ADR Choix d’architecture pour le Projet Hearthstone Battleground

Titre : Utiliser une architecture microservices

Date : 12/14/2023

Statut : Accepté

Contexte : On cherche à développer Hearthstone: Champs de bataille qui est un jeu stratégique à 8 joueurs dans l'univers de Hearthstone. Les joueurs recrutent des serviteurs, composent un plateau évolutif, et s'affrontent automatiquement dans des duels pour être le dernier héros en vie.

Décision : Nous avons décidé d'utiliser l’architecture microservices.

Conséquences :

- Positives :
    - Les microservices peuvent être mis à l'échelle individuellement en fonction de leurs besoins spécifiques en ressources. Si certaines parties du système, comme le service de catalogue, connaissent une demande accrue, vous pouvez mettre à l'échelle uniquement ces microservices sans affecter l'ensemble de l'application.
    - Une architecture de microservices permet de diviser le système en petits services indépendants. Chaque microservice, tel que le service de catalogue, peut être développé, déployé et mis à l'échelle indépendamment. Cette modularité facilite la compréhension, la maintenance et l'évolution de chaque composant.
    - Chaque service peut être développé, testé et déployé indépendamment des autres, ce qui accélère notre cycle de développement.
    - Chaque service a son propre référentiel de données, évitant ainsi les conflits potentiels et permettant une meilleure gestion des données.

    - Avec des services indépendants, nous pouvons apporter des mises à jour à un service sans perturber l'ensemble du système. Cela améliore la stabilité de notre application et facilite la gestion des modifications.

- Négatives :
    - Les microservices introduisent une complexité accrue en raison de la nécessité d'une communication réseau entre les services. Comprendre et gérer les interactions entre les microservices peut être plus difficile, entraînant des problèmes potentiels de latence réseau, de défaillances de communication et la nécessité d'une gestion robuste des erreurs.
    - La gestion et le déploiement de plusieurs microservices nécessitent des efforts opérationnels supplémentaires. Chaque microservice doit être déployé, surveillé et mis à l'échelle individuellement, ce qui ajoute une complexité opérationnelle. Les pratiques DevOps deviennent cruciales pour garantir la fiabilité et la disponibilité de l'ensemble du système.
    - La coordination de la cohérence des données sur plusieurs bases de données dans une architecture de microservices peut être complexe. La mise en œuvre de transactions qui englobent plusieurs services est difficile, et le maintien de l'intégrité des données nécessite une conception soigneuse. Une synchronisation adéquate des données et une communication entre les microservices sont essentielles pour relever ces défis.

Références :

Titre : Utilisation de Node.js avec Express.js pour le service jeu

Date : 12/14/2023

Statut : Accepté

Contexte : On cherche à développer Hearthstone: Champs de bataille qui est un jeu stratégique à 8 joueurs dans l'univers de Hearthstone. Les joueurs recrutent des serviteurs, composent un plateau évolutif, et s'affrontent automatiquement dans des duels pour être le dernier héros en vie.

Décision : Nous avons décidé d'utiliser Node.js avec Express.js pour le développement des fonctionnalités Backend de notre jeu.

Conséquences :

- Positives :
  - Node.js est conçu pour être non bloquant et asynchrone, ce qui le rend bien adapté pour gérer un grand nombre de connexions simultanées. Dans un service de jeu, où la communication en temps réel et la réactivité sont cruciales, la nature événementielle et non bloquante de Node.js peut être avantageuse.
  - Node.js est souvent privilégié pour les applications nécessitant des fonctionnalités en temps réel, telles que les jeux multijoueurs. La capacité à gérer les connexions WebSocket et à transmettre des données en temps réel peut constituer un avantage significatif dans un scénario de jeu.
  - Node.js et Express.js sont réputés pour leur simplicité et leur légèreté. Le processus de développement peut être rapide, et le framework fournit juste assez de structure pour construire des applications évolutives sans introduire de complexité inutile.
  - L'écosystème Node.js dispose d'une vaste collection de modules (paquets npm) qui peuvent être facilement intégrés dans une application Express.js. Cela peut accélérer le développement en tirant parti de solutions existantes pour diverses fonctionnalités.
  - Node.js bénéficie d'une communauté nombreuse et active, et de nombreuses ressources d'apprentissage et tutoriels sont disponibles. Si votre équipe de développement est déjà familière avec JavaScript et Node.js, cela pourrait contribuer à un processus de développement plus fluide.
- Négatives :
  - Node.js est monoprocessus et basé sur des événements, ce qui pourrait limiter ses performances pour les tâches intensives en CPU. Les services de jeu impliquent souvent des calculs et des simulations, et la nature monoprocessus de Node.js peut ne pas exploiter pleinement les processeurs multi-cœurs.
  - Le modèle de programmation asynchrone de Node.js repose fortement sur les rappels (callbacks), ce qui peut conduire à ce que l'on appelle "l'enfer des rappels" (callback hell) - une situation où le code devient profondément

imbriqué et plus difficile à lire. Gérer correctement le code asynchrone nécessite une bonne compréhension des motifs de rappels, des Promesses (Promises), ou de async/await, ce qui peut représenter une courbe d'apprentissage pour les développeurs.

- Bien que Node.js excelle dans la gestion de nombreuses connexions simultanées, il pourrait ne pas exploiter pleinement la puissance des machines multi-cœurs. Cette limitation pourrait avoir un impact sur la scalabilité d'un service de jeu nécessitant une distribution efficace de charges de travail computationnelles importantes sur plusieurs cœurs.

Références :

Documentation officielle de node.js: https://docs.spring.io/spring-framework/reference/

Site officiel de express.js: https://expressjs.com/

Titre : Utilisation de Java avec Spring Boot pour le service gestion des utilisateurs

Date : 12/14/2023

Statut : Accepté

Contexte : Nous devons gèrer les utilisateur

Décision : Nous avons décidé d'utiliser Java avec Spring Boot pour le développement des fonctionnalités du service gestion de comptes.

Conséquences :

- Positives :
  - Spring Security, faisant partie de l'écosystème Spring, est un framework de sécurité puissant et personnalisable. Il peut être intégré de manière transparente dans les applications Spring Boot pour gérer l'authentification, l'autorisation et d'autres aspects liés à la sécurité. Cela est crucial pour un service de gestion des utilisateurs, qui implique généralement la gestion de l'authentification des utilisateurs et du contrôle d'accès.
  - Java est un langage de programmation mature et largement adopté dans le monde de l'entreprise. Spring Boot s'appuie sur l'écosystème Java et offre une manière rationalisée de construire des applications robustes et évolutives. La

stabilité et la maturité de Java et de Spring Boot en font un choix fiable pour les applications de niveau entreprise.

    - Spring Boot est bien adapté à la construction de microservices. Il offre des fonctionnalités telles que des serveurs intégrés, une configuration facile et la prise en charge de la création d'API RESTful. Ces fonctionnalités correspondent bien aux exigences de l'architecture des microservices, en en faisant un choix naturel pour un service de gestion des utilisateurs dans une application basée sur des microservices.

- Négatives :
    - Pour les développeurs qui sont nouveaux dans le domaine de Java ou de Spring Boot, il peut y avoir une courbe d'apprentissage associée à la compréhension du framework et de ses conventions. Bien que Spring Boot vise à simplifier le développement, ses fonctionnalités étendues peuvent prendre du temps à maîtriser.
    - Les applications Java, y compris celles construites avec Spring Boot, peuvent consommer plus de mémoire par rapport aux applications écrites dans certains autres langages. Cela peut être une considération dans des environnements limités en ressources, surtout lors de l'exécution d'un grand nombre de microservices.

Références :

Documentation officielle de Java: https://docs.oracle.com/en/java/

Documentation officielle de Spring Boot: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/