

Rapport projet Cloud

BERREKAM Romaïssa
GRINE Omayma
RODRIGUES Enzo
HATTAB Wassan
BARACHET Marius

Table des matières

Description générale.....	3
Technologies utilisées.....	3
Frontend.....	3
Backend.....	3
Azure.....	3
Terraform.....	4
Guide de démarrage.....	4
Architecture de l'application.....	5
Le Backend.....	5
L'interaction entre Frontend et Backend.....	6
L'interaction d'Azure et Terraform avec l'application.....	7
Configuration.....	7
Application.....	8
Le monitoring dans le projet.....	8
L'interaction de la sécurité avec les autres éléments du projet.....	9
GitHub.....	10
Frontend.....	10
Backend.....	10
Fonctionnalités détaillées.....	11
Backend.....	11
Frontend.....	12
Swagger.....	13
Sécurité.....	14
Backend.....	14
Frontend.....	15
Terraform.....	15
GitHub.....	16
Monitoring.....	16
Conclusion.....	18
12 Facteurs.....	19

Description générale

L'objectif principal de l'application est de créer une architecture frontend-backend qui offre une plateforme d'affichage des films Ghibli.

Cette application est déployée dans le cloud grâce à l'utilisation de Terraform, un outil conçu pour orchestrer et automatiser la gestion de l'infrastructure cloud. Les ressources créées et gérées sont directement déployées dans l'environnement cloud d'Azure, permettant ainsi une infrastructure flexible.

Dans ce projet, Terraform est mis en place en collaboration avec des **Dockerfiles**, utilisés à la fois pour le backend et le frontend. Les Dockerfiles définissent les instructions nécessaires pour construire les images Docker de l'application, contenant tout ce qu'il faut pour exécuter le frontend et le backend (dépendances, configuration, code, etc.).

Terraform récupère ensuite ces images pour déployer et orchestrer les différents composants (Front + Back) de l'application sur l'infrastructure cible.

Technologies utilisées

Frontend

Pour le développement du Frontend nous avons choisi d'utiliser Angular notamment pour sa facilité de mise en place mais aussi pour sa bonne compatibilité avec le Backend grâce à ses capacités de gestion des requêtes HTTP et son système de liaison bidirectionnelle de données facilitant la communication entre ces derniers.

Backend

Nous avons choisi d'utiliser un Backend en JAVA et plus précisément en Spring Boot car c'est un langage que l'on a eu l'occasion de beaucoup utilisé lors de notre parcours universitaire. Ceci nous a permis de travailler avec aisance et efficacité. De plus, la technologie Spring facilite grandement la mise en place d'un Backend grâce à ses fonctionnalités intégrées et notamment la sécurisation de ce dernier (Spring Security).

Azure

Nous avons choisi d'utiliser Azure la plateforme cloud de Microsoft au sein de notre projet, ce choix a été fait car Azure offre une large gamme de services adaptés aux besoins du projet, parmi ces services nous y retrouvons les suivants :

- Azure Container Registry (ACR) : Nous permet de stocker les images Docker du Frontend et du Backend
- Azure Container Instance (ACI) : nous permet de déployer et de gérer les conteneurs Backend et Frontend.
- Azure Application Insights : nous permet de garantir la fiabilité et la performance de notre application en fournissant des données approfondies pour le diagnostic des problèmes, le suivi des utilisateurs et l'optimisation des performances.
- Intégration avec GitHub : Grâce à son partenariat avec GitHub, Azure permet une intégration native pour CI/CD, ce qui simplifie l'automatisation des déploiements.

Terraform

Terraform nous a été indispensable pour automatiser et gérer l'infrastructure Azure grâce à son intégration avec Azure. Terraform nous a permis de décrire toute l'infrastructure dans des fichiers de configuration, ce qui nous a garanti une création et configuration identique des ressources dans tous les environnements.

Terraform nous a été important également par sa collaboration et son versioning avec Git. Étant donné qu'on a deux environnements à déployer. Deux répertoires ont été créés dev et prod tous deux dans le répertoire terraform. Ces répertoires contiennent les fichiers de configuration .tf permettant de déployer l'application en développement ou bien en production. Les consignes de déploiement sont disponibles dans la partie guide de démarrage.

Nous avons fait le choix d'utiliser l'instance de conteneur pour exécuter les conteneurs Backend et Frontend en pensant à le remplacer par l'Azure app service en fin du projet, nous avons fait plusieurs tentatives pour le faire fonctionner mais toutes les tentatives ont échouées, c'est pour cette raison que nous avons gardé le fonctionnement existant à l'aide de l'utilisation de l'instance de conteneur.

Guide de démarrage

URL officielles

Environnement DEV

- Back : equipe5-dev.eastus.azurecontainer.io:8080
- Front : equipe5-dev.eastus.azurecontainer.io:9090

Environnement PROD

- Back : equipe5-prod.eastus.azurecontainer.io:8080
- Front : equipe5-prod.eastus.azurecontainer.io:9090

Lancer l'applicaton avec Docker

Back

- `cd back`
- `docker build -t back-app:groupe5 -f .\Dockerfile .`
- `docker run -p 8080:8080 back-app:groupe5`

Front

- `cd front`
- `docker build -t front-app:groupe5 -f .\Dockerfile .`
- `docker run -p 4200:9090 front-app:groupe5`

Lancer l'application en local

Back

- Ouvrir le dossier back avec un IDE
- Ouvrir la classe BackAppApplication qui se trouve dans `back\src\main\java\com\groupe5\backapp\BackAppApplication.java`
- Run le main

Front

- `cd front`
- décommenter la ligne 14 et commenter la ligne 15 du Dockerfile
- installer nodeJS si vous ne l'avez pas (verification avec `node -v`)
- `npm install -g @angular/cli`
- `npm run start local`
- aller sur <http://localhost:4200/>

Déploiement

Déploiement sur l'environnement de développement

Etape 1 : Push en dev

- `git checkout dev`
- `git push`

- Se rendre sur l'onglet action et le workflow 'deploy to developement' et récupérer la valeur `githubsha` lors de l'étape "Retrieve information"
- Garder cette valeur dans le presse-papier

Etape 2 : Déployer en utilisant terraform

- se placer dans le repertoire terraform `cd terraform/dev`
- `terraform init -upgrade`
- `terraform plan -var="tag=[github sha récupéré plus tôt]"`
- `terraform apply -var="tag=[github sha récupéré plus tôt]"`

Déploiement sur l'environnement de production

Etape 1 : Push en production

- `git checkout main` /\ bien mettre les modifications validé de dev vers main
- `git tag -a vX.X.X -m "Version X.X.X"` /\ Faire attention la norme semver est utilisé
- `git push origin vX.X.X`
- (possibilité de faire une release sur github)
- Bien garder la référence du tag, elle sera utile plus tard

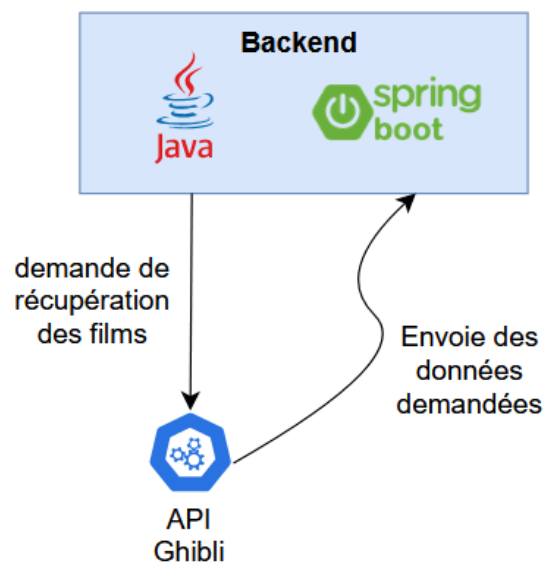
Etape 2 : Déployer en utilisant terraform

- se placer dans le repertoire terraform `cd terraform/prod`
- `terraform init -upgrade`
- `terraform plan -var="tag=[tag récupéré plus tôt]"`
- `terraform apply -var="tag=[tag récupéré plus tôt]"`

Architecture de l'application

Car un schéma est plus parlant qu'un paragraphe, voici les diagrammes d'architectures concernant chaque partie constituant le projet et les explications relatives à sa compréhension.

Le Backend



L'image ci-dessus représente l'architecture du Backend développé en Java avec le framework Spring Boot, qui interagit avec l'API externe Ghibli.

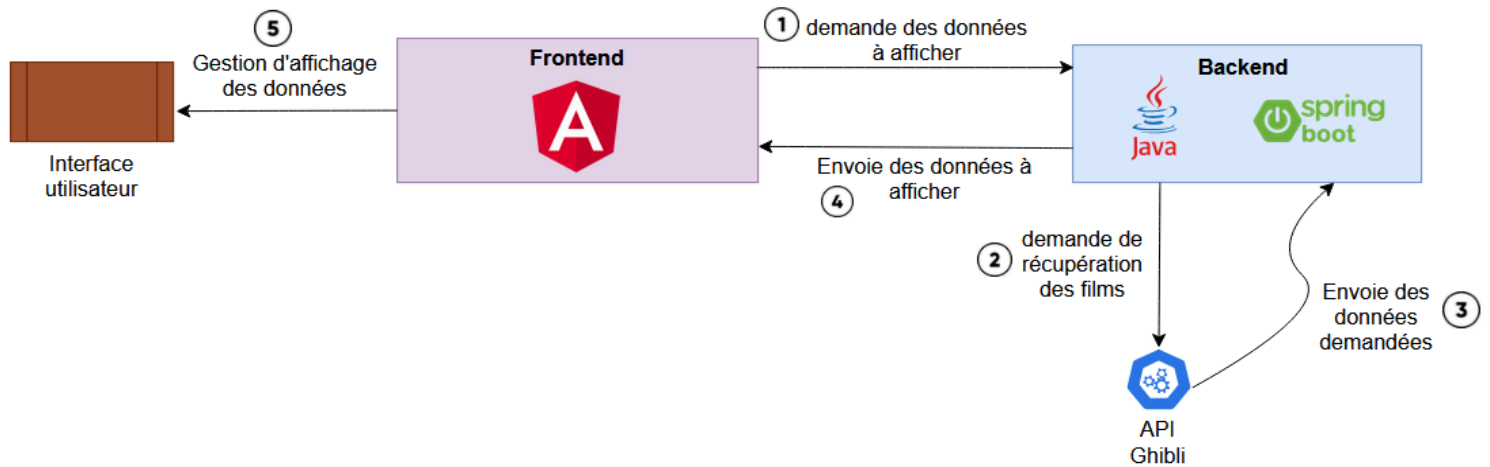
Le Backend envoie une demande à l'API Ghibli pour récupérer la liste des films ou un film en particulier. L'API nous fournira des informations supplémentaires sur les films comme :

- le titre
- le producteur
- l'année de sortie

L'API Ghibli répond ensuite à cette demande en envoyant les données demandées au backend.

Cette communication est illustrée par deux flèches : une flèche partant du backend vers l'API Ghibli, indiquant la "demande de récupération des films", et une flèche de retour de l'API Ghibli vers le backend, indiquant "l'envoi des données demandées".

L'interaction entre Frontend et Backend



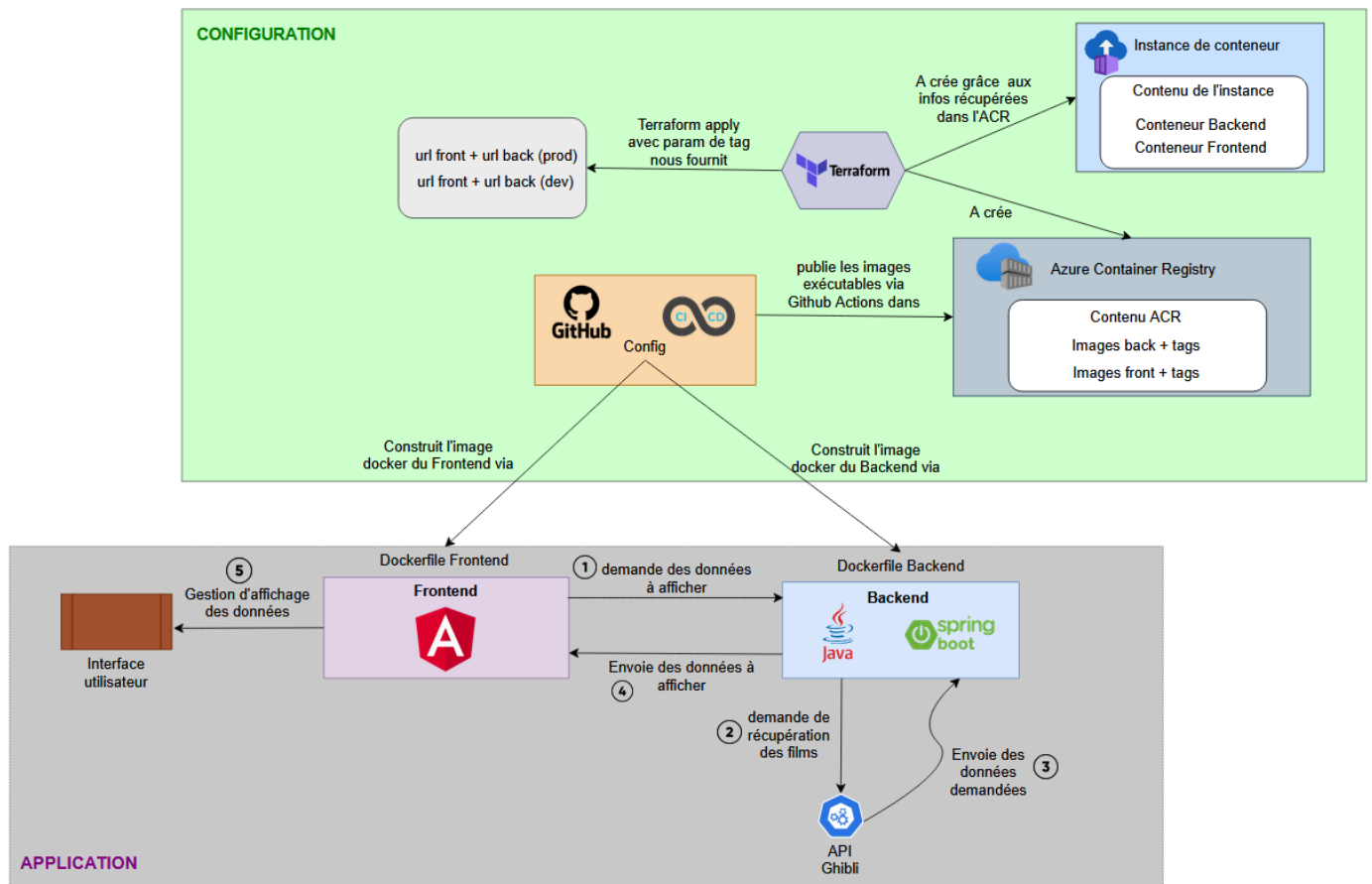
L'image ci-dessus illustre l'architecture du Frontend de l'application et son interaction avec le Backend.

Afin de récupérer les données nécessaires de l'application et les afficher aux utilisateurs, le Frontend développé avec Angular, interagit avec le Backend via une requête HTTP dans laquelle on interroge l'url du Backend pour qu'il interroge par la suite l'API Ghibli et renvoie les informations reçues au Frontend.

Après la récupération de ces données, nous les traitons et les mettons en forme pour l'affichage à l'utilisateur via l'interface utilisateur.

Cette image met en évidence la séparation des responsabilités entre le frontend (gestion de l'interface utilisateur), le backend (logique métier et gestion des données), et l'API externe (fournisseur des données).

L'interaction d'Azure et Terraform avec l'application



L'image ci-dessus représente deux parties importantes du projet, on y trouve la partie Configuration et la partie Application entre lesquelles il y a des interactions pour assurer le bon fonctionnement de l'application, voici quelques explications concernant le contenu de ces deux parties :

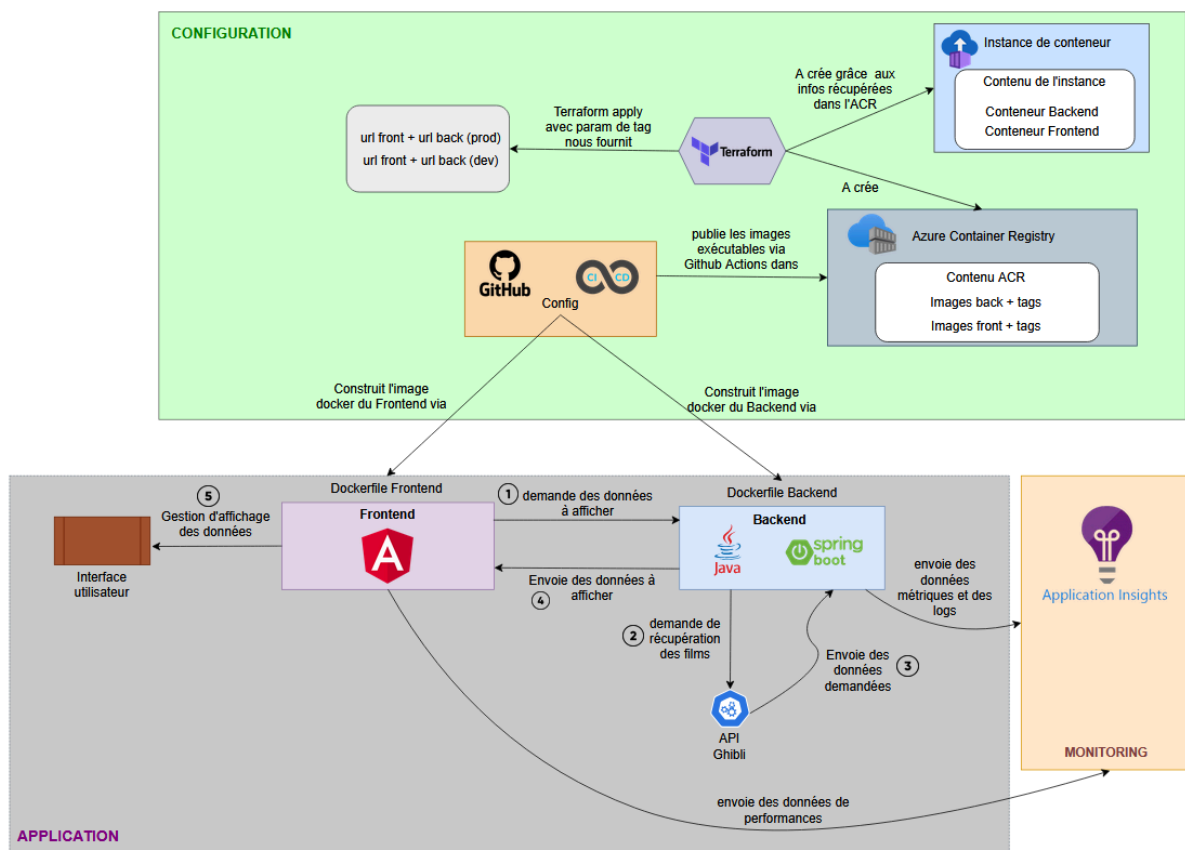
Configuration

- **GitHub Actions :**
 - Sert à déclencher la génération des images Docker pour le frontend (Angular) et le backend (Spring Boot).
 - Les configurations sont stockées dans GitHub pour les deux environnements (production et développement).
- **Azure Container Registry (ACR) :**
 - Stocke les images Docker du frontend et du backend, chacune taguée pour identifier la version ou l'environnement (prod/dev).
- **Terraform :**
 - Utilisé pour déployer et gérer l'infrastructure cloud.
 - Créer des instances de conteneur sur Azure, en récupérant les images Docker des registres d'Azure Container Registry (ACR).

Application

- Flux de traitement des données :
 - Comme expliqué dans la partie “interaction Frontend et Backend” le Backend se charge de récupérer les données de l'API et de les envoyer au Frontend pour la mise en forme et l'affichage.
- Dockers Files :
 - Le frontend et le backend sont chacun encapsulés dans des conteneurs Docker, définis par des fichiers Dockerfile respectifs.

Le monitoring dans le projet

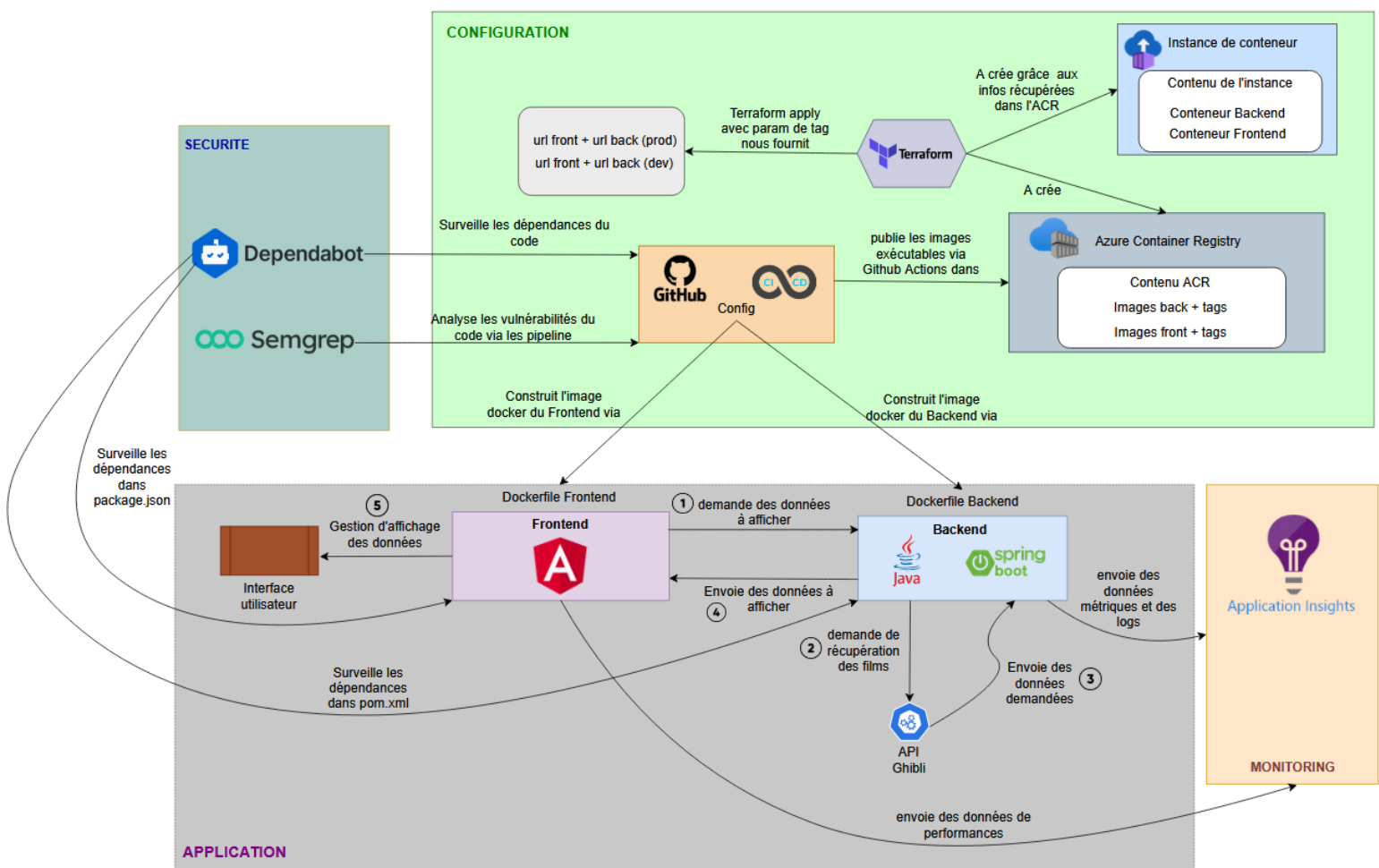


Dans l'image ci-dessus, le rôle du **monitoring**, représenté par **Application Insights**, est de surveiller et analyser les performances et les logs de l'application. Il collecte des données provenant des différentes parties de l'application, notamment le frontend et le backend, pour détecter les anomalies, mesurer les métriques comme les temps de réponse par exemple.

Ce monitoring assure la stabilité de l'application en offrant une visibilité complète sur son fonctionnement et en permettant de prendre des mesures correctives en cas de problème.

Le monitoring est expliqué d'une façon détaillée dans la partie [Monitoring](#) du rapport.

L'interaction de la sécurité avec les autres éléments du projet



Afin de sécuriser notre projet, nous avons intégré Semgrep et Dependabot, ces deux outils vont interagir avec le GitHub, le Frontend ainsi que le Backend de la façon suivante :

GitHub

- *Semgrep* analyse les pipelines CI/CD pour détecter des mauvaises pratiques, comme la mauvaise gestion des secrets ou l'absence de restrictions sur les permissions d'exécution des workflows.
- *Dependabot* vérifie les actions utilisées dans les workflows GitHub Actions pour s'assurer qu'elles sont à jour et sécurisées, réduisant ainsi les risques de failles dans la CI/CD.

Frontend

- *Semgrep* analyse les fichiers sources Angular pour détecter les vulnérabilités de sécurité potentielles ou les violations des bonnes pratiques de codage.
- *Dependabot* vérifie le fichier **package.json** et propose des mises à jour pour les bibliothèques Angular utilisées, en s'assurant qu'elles sont à jour et sécurisées.

Backend

- *Semgrep* analyse les fichiers sources Spring Boot pour détecter également les vulnérabilités de sécurité ainsi pour une vérification des configurations de sécurité (Spring Security)
- *Dependabot* surveille les dépendances Maven définies dans le fichier **pom.xml**. Il alerte sur les versions obsolètes ou contenant des vulnérabilités connues et propose des correctifs.

Fonctionnalités détaillées

Backend

Le Backend est composé d'un dossier de Config, de Contrôleur, de Modèle et de Service puis la classe de lancement du Backend.

- La config situé dans le dossier Config contient la configuration de CORS (Cross-Origin Resource Sharing) ainsi que la configuration de Spring Security.
- Le contrôleur comporte deux fonctions qui nous permettent d'accéder à deux endpoints :
 - **/films** pour récupérer tous les films Ghibli.
 - **/films/{id}** pour récupérer un film spécifique par son ID.
- Le service (**FilmService**) communique avec l'API Ghibli :
 - Il envoie des requêtes HTTP GET à l'API externe pour récupérer des données sous forme de JSON.
 - Il transforme ensuite la réponse JSON en objets Java (**Film[]**) pour plusieurs films et **Film** pour un film spécifique).
- Le modèle (**Film**) est utilisé pour représenter les données des films dans l'application. Il contient des attributs correspondant aux champs JSON renvoyés par l'API.
- Gestion des erreurs : Les erreurs HTTP sont capturées dans le contrôleur avec une gestion des exceptions basée sur le statut de l'erreur (par exemple, **404 Not Found** ou **400 Bad Request**).
- Tests unitaires : nous avons mis en place des tests unitaires et d'intégration pour valider les fonctionnalités du Backend concernant les films :
 - Un test de récupération d'une liste complète de films, les tests valident les scénarios où l'API retourne correctement les films disponibles ainsi que leurs détails, ce test couvre également le cas où la liste est vide ou où des erreurs HTTP
 - Un test de récupération d'un film par son ID, le test garantit que l'API retourne le film concerné par l'ID choisit et gère les erreurs appropriées, comme un ID inexistant par exemple
 - Une création dynamique des objets films a été mise en place pour simplifier et standardiser les tests sans dupliquer le code
 - Des mocks ont été utilisées pour simuler les interactions entre le contrôleur et le service externe d'API.
 - Nous avons utilisé MockMvc dans notre projet afin de garantir la robustesse et la fiabilité de notre API, en simulant des requêtes

HTTP avec MockMvc nous pouvons identifier et corriger les erreurs sans dépendre d'un environnement externe.

Frontend

Le Frontend est composé d'un dossier principal appelé "src", contenant un autre dossier appelé "app" et des fichiers de config. A l'intérieur de ce dernier on y trouve un dossier pour tout ce qui concerne les films et un autre contenant le service et les fichier de config de l'app

Le frontend est structuré autour d'un dossier principal nommé **src**, qui contient à la fois des fichiers de configuration et un sous-dossier **app**. À l'intérieur de ce dernier, nous trouvons :

- Un dossier component, regroupant les différents component correspondant à chaque pages :
 - Le component home
 - Le component film
 - Le component about
- Un dossier models, contenant le model Film et tous futurs models
- Un dossier service, regroupant les différents services
 - Un service : application-insights qui configure et utilise Application Insights pour collecter des données de télémétrie, comme les vues de pages, événements, exceptions, traces et métriques, afin de surveiller les performances et les comportements de l'application. Ce code va être mieux expliqué dans la partie MONITORING.
 - Un service : api.service.ts permettant de centraliser tous les appels api.
- Les fichiers de configuration spécifiques à l'application.
 - **app.component.ts** : Gère le composant principal avec suivi des pages et événements via Application Insights.
 - **app.config.ts** : Configure les routes, le client HTTP et optimise les performances.
 - **app.module.ts** : Configure les modules essentiels et intègre Application Insights.
 - **app.routes.ts** : Définit les routes principales pour la navigation entre l'accueil et les films.
 - etc

A la racine du projet nous trouvons aussi un fichier appelé "set-environments.js", ce fichier s'exécutera au même moment que les commandes "npm start" et "npm build" afin de générer les fichiers d'environnements automatiquement à partir des variables d'environnement. Ces dernières seront récupérées depuis github à travers le workflow.

Swagger

Nous avons choisi d'intégrer Swagger dans notre projet afin d'offrir une documentation interactive. Swagger fournit une interface utilisateur claire et intuitive, décrivant en détail les endpoints de l'API, les paramètres requis, les types de réponses attendues, ainsi que les codes de statut HTTP (comme 400, 401, etc.).

Grâce à Swagger, nous disposons d'une vue détaillée des services disponibles, ce qui facilite l'accès et la compréhension de l'API par les parties prenantes du projet ou par d'autres systèmes. Cette intégration améliore également la collaboration entre les équipes et les membres d'équipe, en rendant les échanges plus fluides et les développements mieux coordonnés.

Notre Swagger expose deux endpoints principaux :

1. **/films** : Fournit toutes les informations relatives à la récupération des films, notamment les codes d'erreur, le format des données, et les réponses en cas d'échec ou de succès.
2. **/films/{id}** : Décrit les détails nécessaires pour récupérer un seul film à partir de son ID, avec des informations similaires sur les erreurs et les réponses.

Le Swagger est accessible à partir de l'URL (en dev) suivant : <http://equipe5-dev.eastus.azurecontainer.io:8080/swagger-ui/index.html>

Le Swagger est accessible à partir de l'URL (en prod) suivant : <http://equipe5-prod.eastus.azurecontainer.io:8080/swagger-ui/index.html>

Sécurité

Dans cette section, nous présentons les mesures de sécurité mises en place pour chaque composante principale du projet, dans le but de garantir un système sécurisé et protégé :

Backend

Configuration CORS (Cross-Origin Resource Sharing) :

Nous avons implémenté une configuration Spring Boot pour activer et gérer le CORS via l'interface **WebMvcConfigurer**. Cette configuration se trouve dans le dossier **Config** et utilise une variable externe (**\${frontend.url}**) pour définir dynamiquement l'URL autorisée à accéder aux ressources backend.

- **Avantage** : La variable externalisée permet une flexibilité accrue et facilite l'automatisation, notamment lors de déploiements sur différents environnements (dev, prod, etc.).
- **Règles CORS** : La méthode **addCorsMappings** autorise toutes les routes (**/****), prend en charge les méthodes HTTP courantes (GET, POST, PUT, DELETE), et permet une gestion sécurisée des headers et des requêtes.

Spring Security :

Nous avons intégré **Spring Security** pour renforcer la sécurité des endpoints backend. Cette solution permet de :

- Rejeter automatiquement toute requête dirigée vers une URL non reconnue.
- Contrôler les URL autorisées et leur accès.
- Assurer une gestion sécurisée des requêtes en bloquant toute tentative non conforme. Par exemple, si un utilisateur malveillant tente d'ajouter un header ou un body inattendu dans une requête, celle-ci sera complètement ignorée.

Gestion de Swagger :

Le dossier de configuration inclut également les paramètres nécessaires pour sécuriser l'accès à l'interface Swagger, garantissant que seuls les utilisateurs autorisés puissent consulter ou interagir avec la documentation API.

Pourquoi le choix de Spring Boot ?

Nous avons choisi **Spring Boot** pour sa sécurité native et optimale dans la gestion des appels API. Il offre des mécanismes robustes pour prévenir les comportements malveillants, comme les tentatives d'injection de données dans les requêtes, et garantit une réponse rapide et sécurisée en cas de détection d'anomalies.

Frontend

Pour sécuriser le front, nous avons mis en place des fichiers d'environnements pour séparer l'environnement local de l'environnement de déploiement. Pour ce faire, l'url de l'api est renseigné dans les variables github. Ensuite lorsque le workflow s'exécute, ces variables github sont passées en variable d'environnement, afin que le script `set-environments.js`, puisse le récupérer et construire les fichiers d'environnement.

Terraform

Pour sécuriser au mieux notre projet, nous avons mis en place deux environnements distincts :

- Un environnement de développement (dev)
- Un environnement de production (prod)

Chaque environnement dispose de son propre Azure Container Registry, l'accès à ces environnement est restreint aux utilisateurs disposant des droits.

Les informations d'authentification pour accéder à l'ACR ne sont jamais codées en dur. Elles sont générées dynamiquement grâce aux propriétés de ressources définies dans Terraform, ce qui renforce la sécurité et réduit le risque d'exposition des données sensibles.

De plus, les variables d'environnement spécifiques aux différents contextes (dev et prod) sont gérées via des fichiers de variables dans Terraform, garantissant une configuration adaptée et isolée pour chaque environnement. Côté versioning, nous avons adopté des pratiques adaptées à chaque environnement. Pour la production, nous utilisons un système de versioning classique nommé `semver` (v1.0.0, v2.0.0, v3.0.0), tandis que pour le développement, nous avons utilisé `SHA` pour assurer une meilleure granularité et un suivi précis des modifications.

Nous avons configuré des limites de CPU et de mémoire pour les environnements de développement et de production. Cette configuration avait pour but d'offrir de la scalabilité à notre projet en garantissant une allocation optimale des ressources, empêchant toute surcharge ou gaspillage. En

ajustant ces limites, nous avons facilité la montée en charge de l'application, assurant ainsi qu'elle peut s'adapter efficacement à des variations de charge tout en maintenant ses performances et sa fiabilité.

Bien que nous aurions souhaité automatiser les déploiements Terraform via GitHub Actions, les restrictions liées à notre compte étudiant nous empêchent d'accéder au service principal nécessaire pour cette intégration. Malgré cette contrainte, la structure mise en place assure une séparation claire et sécurisée entre les environnements et facilite leur gestion.

GitHub

Du côté de GitHub, nous avons adopté plusieurs mesures pour garantir la sécurité de notre projet. Les informations sensibles, telles que les clés d'accès et les identifiants nécessaires pour l'authentification, sont stockées en tant que **Secrets** dans GitHub, et aucune donnée confidentielle n'est exposée directement dans les fichiers de configuration ou la CI/CD.

Nous avons également automatisé le déploiement des images Docker vers les Azure Container Registry des environnements de développement et de production à l'aide de pipelines GitHub Actions. Ces pipelines s'exécutent en utilisant les secrets configurés, ce qui permet un déploiement sécurisé et sans intervention manuelle.

Nous avons également mis en place des tests côté backend. Ceux-ci se lancent à chaque push sur la branche dev, main et les tags semver (utiles à la mise en production).

Monitoring

Pour mettre en place un système de monitoring, nous avons choisi d'intégrer Application Insights à notre projet, car ce dernier nous fournit une surveillance en temps réel de l'application.

Dans le **Frontend** : Application Insights est intégré pour suivre les interactions des utilisateurs et surveiller les performances. La classe **AppModule** configure le fournisseur Angular Plugin pour utiliser le service **ApplicationInsightsService**, qui initialise et gère Application Insights. Ce service est responsable de configurer le suivi automatique des changements de route (**enableAutoRouteTracking**) et de capturer des données comme les événements, les exceptions, les traces, et les métriques via des méthodes comme **logPageView**, **logEvent**, et **logException**. Dans le composant principal **AppComponent**, des interactions spécifiques sont enregistrées, comme les vues

de page et les clics de bouton. Cette configuration centralisée permet de collecter des données précieuses sur le comportement des utilisateurs et les performances de l'application, aidant à diagnostiquer les problèmes et à optimiser l'expérience utilisateur.

Dans le Backend : Application Insights est intégré principalement pour capturer des métriques et des données télémétriques afin de surveiller et diagnostiquer le comportement de l'application. La classe principale **BackAppApplication** utilise la méthode **ApplicationInsights.attach()** pour activer automatiquement le suivi télémétrique lors du démarrage de l'application. Cela permet de capturer des informations sur les requêtes HTTP, les erreurs, et d'autres événements importants sans nécessiter de configuration manuelle supplémentaire.

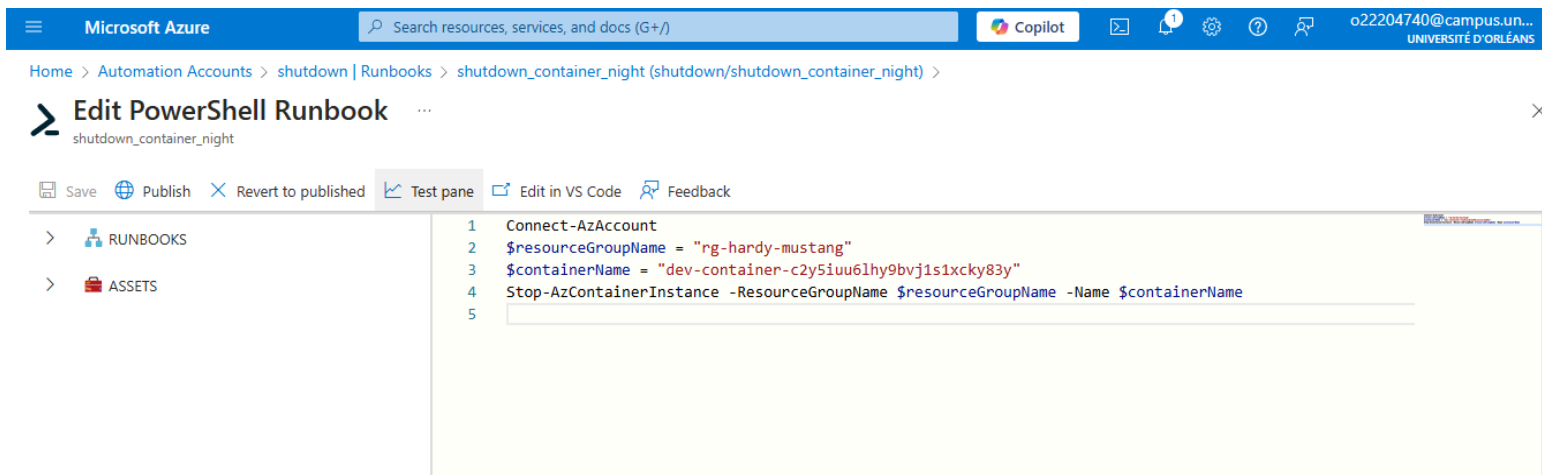
Malgré nos efforts pour intégrer Application Insights dans les différentes composantes du projet, son fonctionnement n'a pas pu être pleinement validé. Bien que la configuration ait été soigneusement mise en place côté Frontend et Backend pour capturer des données télémétriques et des interactions utilisateur, nous n'avons pas observé de remontée effective des données dans le tableau de bord d'Application Insights.

Nous avons également tenté de mettre en place un compte d'automatisation sur Azure, dans lequel nous avons intégré un runbook spécifiquement conçu pour arrêter les conteneurs pendant la nuit. Cette approche visait à éviter que les conteneurs fonctionnent en continu, réduisant ainsi la consommation inutile de ressources et les coûts associés. Cette solution aurait permis d'automatiser efficacement la gestion des conteneurs en fonction des périodes d'inactivité tout en maintenant une flexibilité opérationnelle pour leur redémarrage à des moments stratégiques.

The screenshot shows the Microsoft Azure portal interface. At the top, there's a blue header with the Microsoft Azure logo, a search bar, and user information. The main content area is divided into a left sidebar and a main panel. The sidebar shows the 'Automation Accounts' section, with 'shutdown' selected. The main panel displays the 'shutdown' Automation Account details, including a warning about the retirement of Log Analytics Agent-based Hybrid Runbook Worker and a table with fields like Resource group, Location, Subscription, Subscription ID, Status, and Last modified.

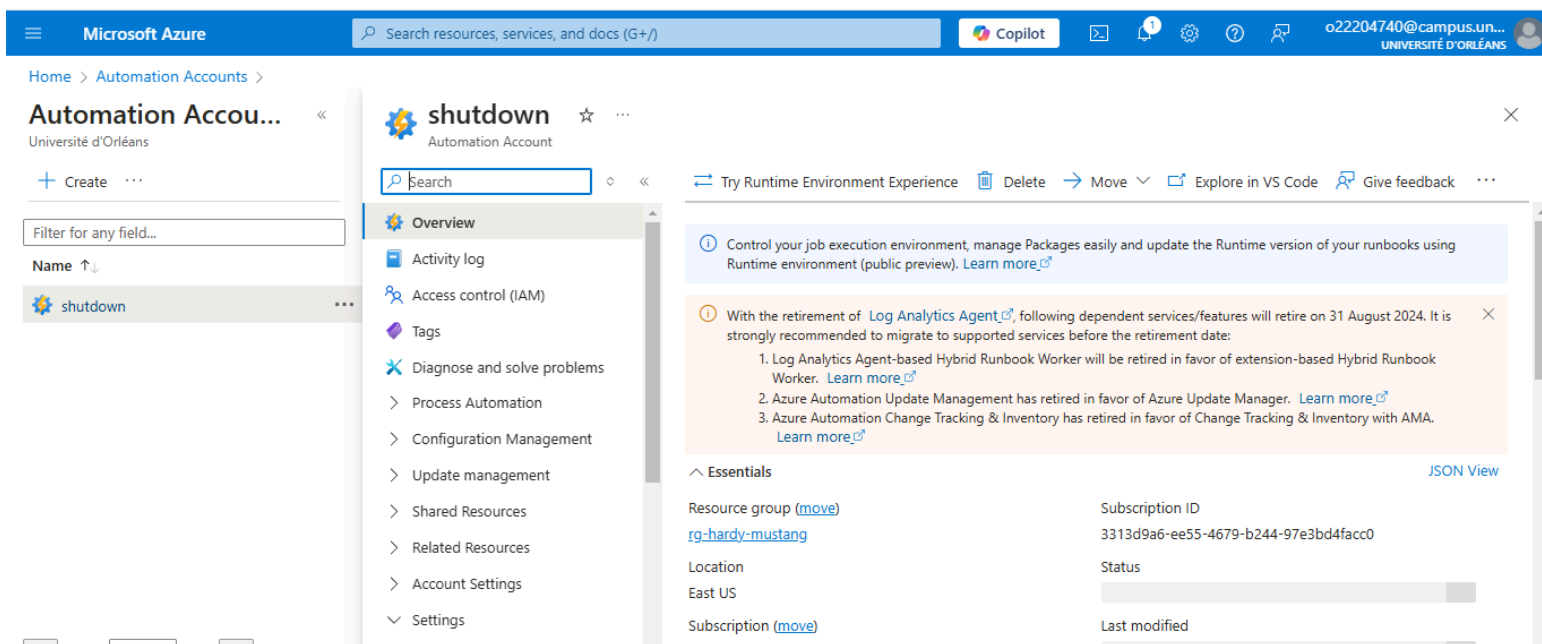
Field	Value
Resource group	rg-hardy-mustang
Location	East US
Subscription	(move)
Subscription ID	3313d9a6-ee55-4679-b244-97e3bd4facc0
Status	
Last modified	

Le code pour ce faire que nous avons essayé d'exécuter est le suivant



```
1 Connect-AzAccount
2 $resourceGroupName = "rg-hardy-mustang"
3 $containerName = "dev-container-c2y51uu6lhy9bvj1s1xcky83y"
4 Stop-AzContainerInstance -ResourceGroupName $resourceGroupName -Name $containerName
5
```

l'exécution de ce dernier a échoué car nous avons pas le rôle nécessaire pour effectuer cette action, en essayant de créer le rôle, un message d'erreur s'est affiché pour nous informer que nous avons pas les permissions pour créer de tel rôle, ce problème est peut être dû à notre compte azure etudiant.



Automation Account: shutdown

Search

Try Runtime Environment Experience Delete Move Explore in VS Code Give feedback

Overview

- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems
 - Process Automation
 - Configuration Management
 - Update management
 - Shared Resources
 - Related Resources
 - Account Settings
- Settings

Essentials

Resource group (move)	Subscription ID
rg-hardy-mustang	3313d9a6-ee55-4679-b244-97e3bd4facc0
Location	Status
East US	
Subscription (move)	Last modified

[JSON View](#)

Notification: With the retirement of [Log Analytics Agent](#), following dependent services/features will retire on 31 August 2024. It is strongly recommended to migrate to supported services before the retirement date:

- Log Analytics Agent-based Hybrid Runbook Worker will be retired in favor of extension-based Hybrid Runbook Worker. [Learn more](#)
- Azure Automation Update Management has retired in favor of Azure Update Manager. [Learn more](#)
- Azure Automation Change Tracking & Inventory has retired in favor of Change Tracking & Inventory with AMA. [Learn more](#)

Conclusion

Ce projet cloud a été conçu pour allier sécurité, performance et flexibilité. Grâce à Terraform, nous avons structuré deux environnements distincts. Suivi par une gestion des secrets, des registres de conteneurs et un suivi précis des versions pour les deux environnements (SHA en dev et semver en prod).

Nous avons mis en place un système de monitoring via Application Insights et nous avons sécurisé le projet à l'aide de la variabilisation des informations sensibles ou encore en ajoutant des outils comme Dependabot et Semgrep.

Sur le plan applicatif, les fonctionnalités du backend ont été testées rigoureusement grâce à des tests automatisés basés sur MockMvc, garantissant la fiabilité des endpoints API et la conformité avec les exigences fonctionnelles. Swagger a offert une documentation interactive, facilitant la collaboration et l'intégration avec d'autres équipes et systèmes.

Ce projet a également été une expérience enrichissante pour notre équipe. Il nous a permis de collaborer de manière efficace en répartissant les responsabilités tout en assurant une coordination fluide entre les différentes parties prenantes. Nous avons renforcé nos compétences en développement cloud, en sécurité et en déploiement automatisé, tout en découvrant l'importance des outils de monitoring et d'analyse dans un environnement moderne. Cette aventure nous a également appris à surmonter les défis techniques et organisationnels grâce à une approche structurée et une communication constante.

12 Facteurs

[1. L'application doit être maintenue dans un système de contrôle de version.](#)

[2. Les dépendances doivent être isolées et déclarées de manière explicite.](#)

[3. La configuration et le code doivent être strictement séparés.](#)

[Stockez la configuration à l'extérieur de l'application \(variables d'environnement\).](#)

4. Le code d'une application ne fait pas de différence entre le local et un service tiers.

[Voir le code source](#)

5. Séparez de manière stricte les étapes de Build, Release et de Run.

[Utilisation de Dockerfiles](#)

6. Les processus sont sans état (stateless) et ne se partagent rien.

[Voir le code source](#)

[7. L'application doit être self-contained. La gestion des ports est gérée par la plateforme cloud.](#)

[8. Chaque type de travail de l'application est associé à un type de processus.](#)

[9. Augmentez la robustesse avec des démarrages rapides et des arrêts propres.](#)

[10. Gardez les environnements de développement, staging et de production les plus similaires possibles.](#)

11. Traiter les journaux (logs) comme un flux d'événements. L'application ne doit pas s'occuper du routage ou du stockage du flux généré en sortie.

[12. Les tâches de maintenance/administration sont versionnées avec le code et sont mises à jour avec les releases. Les tâches sont entièrement automatisées.](#)