Rapport Projet : Architecture des composants d'entreprise

Application CLUNT

1. Introduction

Aperçu du projet

Le projet vise à concevoir et mettre en œuvre une architecture des composants d'entreprise basée sur le modèle de microservices. Cette approche permettra une gestion plus flexible et évolutive des services au sein de l'entreprise.
clunt est une application simple de conseiller financier conçue pour démontrer le modèle d'architecture de microservices à l'aide de Spring Boot, Spring Cloud et Docker. Le projet est conçu comme un tutoriel, mais vous pouvez le créer et le transformer en autre chose !

Importance de l'architecture microservices

L'adoption de l'architecture microservices est cruciale pour favoriser la modularité, la scalabilité et la résilience des applications. Elle permet également une plus grande agilité dans le développement et la maintenance des systèmes.

2. Architecture Microservices

Architecture

L'architecture microservices consiste en un ensemble de services indépendants, déployés de manière isolée, et communiquant entre eux par des interfaces bien définies. Cela favorise la résilience et la facilité de mise à l'échelle.

Description des services

Chaque service est dédié à une fonctionnalité spécifique de l'entreprise, assurant ainsi la séparation des préoccupations. Les services sont autonomes, ce qui simplifie le déploiement, la maintenance et les mises à jour.

Account service

Contient la logique générale de saisie et la validation : éléments de revenus/dépenses, paramètres d'épargne et de compte

.

Method	Path	Description	User authenticated	Available from UI
GET	/accounts/{account}	Get specified account data		

Method	Path	Description	User authenticated	Available from UI
GET	/accounts/current	Get current account data	×	×
GET	/accounts/demo	Get demo account data (pre-filled incomes/expenses items, etc)		×
PUT	/accounts/current	Save current account data	×	×
POST	/accounts/	Register new account		×

Statistics service

Effectue des calculs sur les principaux paramètres statistiques et capture des séries chronologiques pour chaque compte. Le point de données contient des valeurs normalisées par rapport à la devise de base et à la période. Ces données sont utilisées pour suivre la dynamique des flux de trésorerie pendant la durée de vie du compte.

Method	Path	Description	User authenticated	Available from UI
GET	/statistics/{account}	Get specified account statistics		
GET	/statistics/current	Get current account statistics	×	×

Method	Path	Description	User authenticated	Available from UI
GET	/statistics/demo	Get demo account statistics		×
PUT	/statistics/{account}	Create or update time series datapoint for specified account		

Notification service

Stocke les informations de contact de l'utilisateur et les paramètres de notification (rappels, fréquence de sauvegarde, etc.). Le travailleur planifié collecte les informations requises auprès d'autres services et envoie des messages électroniques aux clients abonnés.

Method	Path	Description	User authenticated	Available from UI
GET	/notifications/settings/current	Get current account notification settings	×	×
PUT	/notifications/settings/current	Save current account notification settings	×	×

Notes

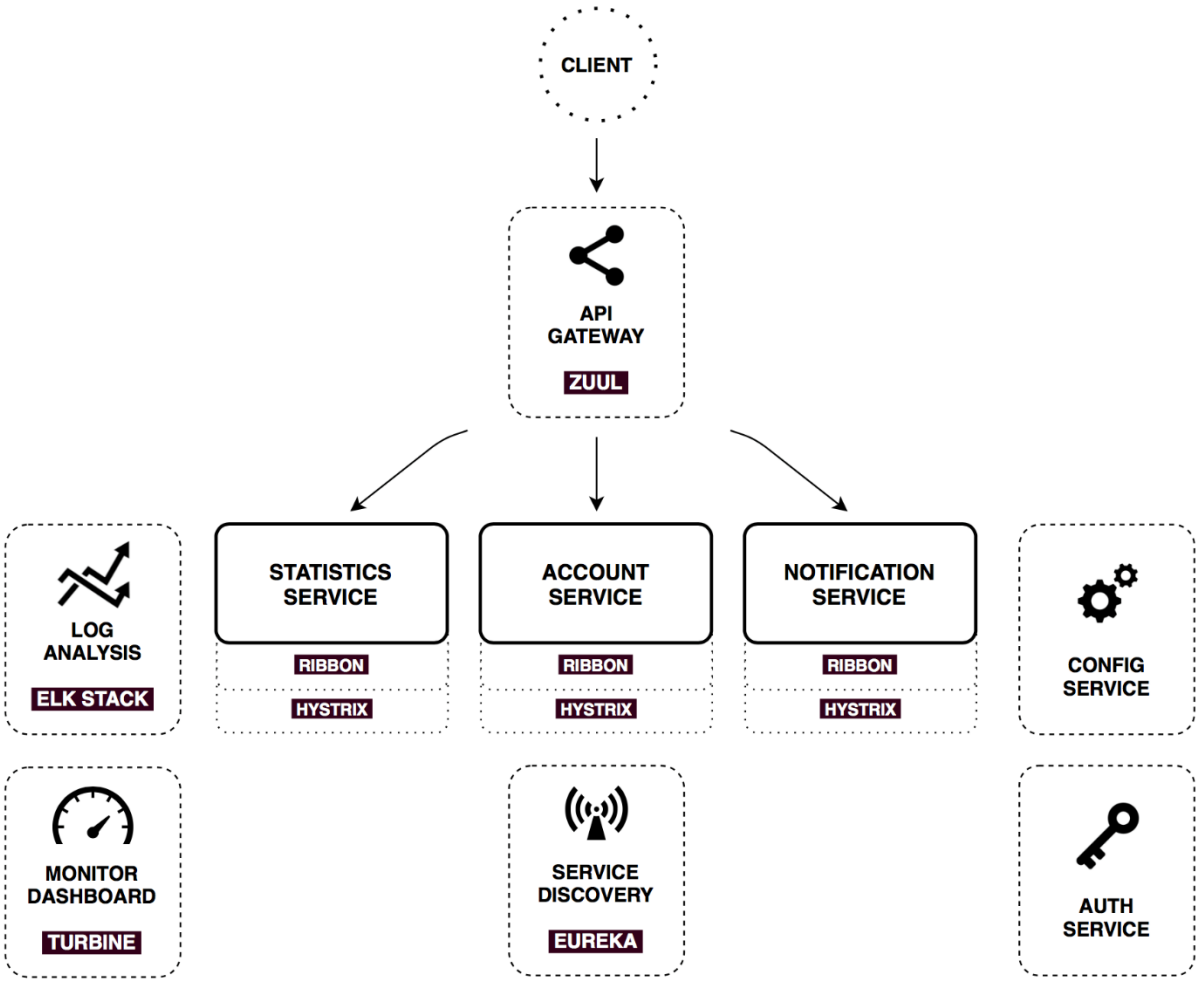
• Chaque microservice possède sa propre base de données, il n'y a donc aucun moyen de contourner l'API et d'accéder directement aux données de persistance.

• MongoDB est utilisé comme base de données principale pour chacun des services.

• Tous les services communiquent entre eux via l'API Rest

Infrastructure

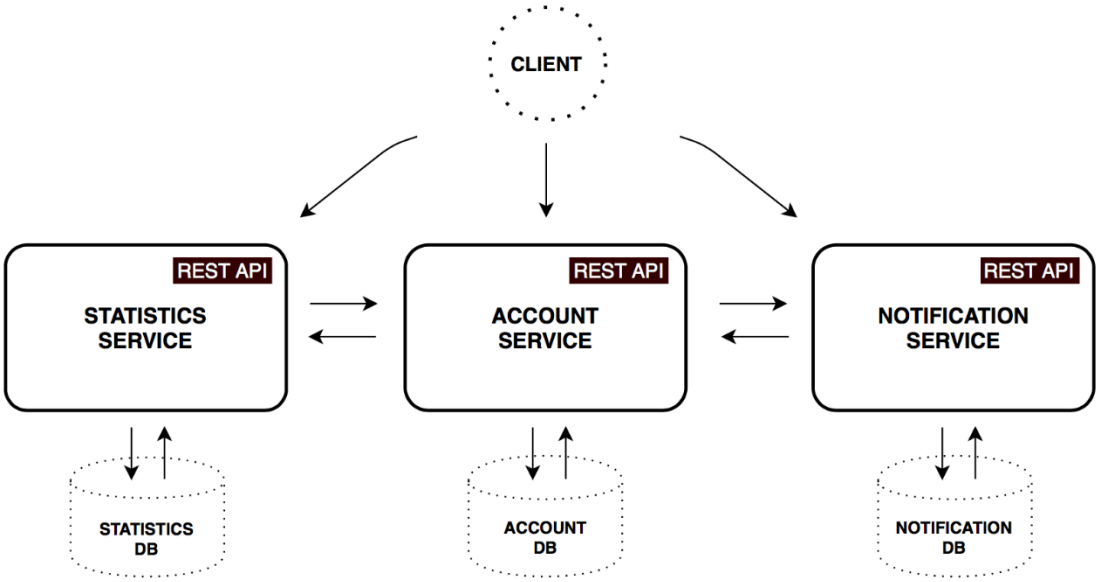
Spring cloud provides powerful tools for developers to quickly implement common distributed systems patterns
-



Config service

Mécanismes de communication

La communication entre les microservices s'effectue généralement via des API REST ou des mécanismes de messagerie, assurant une interaction efficace tout en minimisant les dépendances.



3. Conception des Microservices

Approche de conception pour chaque service

Chaque service est conçu en suivant les principes SOLID et en utilisant les technologies les plus adaptées à sa fonction. L'approche "Design Thinking" est utilisée pour garantir une conception centrée sur l'utilisateur.

Utilisation côté client

Il suffit de construire une application Spring Boot avec la dépendance spring-cloud-starter-config, la configuration automatique fera le reste.

Maintenant, vous n'avez plus besoin de propriétés intégrées dans votre application. Fournissez simplement un bootstrap.yml avec le nom de l'application et l'URL du service de configuration :

yamlCopy code

spring: application: name: notification-service cloud: config: uri: http://config:8888 fail-fast: true

Avec Spring Cloud Config, vous pouvez changer la configuration de l'application dynamiquement. Par exemple, le bean EmailService est annoté avec @RefreshScope. Cela signifie que vous pouvez changer le texte et le sujet de l'e-mail sans reconstruire et redémarrer le service de notification.

Pour ce faire, modifiez les propriétés requises dans le serveur de configuration. Ensuite, effectuez un appel de rafraîchissement au service de notification :

bashCopy code

curl -H "Authorization: Bearer #token#" -XPOST http://127.0.0.1:8000/notifications/refresh

Vous pouvez également utiliser des webhooks de référentiel pour automatiser ce processus.

Remarques :

- @RefreshScope ne fonctionne pas avec les classes @Configuration et n'ignore pas les méthodes @Scheduled.
- La propriété fail-fast signifie que l'application Spring Boot échouera immédiatement au démarrage si elle ne peut pas se connecter au service de configuration.

2. Service d'Authentification

Les responsabilités d'authentification sont extraites vers un serveur distinct, qui accorde des jetons OAuth2 pour les services de ressources en arrière-plan. Le serveur d'authentification est utilisé pour l'autorisation des utilisateurs ainsi que pour la communication sécurisée de machine à machine à l'intérieur du périmètre.

Dans ce projet, j'utilise le type d'autorisation de mot de passe pour l'autorisation des utilisateurs (car il est utilisé uniquement par l'interface utilisateur) et le type d'autorisation de client pour la communication inter-services.

Spring Cloud Security fournit des annotations pratiques et une autoconfiguration pour faciliter la mise en œuvre à la fois du côté serveur et du côté client. Vous pouvez en savoir plus dans la documentation.

Du côté client, tout fonctionne exactement de la même manière qu'avec une autorisation basée sur une session traditionnelle. Vous pouvez récupérer l'objet Principal de la requête, vérifier les rôles de l'utilisateur en utilisant le contrôle d'accès basé sur des expressions et l'annotation @PreAuthorize.

Chaque client de PiggyMetrics a une portée : server pour les services en arrière-plan et ui - pour le navigateur. Nous pouvons utiliser l'annotation @PreAuthorize pour protéger les contrôleurs contre un accès externe :

javaCopy code

@PreAuthorize("#oauth2.hasScope('server')") @RequestMapping(value = "accounts/{name}", method = RequestMethod.GET) public List<DataPoint> getStatisticsByAccountName(@PathVariable String name) { return statisticsService.findByAccountName(name); }

3. Passerelle API (API Gateway)

La passerelle API (API Gateway) est un point d'entrée unique dans le système, utilisé pour gérer les requêtes et les router vers le service en arrière-plan approprié ou en agrégeant les résultats d'un appel scatter-gather. Elle peut également être utilisée pour l'authentification, les insights, les tests de stress et de canari, la migration de service, la gestion des réponses statiques et la gestion active du trafic.

Netflix a open-sourcé un tel service d'entrée et Spring Cloud permet de l'utiliser avec une simple annotation @EnableZuulProxy. Dans ce projet, nous utilisons Zuul pour stocker certains contenus statiques (l'application UI) et pour router les requêtes vers les microservices appropriés. Voici une configuration de routage basée sur le préfixe pour le service de notification :

yamlCopy code

zuul: routes: notification-service: path: /notifications/** serviceId: notification-service stripPrefix: false

Cela signifie que toutes les demandes commençant par /notifications seront routées vers le service de notification. Il n'y a pas d'adresses codées en dur, comme vous pouvez le voir. Zuul utilise le

mécanisme de découverte de service pour localiser les instances du service de notification et également le disjoncteur et l'équilibrage de charge, décrits ci-dessous.

4. Découverte de Service

La découverte de service permet la détection automatique des emplacements réseau de tous les services enregistrés. Ces emplacements peuvent avoir des adresses assignées dynamiquement en raison de la mise à l'échelle automatique, des pannes ou des mises à niveau.

La partie clé de la découverte de service est le Registre. Dans ce projet, nous utilisons Netflix Eureka. Eureka est un bon exemple du modèle de découverte côté client, où le client est responsable de rechercher les emplacements des instances de service disponibles et de faire l'équilibrage entre elles.

Avec Spring Boot, vous pouvez facilement construire un registre Eureka en utilisant la dépendance spring-cloud-starter-eureka-server, l'annotation @EnableEurekaServer et des propriétés de configuration simples.

Le support client est activé avec l'annotation @EnableDiscoveryClient et un bootstrap.yml avec le nom de l'application :

yamlCopy code

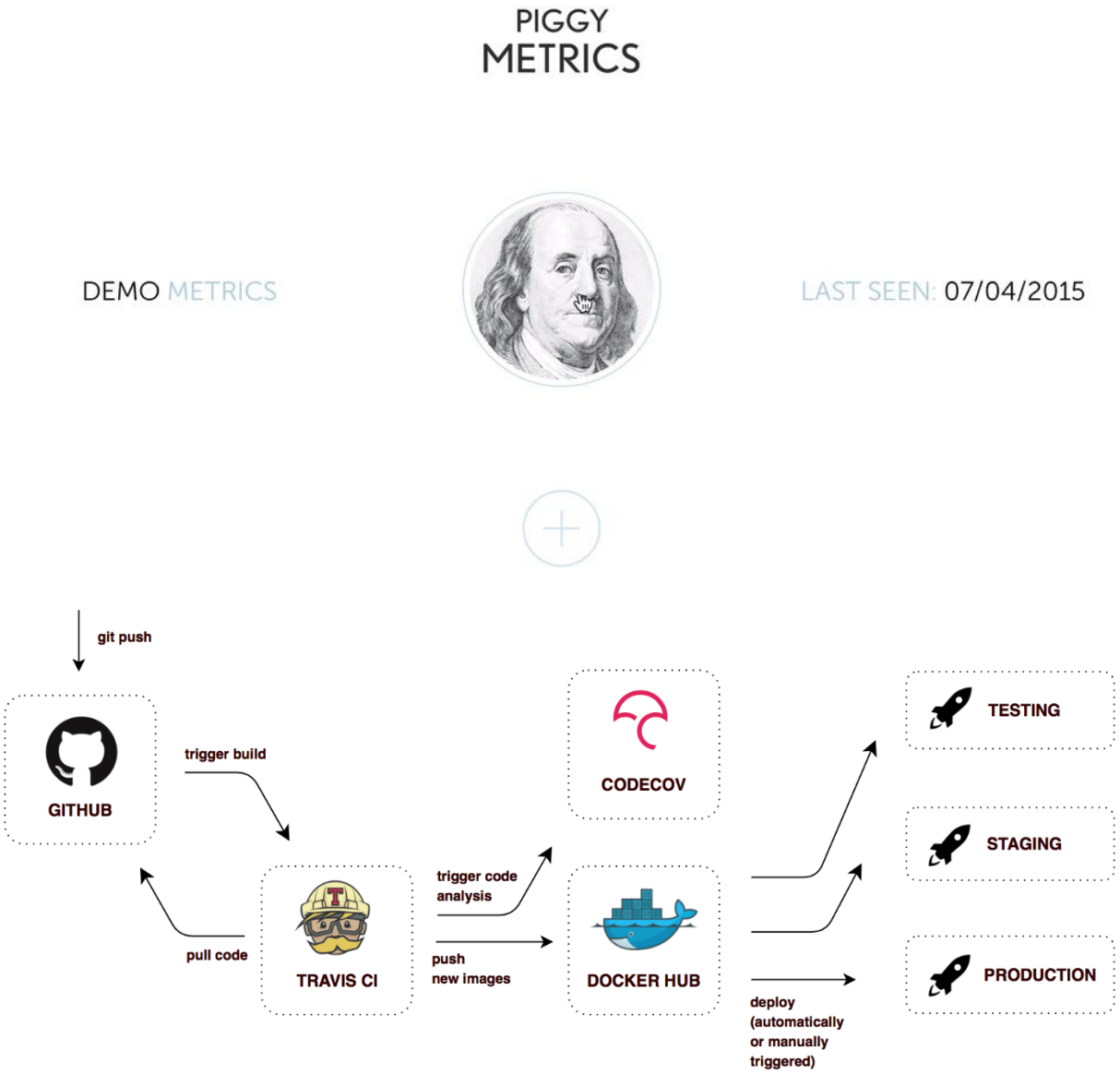
spring: application: name: notification-service

Ce service sera enregistré auprès du serveur Eureka et fourni avec des métadonnées telles que l'hôte, le port, l'URL de l'indicateur de santé, la page d'accueil, etc. Eureka reçoit des messages de battement de cœur de chaque instance appartenant au service. Si le battement de cœur échoue sur un calendrier configurable, l'instance sera supprimée du regist

4. Conteneurisation avec Docker

Implémentation et avantages

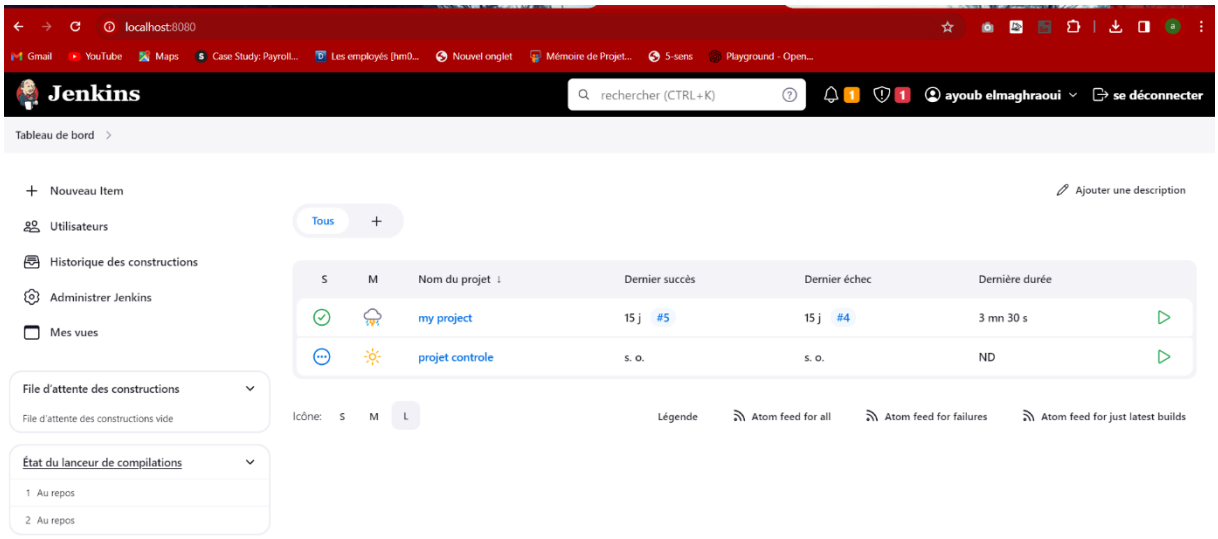
Les microservices sont conteneurisés à l'aide de Docker pour assurer une portabilité et une isolation optimales. Cela facilite le déploiement sur différents environnements et simplifie la gestion des dépendances.

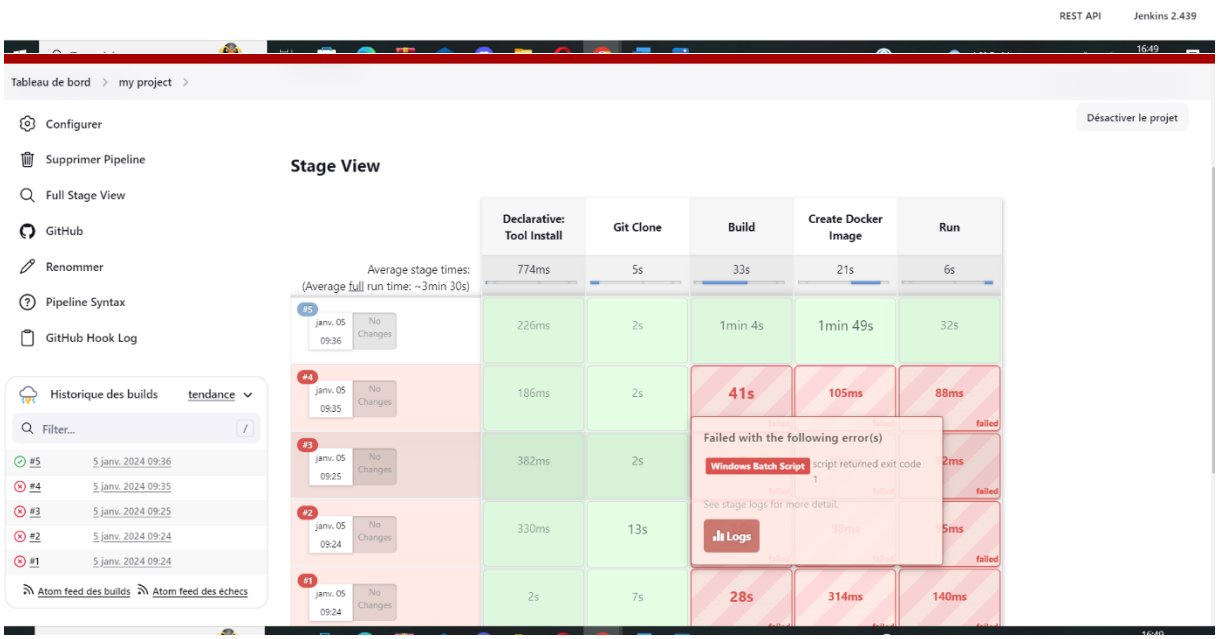


5. CI/CD avec Jenkins

Processus et configuration

L'intégration continue (CI) et le déploiement continu (CD) sont mis en œuvre avec Jenkins pour automatiser les tests, la construction et le déploiement des microservices. Cela garantit une livraison rapide et fiable.





6. Déploiement Automatique

Utilisation de Ngrok ou Azure Cloud

Le déploiement automatique s'effectue en utilisant des outils tels que Ngrok pour des déploiements rapides en environnement de test, et Azure Cloud pour des déploiements plus stables et sécurisés en production.

7. Intégration de SonarQube

Configuration et bénéfices pour la qualité du code

SonarQube est intégré pour analyser et mesurer la qualité du code des microservices. Cela permet d'identifier les problèmes de code, d'améliorer la lisibilité et de garantir une meilleure maintenabilité.

8. Conclusion

Résumé des accomplissements

L'adoption de l'architecture microservices, la conteneurisation avec Docker, l'automatisation avec Jenkins, le déploiement automatique, et l'intégration de SonarQube ont considérablement amélioré la gestion des composants d'entreprise.

Perspectives futures

Les prochaines étapes pourraient inclure l'exploration de technologies émergentes, l'optimisation continue des processus CI/CD, et l'extension de l'architecture pour répondre aux besoins futurs de l'entreprise.

Important endpoints

- http://localhost:80 - Gateway
- http://localhost:8761 - Eureka Dashboard
- http://localhost:9000/hystrix - Hystrix Dashboard (Turbine stream link: http://turbine-stream-service:8080/turbine/turbine.stream)
- http://localhost:15672 - RabbitMq management (default login/password: guest/guest)