- Ouverture d'un lien référencant un test qui pour rappel est interdit aux utilisateurs de l'application mobile

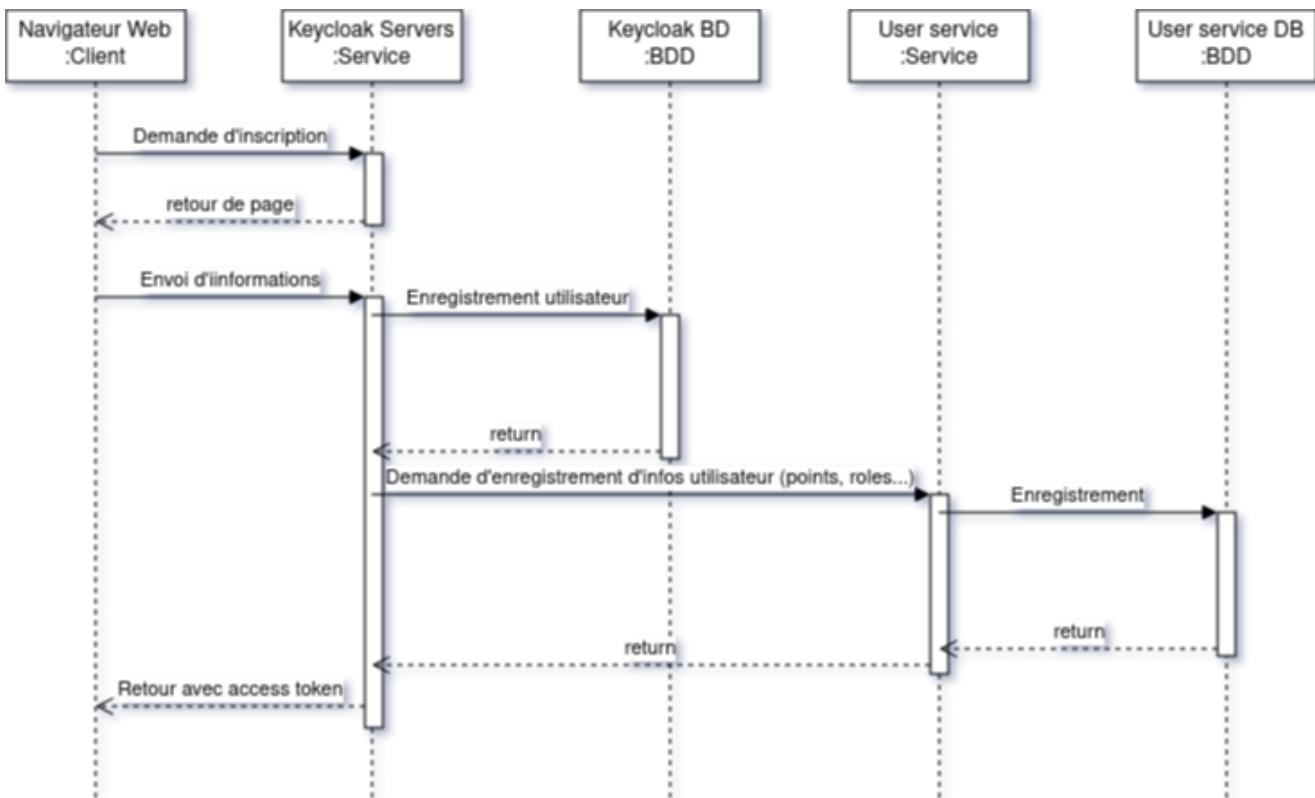
API

L'application mobile communiquera avec notre architecture grâce à une API Rest. Nous aurions pu privilégier GraphQL, un langage de requête orienté client, dans lequel le client formule auprès d'un serveur la forme de sa réponse. Notre application mobile n'étant pas orienté autour de la consommation de données, mais plutôt de l'envoi de données, nous avons préféré utiliser une API Rest, de plus le choix de l'API REST permet de mettre en cache les réponses grâce à l'en-tête HTTP **Headers**.

Comme vous pourrez le voir sur la page Swagger de notre API [\[api_swagger\]](#) que l'API consiste en l'API de la webapp privée de Routes **PATCH**, **PUT** et **DELETE** ces dernières étant des routes modifiant ou supprimant des ressources.

Authentification

Concernant l'authentification de l'application mobile nous allons garder la même séquence que pour une connexion normale à savoir l'utilisation du front end servi par keycloak, ce qui implique donc l'absence des routes API pour la connexion et l'inscription dans le swagger ci-dessus.



Q9: Comment gérer les problématiques de sécurité

Sécurisation du réseau

Pour sécuriser le réseau de notre application nous allons mettre en place une architecture *Zero Trust*, architecture dans laquelle chaque communication entre service est prédéfinie, authentifiée et sécurisée.

premièrement nous allons pouvoir prédéfinir les communications inter-service grâce au système intentions de Consul. Ce système permet d'interdire de base tous les inter-connections de nos services dans le cluster, cette approche de *white listing* empêchera à un acteur ayant accès au cluster de créer des pods potentiellement dangereux et de les connecter à nos autres services.

La seconde des choses à faire dans le cadre de la sécurisation du réseau est de mettre en place un certificat TLS aussi bien pour sécuriser le traffic venant des utilisateurs mais aussi pour sécuriser le traffic à l'intérieur de notre architecture, car si un acteur malveillant parvient à se placer dans notre architecture il pourrait intercepter des données sensibles, ou ce que l'on appelle plus communément une attaque *man in the middle*.

Cependant pour assurer une meilleure sécurité nous pouvons implémenter un mTLS (multiple TLS), CloudFlare définit le mTLS comme suit dans un article [\[cloudflare\]](#):

Mutual TLS, ou mTLS en abrégé, est une méthode d'authentification mutuelle. mTLS garantit que les parties à chaque extrémité d'une connexion réseau sont bien celles qu'elles prétendent être en vérifiant qu'elles possèdent toutes deux la bonne clé privée. Les informations contenues dans leurs certificats TLS respectifs fournissent une vérification supplémentaire. mTLS est souvent utilisé dans un cadre de sécurité Zero Trust pour vérifier les utilisateurs, les appareils et les serveurs au sein d'une organisation. Il peut également contribuer à sécuriser les API.

— Cloudflare, Qu'est-ce que le TLS mutuel (mTLS) ?

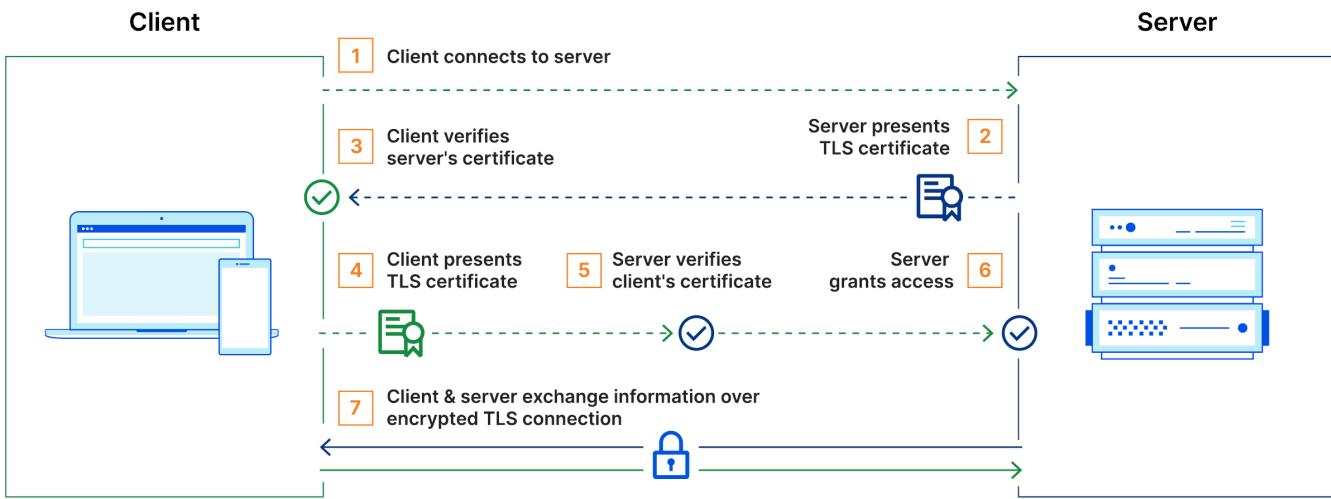


Figure 28. Schématisation de mTLS

Selon ce même article [cloudflare], le mTLS répond à 5 types d'attaque:

- Les attaques *man in the middle*.
- Les usurpations d'identité.
- Les attaques par brute force, qui nécessiteront encore plus d'essais.
- Le phishing, un attaquant à besoin d'une clé privée et d'un certificat.
- Requêtes venant d'API malveillantes.

Notre service mesh HashiCorp Consul met en place un mTLS et gère les certificats TLS pour nous, il suffit de définir les intentions de communication entre nos services et consul se chargera de tout selon un article de la documentation HashiCorp [article_consul].

En plus de chiffrer nos données envoyées sur le réseau, il faut se charger de vérifier l'intégrité des données si jamais la protection par mTLS est compromise. Pour cela nous allons utiliser des JSON Web Tokens (JWT) qui sont des tokens contenant les données envoyées ainsi qu'une signature permettant de vérifier l'intégrité des données qui consiste en un hachage des données et d'une valeur secrète.

Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmb28i0iJiYXIiLCJsb3JlbSI6MTUxNjIzOTAyMn0.FYkmRZLmM3NOZ_ahoKnk4RN12QRFmjEiLcfI46DWP84
```

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

PAYOUT: DATA

```
{  
  "foo": "bar",  
  "lorem": 1516239022  
}
```

VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  your-256-bit-secret  
)  secret base64 encoded
```

✓ Signature Verified

SHARE JWT

Même si un acteur malveillant venait à modifier les données de nos requêtes (en rouge et rose ci dessus) et à les envoyer, la signature (en bleu) ne serait pas correcte car l'acteur ne connaît pas la valeur secrète, nécessaire au calcul de la signature.

Une partie ces protections ont été implémentées dans un POC que vous pourrez retrouver sur GitHub [\[poc_q9\]](#).

Gestion des secrets

Dans notre applications nous aurons de nombreuses variables que nous souhaiterons garder secrètes, comme par exemple les secrets JWT ou encore les identifiants pour nos bases de données. Kubernetes fournit de base des objets Secrets dans son API pour stocker ces valeurs et les sécuriser, pourtant il s'agit de la pire des solutions car les valeurs sont simplement encodés en base64 accessible par n'importe qui ayant un accès au cluster, signifiant que quiconque de mal intentionné s'ayant procuré des accès au cluster aura accès à ces valeurs. Nous allons donc utiliser un gestionnaire de secrets, permettant de chiffrer les secrets et de restreindre l'accès à ces données.

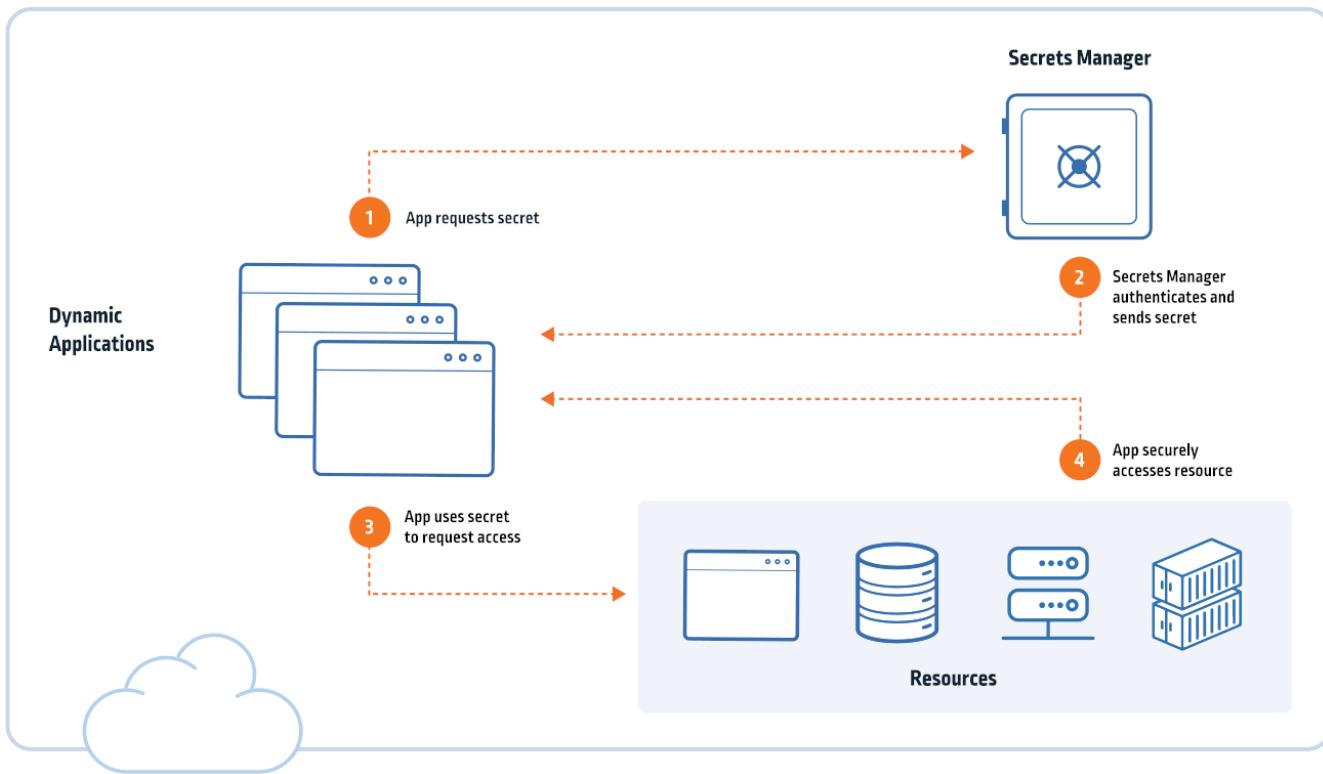


Figure 29. Fonctionnement global d'un gestionnaire de secrets

Lors de recherches sur le sujet les trois noms qui sont le plus ressortis sont Hashicorp Vault, Google Cloud KMS et AWS Secrets Manager. Cependant nous allons choisir Vault car il s'agit de la solution offrant le plus de fonctionnalités intéressantes possibles:

- Vault est doté d'un moteur PKI permettant alors de générer des certificats X.509 à la demande, ce qui peut être utilisé dans les cas où nous avons besoins de systèmes de clés publiques/privés. Nous pouvons prendre l'exemple du mTLS où Vault peut servir d'autorité de certificats (Ce qui est totalement possible comme le montre un tutoriel sur leur site [\[vault_ca\]](#)).
- Il permet de générer des secrets dynamiques. C'est à dire que Vault va, à intervalles réguliers, changer certains secrets dont il a la charge tel que les identifiants des bases de données (changements qu'il va d'ailleurs faire appliquer aux SGBD), ce mécanisme sert à contrer les attaques par brute force et à réduire les fenêtres d'attaques car on peut changer les identifiants très fréquemment
- Vault se base sur un système de rôles et d'actions, qui refuse tout accès par défaut. Cet aspect est important quand on sait que les contrôles d'accès défaillants étaient le type de faille présentant le plus de risque dans les systèmes d'information selon le top 10 de l'OWASP en 2021 [\[owasp10\]](#).

Ce sont les raisons pour lesquelles nous utiliserons vault pour sécuriser nos secrets, car les applications auront un accès à des secrets éphémères et régis par des rôles.

Pour ce faire nous utiliseront des *Sidecar vaults*, ce sont des conteneurs placés dans les pods de nos services qui se chargeront de récupérer les secrets auprès de Vault et de les injecter dans les conteneurs du pod.

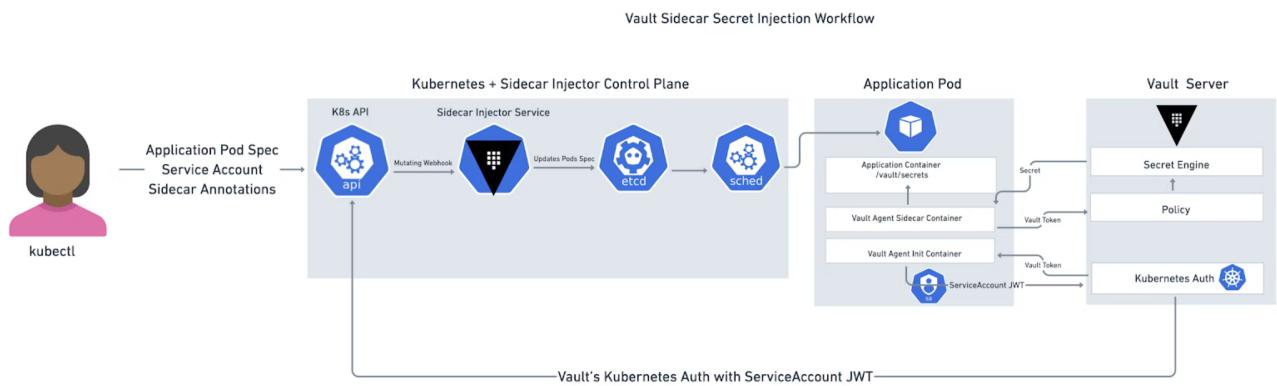


Figure 30. Schématisation de l'injection de secrets

Protection contre les attaques DDoS

Une attaque par déni de service ou attaque DDoS (Distributed Denial of Service) est une cyberattaque qui repose sur un principe simple : Empêcher l'accès à un serveur en le saturant avec des flots de requêtes. Dans un des articles de Cloudflare prévoquant le sujet [\[cloudflare_dns\]](#) on peut retrouver cette analogie

En généralisant, une attaque DDoS ressemble à un embouteillage inattendu qui bloque une autoroute et empêche le trafic normal d'arriver à destination.

— CloudFlare, Qu'est-ce qu'une attaque DDoS ?

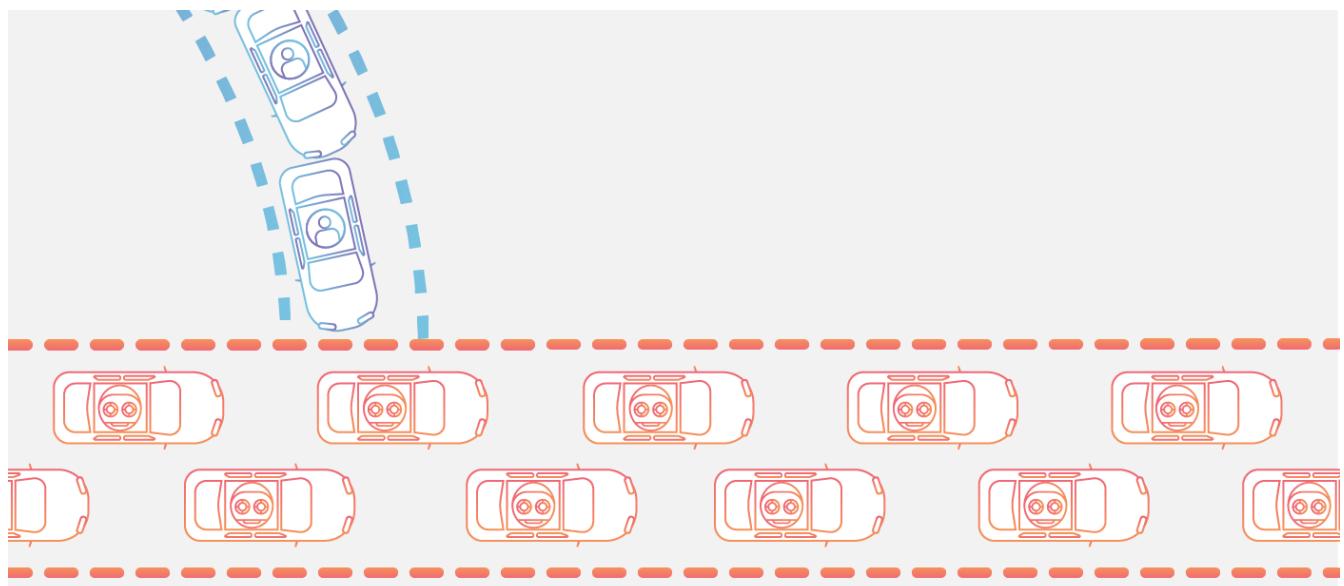


Figure 31. Illustration tirée de l'article de CloudFlare sur le DDoS

Ce type d'attaque à beau sembler primitif, il existe différents types d'attaques DDoS qui exploitent différentes couches du modèle OSI ce qui les rend redoutables et implique la mise en place de quelques dispositifs.

- Les pare feu applicatif (WAF), est une porte d'entrée vers notre application qui va se charger d'évaluer les requêtes entrentes et de les rejeter si elles représentent un danger quelconque

comme une attaque DDoS ou une injection.

- La limitation du taux, il est possible de limiter le nombre de requêtes en parallèle, ce qui permet d'empêcher notre application de s'écrouler sous une avalanche de requêtes.

Une autre technique pour protéger son application d'attaques DDoS est la technique du trou noir, en cas d'attaque DDoS, tout le trafic peut être redirigé vers une autre destination. Nous n'allons pas garder cette solution car au-delà de déplacer le problème autre part, l'attaque reste réussie car tous les clients illégitimes ou non n'ont pas accès à l'application.

Pour appliquer ces techniques nous allons dans un premier cas, utiliser la suite de CloudFlare qui nous permettra d'appliquer un WAF devant l'entrée de notre cluster. Pour ce qui est de la limitation de taux elle peut être configurée dans Consul.

Q10: Comment intégrer les applications UI?

Comme mentionné précédemment, Polycode intègre une sous application dédiée à tester des utilisateurs au travers de campagnes de test. Afin que cette partie ainsi que les prochaines soient bien intégrée avec le reste de l'application, nous allons les développer en micro front-end.

Le site micro-frontends.org définit le micro front-end comme suit [\[micro-frontends\]](#).

L'idée derrière les micro frontends est de considérer un site web ou une application web comme une composition de fonctionnalités appartenant à des équipes indépendantes. Chaque équipe a un domaine d'activité distinct ou une mission attribuée dans laquelle elle est spécialisée. Une équipe est transversale et développe ses fonctionnalités de bout en bout, de la base de données à l'interface utilisateur.

— Micro-frontends.org, What is a micro front-end ?

Le but de cette architecture est de pouvoir développer et déployer les différentes parties de l'application indépendamment les unes des autres, contenant chacune leurs propres composants pouvant être directement récupérés et inclus par d'autres services grâce au *Server Side Include* (SSI).

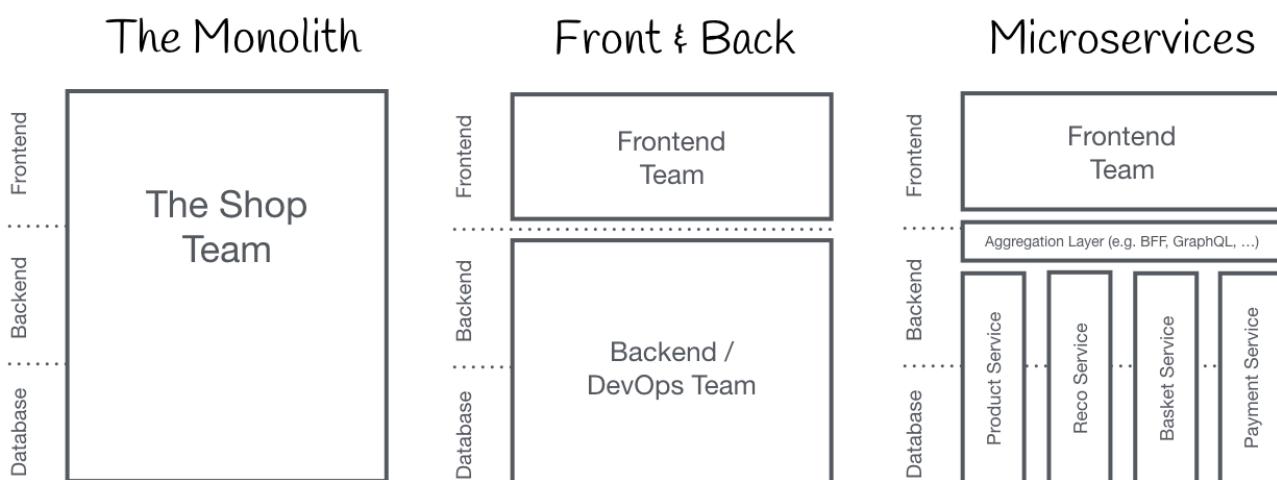


Figure 32. Architecture classique

End-to-End Teams with Micro Frontends

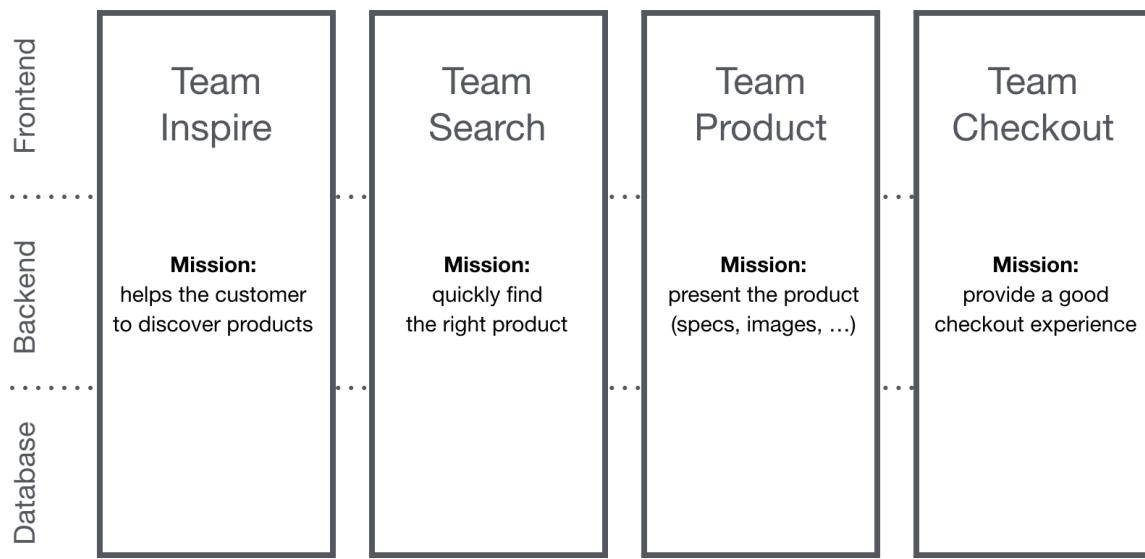


Figure 33. Architecture en micro front-end

Pour pouvoir servir nos composants nous pourrons utiliser NextJS qui permet de servir nos composants React en Server Side Rendering ce qui permettra à l'utilisateur d'accéder rapidement à nos composants si il souffre d'une mauvaise connexion internet.

Par manque de temps et nous n'avons pas eu le temps de développer un proof of concept complet [\[poc10\]](#) pour cette question, veuillez nous excuser pour ce manque.

Bibliographie/Webographie/Liens

- [poc2] <https://gitlab.polytech.umontpellier.fr/yann.pomie/poc2>
- [poc_q3_lp] <https://gitlab.polytech.umontpellier.fr/yann.pomie/poc3-long-polling>
- [poc_q3_ws] <https://gitlab.polytech.umontpellier.fr/yann.pomie/poc3-ws>
- [poc_q3_http] <https://gitlab.polytech.umontpellier.fr/yann.pomie/poc3-http-async>
- [rfc6202] <https://www.rfc-editor.org/rfc/rfc6202#section-2.2>
- [api_swagger] https://woodenmaiden.github.io/rapport_polycode_swagger
- [cloudflare] <https://www.cloudflare.com/fr-fr/learning/access-management/what-is-mutual-tls/>
- [lambdo] <https://github.com/virt-do/lambdo>
- [ESDB] <https://www.eventstore.com/>
- [poc_q9] https://github.com/WoodenMaiden/poc9_polycode
- [vault_ca] <https://developer.hashicorp.com/vault/tutorials/secrets-management/pki-engine>
- [owasp10] <https://owasp.org/Top10/fr/>
- [cloudflare_dns] <https://www.cloudflare.com/fr-fr/learning/ddos/what-is-a-ddos-attack/>
- [article_consul] <https://developer.hashicorp.com/consul/docs/connect/connect-internals>
- [poc10] <https://gitlab.polytech.umontpellier.fr/yann.pomie/poc10>