



Mise en place des pratiques DevOps : du développement au déploiement

Ce travail est réalisé par :
Douaa BOUCHABOUD
Randa EL MAAZOUZA
Meryem DAHHANE
Nada AHASSAD
Abrar EL AMIRI

Introduction

Dans le cadre de ce projet, nous avons choisi de mettre en œuvre les principes fondamentaux de la méthodologie DevOps, en nous concentrant sur la culture, l'automatisation, et l'optimisation des processus de développement et de déploiement. Le principal objectif de ce travail est de créer une application moderne, dédiée à la gestion d'un magasin de produits électroniques, en appliquant les bonnes pratiques DevOps afin de garantir une livraison continue, fiable et scalable.

Le rapport détaillera chaque étape de la mise en œuvre de l'application, de sa conception architecturale à son déploiement final, en passant par la conteneurisation avec Docker et l'automatisation via un pipeline CI/CD. L'application a été développée selon une architecture basée sur des microservices, permettant une meilleure modularité et scalabilité. Chaque service a été conteneurisé à l'aide de Docker, et l'ensemble du système a été déployé sur le cloud Azure.

L'objectif de ce rapport est de démontrer comment la mise en œuvre des pratiques DevOps a permis d'optimiser le développement et le déploiement de notre application, tout en assurant une collaboration efficace au sein de l'équipe.



SOMMAIRE

| | |
|------------------------------------|---|
| Architecture de l'Application | 1 |
| Conteneurisation avec Docker | 2 |
| Déploiement sur Azure | 3 |
| Automatisation CI/CD | 4 |
| Collaboration et Gestion de Projet | 5 |
| Tests et Mesures de Performance | 6 |

1. Architecture de l'Application

1. Architecture Utilisée

Ce projet repose sur une architecture microservices où un premier service produits (App1) gère la liste des produits et un deuxième service magasin (App2) affiche ces produits pour l'utilisateur via une interface React.

Nous avons choisi une architecture basée sur deux microservices qui communiquent via HTTP .

- App1 (Service des Produits) : Stocke et fournit la liste des produits.

```
✓ app1
  > node_modules
  JS App.js
  {} package-lock.json
  {} package.json
```

- App2 (Service magasin) : Récupère les produits depuis App1 et les affiche à l'utilisateur via une interface React.

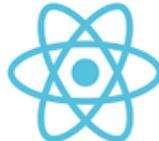
```
✓ app2
  > node_modules
  JS App2.js
  {} package-lock.json
  {} package.json
```

- Frontend (React) : Interface utilisateur pour Afficher les produits disponible..

```
✓ frontend
  > node_modules
  > public
  > src
  ≡ .gitignore
  {} package-lock.json
  {} package.json
  ⓘ README.md
```

1. Architecture de l'Application

2. Technologies utilisées :

| | |
|-----------------------------------|--|
| Backend |  |
| Frontend |  React JS |
| Communication entre microservices | API REST + HTTP |

3. Aperçu de l'interface d'affichage

| Liste des Produits | | | | | | |
|--------------------|---|---------------------|--------|-------|----------|--|
| ID | Image | Nom | Prix | Stock | Statut | |
| 1 |  | Ordinateur Portable | \$1200 | 20 | En Stock | |
| 2 |  | Clavier Mécanique | \$50 | 15 | En Stock | |
| 3 |  | Souris Gaming | \$30 | 10 | En Stock | |
| 4 |  | Écran 27" 4K | \$400 | 8 | En Stock | |
| 5 |  | Casque Audio | \$150 | 12 | En Stock | |

2. Conteneurisation avec Docker

La conteneurisation est une approche clé dans la mise en œuvre d'un processus DevOps, permettant de garantir que les applications s'exécutent de manière identique, peu importe où elles sont déployées. Docker est un outil de conteneurisation qui permet de créer, déployer et exécuter des applications dans des conteneurs.

Dans cette section, nous détaillons les étapes que nous avons suivies pour conteneuriser nos applications (App1 et App2 en Express et le Frontend en React), et comment nous avons utilisé Docker et Docker Compose pour gérer ces conteneurs dans un environnement de développement.

1. Création des Dockerfiles

Un Dockerfile est un fichier texte contenant des instructions nécessaires à la création d'une image Docker. Nous avons créé un Dockerfile pour chaque application (App1 et App2), permettant de configurer l'environnement d'exécution pour nos applications Express.

Voici un exemple du Dockerfile pour App1 :

```
app1 > 🛠 Dockerfile > ...
1   FROM node:18-alpine
2
3   WORKDIR /app
4
5   COPY package*.json .
6   RUN npm install
7
8   COPY . .
9
10  EXPOSE 3001
11
12  CMD ["node", "App.js"]
13
```

2. Conteneurisation avec Docker

- FROM node:18-alpine : Cette ligne définit l'image de base pour le conteneur. Ici, nous utilisons l'image officielle de Node.js basée sur Alpine Linux pour des raisons de légèreté.
- WORKDIR /app : Cette ligne crée un répertoire de travail à l'intérieur du conteneur. C'est ici que tous les fichiers de l'application seront copiés.
- COPY package.json ./ : Cette ligne copie les fichiers package.json et package-lock.json dans le répertoire de travail du conteneur. Cela permet d'installer les dépendances.
- RUN npm install : Installe les dépendances de l'application définies dans le package.json.
- COPY . . : Copie tout le code source de l'application dans le conteneur.
- EXPOSE 3001 : Expose le port 3001 pour que l'application soit accessible depuis l'extérieur du conteneur.
- CMD ["node", "App.js"] : Démarrer l'application avec la commande node App.js.

Le Dockerfile pour App2 est identique à celui d'App1, avec quelques ajustements concernant le port.

Le Dockerfile pour le Frontend React également construit de manière similaire, mais il nécessitera une étape de construction spécifique à React avant de servir l'application.

```
frontend > 🚀 Dockerfile > ...
1   FROM node:18-alpine
2
3   WORKDIR /app
4
5   COPY package*.json ./
6   RUN npm install
7
8   COPY . .
9
10  EXPOSE 3000
11
12  CMD ["npm", "start"]
```

2. Conteneurisation avec Docker

- RUN npm run build : Cette étape génère les fichiers statiques optimisés pour React, qui seront ensuite servis par un serveur.
- CMD ["npm", "start"] : Lance le serveur de développement React.

2. Création du fichier Docker Compose

Le Docker Compose permet de définir et de gérer plusieurs conteneurs Docker dans un seul fichier YAML. Nous avons utilisé Docker Compose pour configurer l'intégration de nos trois services : App1, App2, et le Frontend.

Voici le fichier docker-compose.yml :

```
# docker-compose.yml > {} services > {} app2 > [ ] volumes
    docker-compose.yml - The Compose specification establishes a standard for the definition of multi-container platform-agnostic applications (compose-spec.json)
1   version: '3.8'
2
3     ▷ Run All Services
4   services:
5     ▷ Run Service
6       app1:
7         build: ./app1
8         ports:
9           - "3001:3001"
10        volumes:
11          - ./app1:/app
12          - /app/node_modules
13        environment:
14          - NODE_ENV=production
15
16     ▷ Run Service
17       app2:
18         build: ./app2
19         ports:
20           - "3002:3002"
21         volumes:
22           - ./app2:/app
23           - /app/node_modules
24         environment:
25           - NODE_ENV=production
26
27     ▷ Run Service
28       frontend:
29         build: ./frontend
30         ports:
31           - "3000:3000"
32         volumes:
33           - ./frontend:/app
34           - /app/node_modules
35         stdin_open: true
36         tty: true
37         depends_on:
38           - app1
39           - app2
```

2. Conteneurisation avec Docker

- services : Définit les services qui seront utilisés par Docker Compose. Ici, trois services : app1, app2, et frontend.
- build : Indique où se trouve le Dockerfile de chaque application. Cela permet de construire l'image correspondante.
- ports : Mappe les ports entre l'hôte et le conteneur. Par exemple, 3001:3001 mappe le port 3001 de l'hôte au port 3001 du conteneur.
- volumes : Permet de lier les fichiers du projet local au conteneur. Cela permet de travailler directement sur les fichiers locaux et de voir les changements instantanément.
- depends_on : Assure que les services app1 et app2 sont lancés avant le frontend.

3. Création et Push des Images Docker

Après avoir configuré les Dockerfiles et le fichier Docker Compose, nous avons exécuté les commandes suivantes pour créer et pousser les images Docker vers Docker Hub.

- Construire les images Docker :

```
docker-compose build
```

Cette commande génère les images Docker pour tous les services définis dans le fichier docker-compose.yml. Cela inclut App1, App2 et le Frontend.

Voici le résultat de cette commande avec `docker images` pour vérifier que les images ont bien été créées.

```
PS C:\Users\hp\Downloads\devopsa\devopsa> docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
devopsa-frontend    latest   01bf4910c9ac  10 minutes ago  699MB
devopsa-app1         latest   2cda677fccec2  12 minutes ago  139MB
devopsa-app2         latest   552d9bcdba54  12 minutes ago  139MB
liliafxaki/hadoop-cluster  latest   b90f134f6fcfd  6 months ago   4.92GB
PS C:\Users\hp\Downloads\devopsa\devopsa>
```

- Tagger les images :

```
docker tag devopsa-app1 randamzz/devopsa-app1:latest
```

```
docker tag devopsa-app2 randamzz/devopsa-app2:latest
```

```
docker tag devopsa-frontend randamzz/devopsa-frontend:latest
```

Ces commandes ajoutent des tags aux images locales pour qu'elles puissent être identifiées sur Docker Hub. Cela permet de gérer facilement les versions.

2. Conteneurisation avec Docker

- Pousser les images vers Docker Hub :

```
docker push randamzz/devopsa-app1:latest
```

```
docker push randamzz/devopsa-app2:latest
```

```
docker push randamzz/devopsa-frontend:latest
```

Ces commandes poussent les images Docker vers le registre Docker Hub, ce qui permet de les partager et de les déployer sur d'autres serveurs.

The screenshot shows the Docker Hub website. At the top, there is a blue header bar with the Docker Hub logo, navigation links for 'Explore', 'Repositories' (which is underlined), 'Organizations', and 'Usage', and a search bar with the placeholder 'Search Docker Hub' and a 'ctrl+K' keyboard shortcut. Below the header, there is a search bar with the value 'randamzz', a global search bar, a dropdown for 'All content', and a 'Create a repository' button. The main content area displays a table of three Docker repositories:

| Name | Last Pushed | Contains | Visibility | Scout |
|---------------------------|-------------------|----------|------------|----------|
| randamzz/devopsa-frontend | about 5 hours ago | IMAGE | Public | Inactive |
| randamzz/devopsa-app2 | about 5 hours ago | IMAGE | Public | Inactive |
| randamzz/devopsa-app1 | about 5 hours ago | IMAGE | Public | Inactive |

At the bottom of the table, it says '1-3 of 3'.

Cette dernière figure montre que les images ont bien été poussées vers un dépôt distant.

Conclusion

Nous avons réussi à conteneuriser nos applications App1, App2, et le Frontend en utilisant Docker et Docker Compose. Ce processus a permis de rendre notre environnement de développement cohérent et facilement reproduit. En poussant les images sur Docker Hub, nous avons facilité le déploiement de l'application dans des environnements différents, garantissant ainsi une intégration continue et une livraison continue (CI/CD).

3. Déploiement sur Azure

L'utilisation d'Azure pour la gestion et le déploiement de microservices permet d'exploiter la puissance du cloud tout en assurant une scalabilité optimale. Azure Container Registry (ACR) facilite le stockage et la gestion des images Docker, tandis qu'Azure App Service permet leur déploiement sans nécessiter de gestion d'infrastructure complexe.

Dans cette section, nous détaillons les étapes que nous avons suivies pour créer un registre de conteneurs sur Azure (ACR), configurer Azure App Service pour héberger nos microservices, pousser nos images Docker vers ACR, puis déployer ces microservices sur Azure App Service afin d'assurer une exécution fluide et efficace de notre application.

1. Création d'une Azure Container Registry (ACR)

Azure Container Registry (ACR) est un service de gestion d'images Docker permettant de stocker, sécuriser et distribuer des conteneurs sur Azure.

- Nous avons ouvert "[Azure Portal](#)" et nous nous sommes connectés avec notre compte étudiant.
- Ensuite, nous avons recherché "[Azure Container Registry \(ACR\)](#)" dans la barre de recherche.
- Nous avons accédé à la section "Registres de conteneurs".
- Nous avons cliqué sur "[Créer un registre de conteneurs](#)", ce qui nous a dirigés vers la page de configuration :

 **Créer un Registre de conteneurs** ...

pour tous les types de déploiement de conteneurs. Utilisez les registres de conteneurs Azure avec vos pipelines de développement et de déploiement de conteneurs existants. Utilisez Azure Container Registry Tasks pour générer des images conteneur dans Azure à la demande, ou pour automatiser les builds déclenchées par les mises à jour du code source, les mises à jour de l'image de base d'un conteneur ou les minutiers. [En savoir plus](#)

Détails du projet

Abonnement *

Groupe de ressources * [Créer nouveau](#)

Détails de l'instance

Nom du Registre * [.azurecr.io](#)

Emplacement *

Utiliser des zones de disponibilité Les zones de disponibilité sont activées sur les registres Premium et dans les régions qui prennent en charge les zones de disponibilité. [En savoir plus](#)

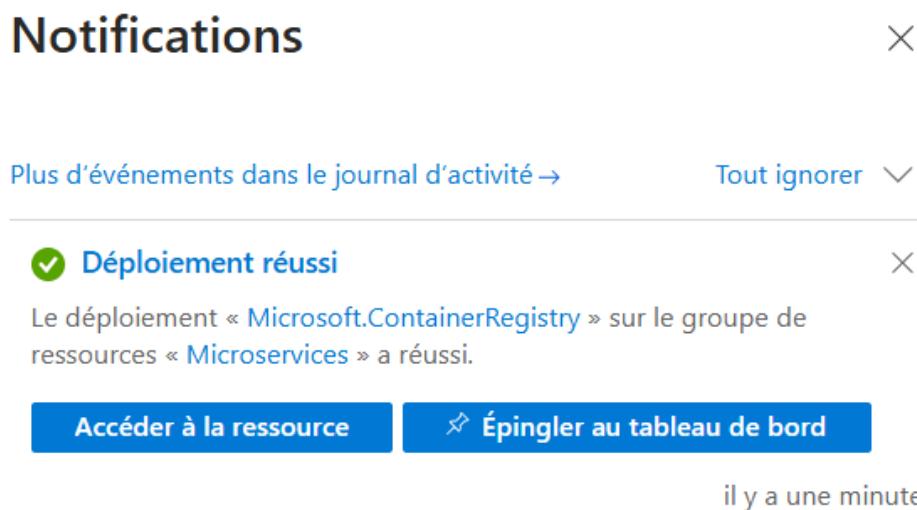
Plan de tarification *

Vérifier + créer [< Précédent](#) [Suivant: Réseau >](#)

3. Déploiement sur Azure

1. **Choisir l'abonnement** : Sélectionnez votre abonnement, par exemple "Azure for Students".
2. **Sélectionner un groupe de ressources** : Créez un nouveau groupe "Microservices".
3. **Configurer l'instance** : Définissez le nom du registre (par exemple "Myregistryy"), sélectionnez l'emplacement (par exemple "France Central"), et choisissez le plan "Standard".
4. **Vérifier et créer** : Confirmez la configuration et cliquez sur "Vérifier + Créer" pour finaliser.

Après ces étapes, notre ACR est créé, comme le montre la notification suivante :



2. Configuration d'Azure App Service

Azure App Service offre une plateforme entièrement managée qui prend en charge l'hébergement des applications Docker sans nécessiter de gestion d'infrastructure.

Dans cette étape, nous avons configuré App Service pour qu'il puisse récupérer nos images Docker stockées dans Azure Container Registry (ACR)

- Nous avons recherché "Azure App Service" dans la barre de recherche.
- Nous avons accédé à la section "App Services".
- Nous avons cliqué sur "Créer" et sélectionné "Web App", ce qui nous a dirigés vers la page de configuration pour app1 :

3. Déploiement sur Azure

Accueil > App Services >

Créer une application web ...

Détails du projet

Selectionnez un abonnement pour gérer les coûts et les ressources déployées. Utilisez les groupes de ressources comme des dossiers pour organiser et gérer toutes vos ressources.

| | |
|------------------------|-------------------------------|
| Abonnement * | Azure for Students |
| Groupe de ressources * | Microservices |
| | Créer nouveau |

Détails de l'instance

| | |
|-----|--|
| Nom | app1-service |
| | -e0b3cvebfqesebf5.francecentral-01.azurewebsites.net |

Sécurisez le nom d'hôte par défaut unique activé. [Découvrir plus d'informations sur cette mise à jour](#)

Publier * Conteneur

Système d'exploitation * Windows

Région * France Central

i Vous ne trouvez pas votre plan App Service ? Essayez une autre région ou sélectionnez votre environnement App Service Environment.

Vérifier + créer

< Précédent

Suivant : Base de données >

- **Choisir l'abonnement :** Sélectionnez votre abonnement, par exemple "Azure for Students".
- **Sélectionner un groupe de ressources :** Choisissez le groupe de ressources "Microservices".
- **Configurer l'instance :** Définissez le nom de l'application, par exemple "app1-service", sélectionnez le système d'exploitation (Linux), et choisissez la région (par exemple "France Central").
- **Poursuivre la configuration :** Cliquez sur "Suivant" jusqu'à ce que vous atteigniez la section "Conteneur" :

3. Déploiement sur Azure

Accueil > App Services >

Créer une application web

...

De base Base de données **Conteneur** Réseau Superviser + sécuriser Balises Vérifier + créer

Selectionnez votre source préférée pour les images conteneur. Vous pouvez modifier ces paramètres et d'autres dépendances après avoir créé l'application. [En savoir plus](#)

Prise en charge des sidecars Configuration améliorée avec prise en charge des sidecars désactivée [En savoir plus](#)

Source d'image * Démarrage rapide
 Registre de conteneurs Azure
 Autres registres de conteneurs

Options de registre de conteneurs Azure

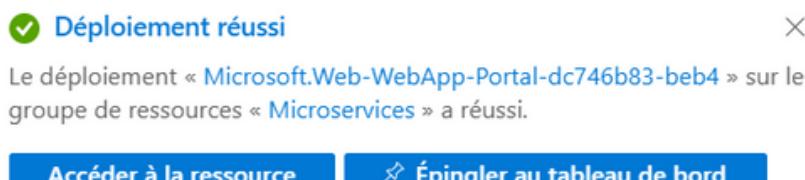
Registre *

Authentication * Managed identity
 Admin credentials

Identity *

- Vérifier la source de l'image :** Sélectionnez votre source d'image conteneur. Par défaut, vous pouvez choisir "Registre de conteneurs Azure" et définir le registre que vous avez créé, comme "Myregistryy".
- Configurer l'authentification :** Choisissez le type d'authentification pour le registre. Vous pouvez opter pour "Managed identity" ou "Admin credentials".
- Finaliser la configuration :** Une fois tous les paramètres définis, cliquez sur "Vérifier + Créer" pour vérifier la configuration avant de finaliser la création de l'application.

Une fois que l'application web est correctement créée, une notification s'affiche pour confirmer sa réussite :



Nous avons ensuite répété le même processus pour créer les applications "App2" et "Frontend".

3. Déploiement sur Azure

Voici la liste des ressources créées :

| Ressources | | | |
|--------------------|--------|----------------------|-------------------|
| Récent | Favori | Type | Dernier affichage |
| 🌐 frontend-service | | App Service | il y a une minute |
| 🌐 Microservices | | Groupe de ressources | il y a une minute |
| 🌐 app2-service | | App Service | il y a 8 minutes |
| 🌐 app1-service | | App Service | il y a 14 minutes |

3. Push des images Docker vers Azure Container Registry

Une fois l'Azure Container Registry (ACR) créé, la prochaine étape consiste à pousser les images Docker vers ce registre. Cette opération permet de stocker les images Docker dans le registre Azure, prêtes à être utilisées pour le déploiement sur Azure App Service ou d'autres services cloud.

- Nous avons ouvert le terminal de notre projet dans VS Code.
- Nous avons listé les images avec la commande `docker images`

| C:\Users\hp>docker images | | | | | |
|---------------------------|--------|--------------|--------------|--------|--|
| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE | |
| devopsa-frontend | latest | 01bf4910c9ac | 46 hours ago | 699MB | |
| devopsa-app2 | latest | 552d9bcd54 | 46 hours ago | 139MB | |
| devopsa-app1 | latest | 2cda677fcec2 | 46 hours ago | 139MB | |
| liliastaxi/hadoop-cluster | latest | b90f134f6fcf | 6 months ago | 4.92GB | |

Pour taguer les images, nous avons exécuté les commandes suivantes :

`Docker tag devopsa-app1 myregistryy.azurecr.io/devopsa-app1:latest`

`Docker tag devopsa-app2 myregistryy.azurecr.io/devopsa-app2:latest`

`Docker tag devopsa-frontend myregistryy.azurecr.io/devopsa-frontend:latest`

```
C:\Users\hp>Docker tag devopsa-frontend myregistryy.azurecr.io/devopsa-frontend:latest
C:\Users\hp>Docker tag devopsa-app2 myregistryy.azurecr.io/devopsa-app2:latest
C:\Users\hp>Docker tag devopsa-app1 myregistryy.azurecr.io/devopsa-app1:latest
```

Nous avons exécuté à nouveau la commande docker images pour vérifier que les images taguées apparaissent dans la liste :

| C:\Users\hp>docker images | | | | | |
|---|--------|--------------|--------------|--------|--|
| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE | |
| devopsa-frontend | latest | 01bf4910c9ac | 46 hours ago | 699MB | |
| myregistryy.azurecr.io/devopsa-frontend | latest | 01bf4910c9ac | 46 hours ago | 699MB | |
| devopsa-app2 | latest | 552d9bcd54 | 46 hours ago | 139MB | |
| myregistryy.azurecr.io/devopsa-app2 | latest | 552d9bcd54 | 46 hours ago | 139MB | |
| devopsa-app1 | latest | 2cda677fcec2 | 46 hours ago | 139MB | |
| myregistryy.azurecr.io/devopsa-app1 | latest | 2cda677fcec2 | 46 hours ago | 139MB | |
| liliastaxi/hadoop-cluster | latest | b90f134f6fcf | 6 months ago | 4.92GB | |

3. Déploiement sur Azure

- Puis, nous avons exécuté la commande `docker login myregistryy.azurecr.io` et fourni le nom d'utilisateur (nom du registre) et le mot de passe associé.
- La connexion a été réussie, comme indiqué par le message "Login Succeeded".

```
C:\Users\hp>docker login myregistryy.azurecr.io
Username: Myregistryy
Password:

Login Succeeded
```

Une fois connectés, nous avons exécuté les commandes suivantes pour pousser les images Docker vers Azure Container Registry (ACR) :

```
docker push myregistryy.azurecr.io/devopsa-app1:latest
docker push myregistryy.azurecr.io/devopsa-app2:latest
docker push myregistryy.azurecr.io/devopsa-frontend:latest
```

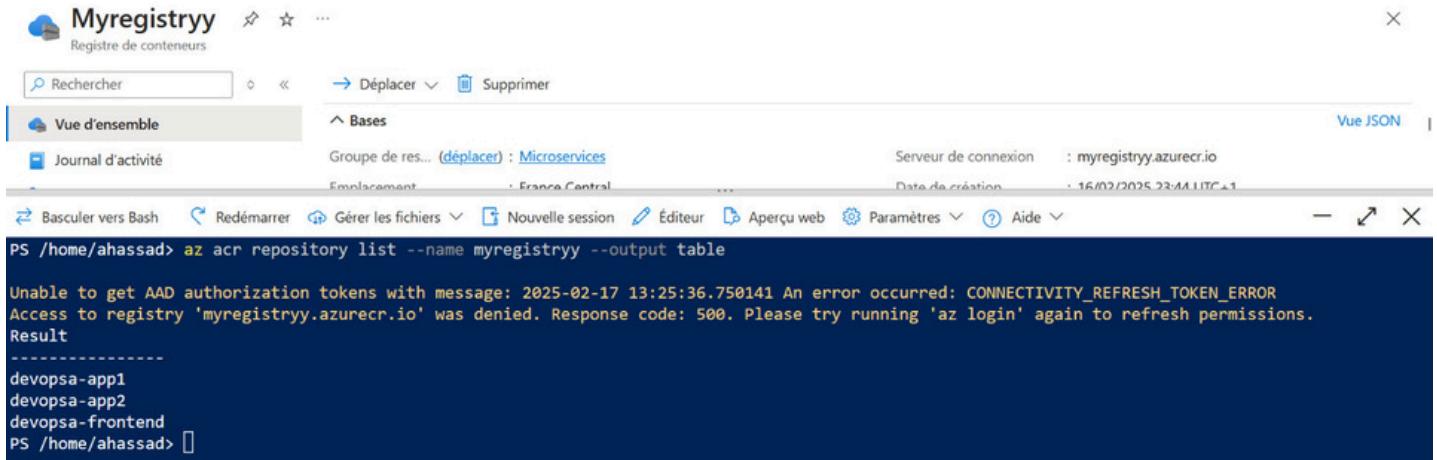
```
C:\Users\hp>Docker push myregistryy.azurecr.io/devopsa-app1:latest
The push refers to repository [myregistryy.azurecr.io/devopsa-app1]
ec07931d6954: Pushed
cdd7e1c0d345: Pushed
2453ac355cda: Pushed
5d9ba1596fb3: Mounted from devopsa-frontend
a9cfa7c9c02b: Mounted from devopsa-frontend
77296e765ec2: Mounted from devopsa-frontend
3766efcb8e44: Mounted from devopsa-frontend
08000c18d16d: Mounted from devopsa-frontend
latest: digest: sha256:c2f1f515baldd8d4d11c25a7c9b70f1577aa98f16171831ad1c1dbe297357cf6 size: 1994

C:\Users\hp>Docker push myregistryy.azurecr.io/devopsa-app2:latest
The push refers to repository [myregistryy.azurecr.io/devopsa-app2]
d045acdb8bc8: Pushed
ef6cbae294fc: Pushed
46e1d4c68a82: Pushed
5d9ba1596fb3: Mounted from devopsa-app1
a9cfa7c9c02b: Mounted from devopsa-app1
77296e765ec2: Mounted from devopsa-app1
3766efcb8e44: Mounted from devopsa-app1
08000c18d16d: Mounted from devopsa-app1
latest: digest: sha256:d0bd11cc38a5545039873c053f68f74ee9b8b85a8b9be76c2b57e296aa460466 size: 1994
```

Pour s'assurer que nos images ont bien été poussées vers l'ACR, nous avons suivi ces étapes :

- Nous nous sommes connectés au portail Azure (<https://portal.azure.com>).
- Nous avons cliqué sur l'icône du Cloud Shell en haut à droite de la page du portail (l'icône ressemblait à un terminal ou une fenêtre de ligne de commande).
- Nous avons sélectionné Bash ou PowerShell en fonction de notre préférence.
- Nous avons exécuté cette commande `az acr repository list`

3. Déploiement sur Azure



The screenshot shows the Azure Container Registry interface for a repository named 'myregistryy'. The 'Vue d'ensemble' (Overview) tab is selected. A command-line session is open in the bottom pane, showing the output of the command 'az acr repository list --name myregistryy --output table'. The output indicates an error due to missing AAD authorization tokens.

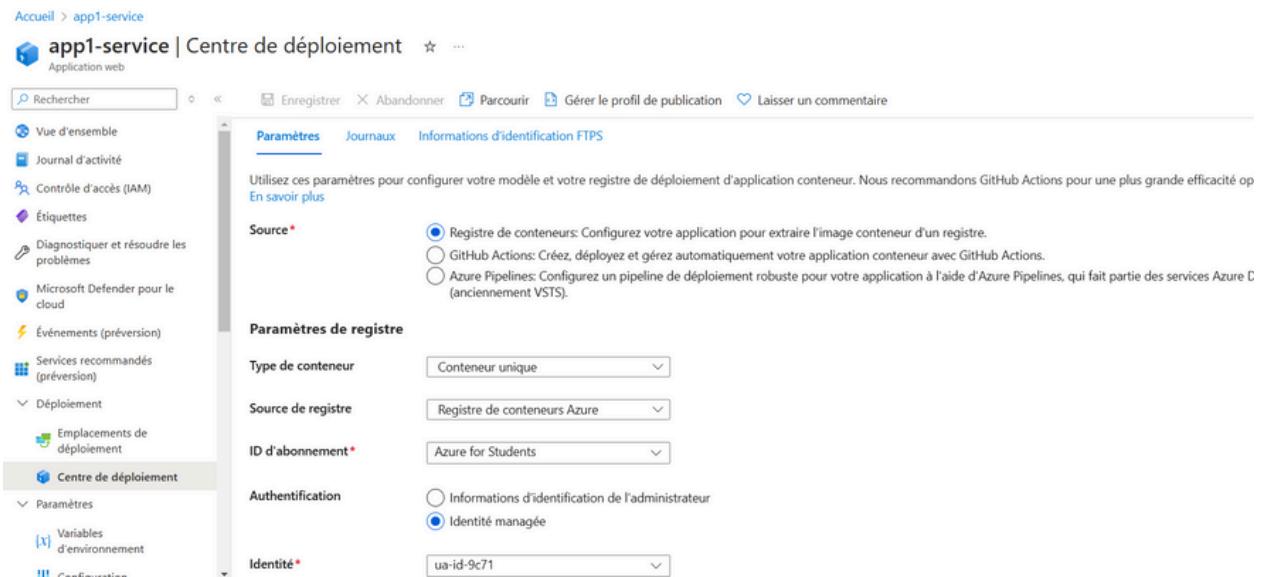
```
PS /home/ahassad> az acr repository list --name myregistryy --output table
Unable to get AAD authorization tokens with message: 2025-02-17 13:25:36.750141 An error occurred: CONNECTIVITY_REFRESH_TOKEN_ERROR
Access to registry 'myregistryy.azurecr.io' was denied. Response code: 500. Please try running 'az login' again to refresh permissions.
Result
-----
devopsa-app1
devopsa-app2
devopsa-frontend
PS /home/ahassad> 
```

4. Déploiement des microservices sur Azure App Services

Une fois les images Docker poussées vers Azure Container Registry, nous avons procédé au déploiement de nos microservices sur Azure App Service.

- Nous avons cliqué sur notre "app1-service."
- Nous avons recherché "Centre de déploiement" dans le menu de gauche.
- Nous avons ensuite procédé à la configuration.

Voici la page qui s'affiche :



The screenshot shows the 'Centre de déploiement' (Deployment Center) page for the 'app1-service' web app. The left sidebar shows navigation options like 'Vue d'ensemble', 'Journal d'activité', and 'Centre de déploiement' (which is currently selected). The main area displays deployment parameters. Under 'Paramètres', the 'Source' section is set to 'Registre de conteneurs' (Container Registry). Other fields include 'Type de conteneur' (Unique container), 'Source de registre' (Azure Container Registry), 'ID d'abonnement' (Azure for Students), 'Authentification' (Managed identity), and 'Identité' (ua-id-9c71).

Nous avons choisi "Registre de conteneur" comme source, puis nous avons configuré les champs suivants :

3. Déploiement sur Azure

Paramètres de registre

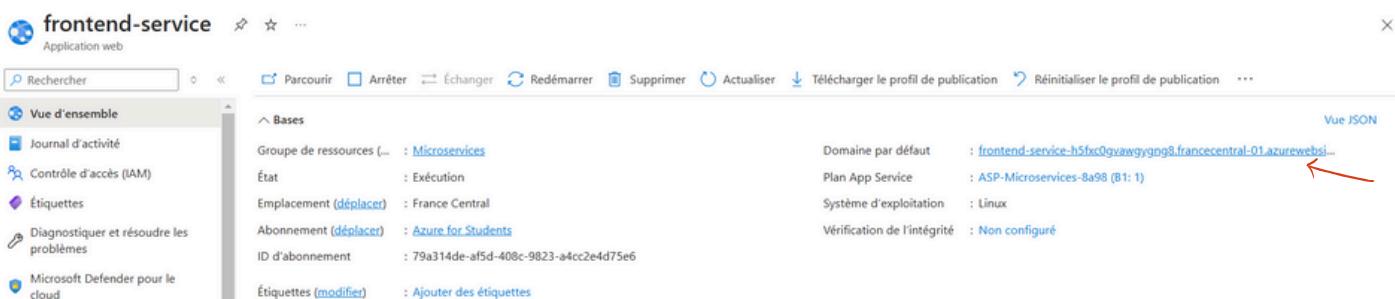
| | |
|--------------------------|--|
| Type de conteneur | Conteneur unique |
| Source de registre | Registre de conteneurs Azure |
| ID d'abonnement* | Azure for Students |
| Authentification | <input checked="" type="radio"/> Informations d'identification de l'administrateur <input type="radio"/> Identité managée |
| Registre* | Myregistryy |
| Image* | devopsa-app1 |
| Balise* | latest |
| Fichier ou commande d... | |
| Déploiement continu | <input type="radio"/> Activé <input checked="" type="radio"/> Désactivé |

Nous avons sélectionné les paramètres suivants pour la configuration du registre :

- **Type de conteneur** : "Conteneur unique".
- **Source de registre** : "Registre de conteneurs Azure".
- **ID d'abonnement** : "Azure for Students".
- **Authentification** : "Identité managée".
- **Registre** : "Myregistryy".
- **Image** : "devopsa-app1".
- **Balise** : "latest".
- **Déploiement continu** : Nous avons activé le déploiement continu.

Nous avons effectué la même configuration pour les services app2-service et frontend-service.

Puis, nous avons cliqué sur "Vue d'ensemble" pour trouver le lien de déploiement :



frontend-service Application web

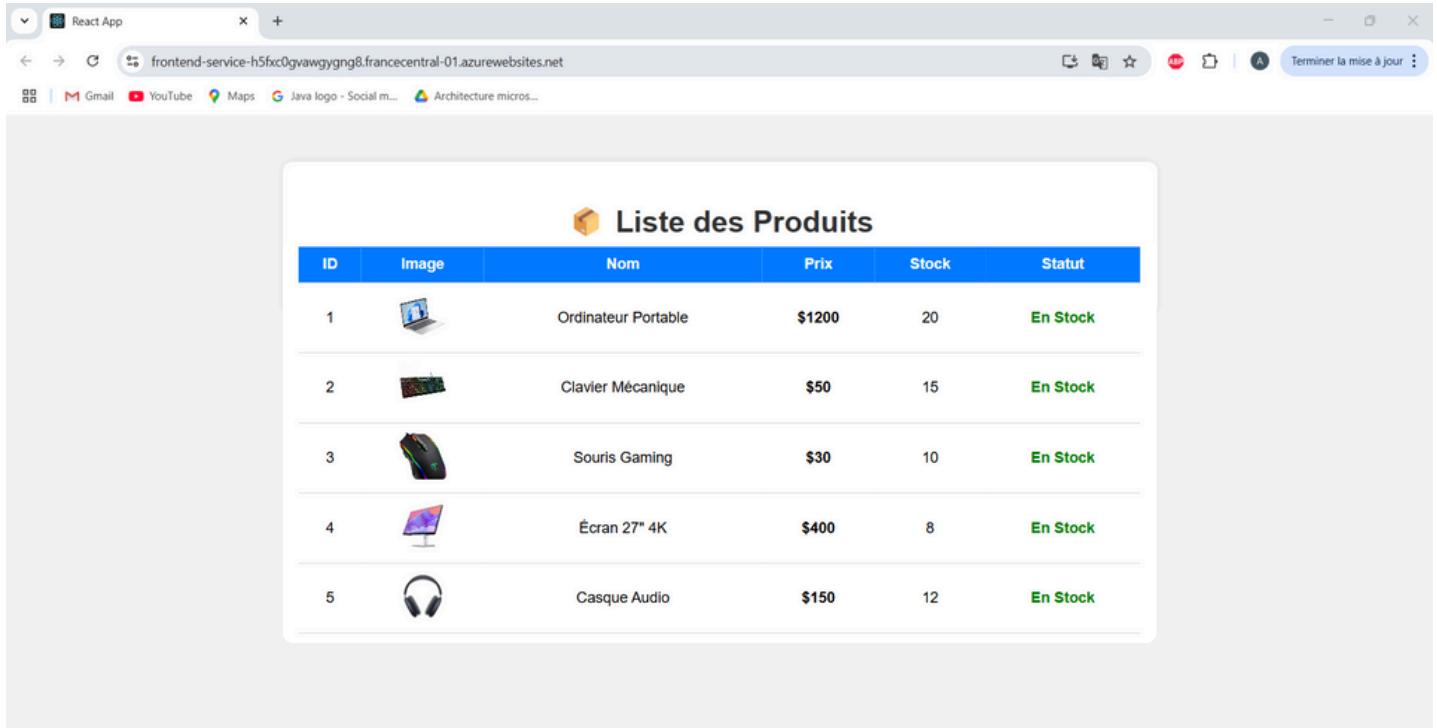
Vue d'ensemble

Bases

| | | | |
|----------------------------|--|-----------------------------|--|
| Groupe de ressources (...) | : Microservices | Domaine par défaut | : frontend-service-h5fc0gwygng8.francecentral-01.azurewebsites.net |
| État | : Exécution | Plan App Service | : ASP-Microservices-8a98 (B1: 1) |
| Emplacement (déplacer) | : France Central | Système d'exploitation | : Linux |
| Abonnement (déplacer) | : Azure for Students | Vérification de l'intégrité | : Non configuré |
| ID d'abonnement | : 79a314de-af5d-408c-9823-a4cc2e4d75e6 | | |
| Étiquettes (modifier) | : Ajouter des étiquettes | | |

3. Déploiement sur Azure

Nous avons cliqué sur ce lien, et le contenu du frontend-service s'est affiché :



Et de la même manière, nous avons suivi ces étapes pour app1-service et app2-service.

Conclusion

Nous avons réussi à déployer nos microservices sur Azure en utilisant Azure Container Registry (ACR) et Azure App Services. Ce processus nous a permis de stocker et gérer efficacement nos images Docker, de configurer les services nécessaires, et de déployer nos applications de manière fluide. En utilisant ces outils, nous avons assuré une intégration et un déploiement continu (CI/CD), garantissant ainsi la scalabilité et la disponibilité de nos services dans le cloud.

4. Automatisation CI/CD

Dans cette section, nous détaillons la mise en place du pipeline d'intégration et de déploiement continu (CI/CD) via GitHub Actions et Docker Hub, permettant d'automatiser les tests, la construction et le déploiement de nos services.

1. Objectifs du pipeline CI/CD pour notre projet:

L'automatisation CI/CD nous permet de:

- Automatiser les tests unitaires et d'intégration pour garantir la fiabilité des services.
- Générer et pousser automatiquement les images Docker de App1 (Service Produits), App2 (Service Magasin) et Frontend (Interface utilisateur React) vers Docker Hub après validation.
- Déployer automatiquement les conteneurs sur Azure App Service après chaque mise à jour du code dans GitHub.

2. Configuration du pipeline CI/CD:

Nous avons mis en place deux workflows YAML :

- Un workflow CI (ci.yaml) pour l'intégration continue (tests et build).
- Un workflow CD (cd.yaml) pour le déploiement continu sur Azure.

3. Intégration continue (CI):

Le fichier ci.yaml est déclenché à chaque push sur la branche main et exécute les étapes suivantes :

1. Cloner le repository contenant App1, App2 et Frontend
2. Installer les dépendances et exécuter les tests unitaires et d'intégration
3. Construire les images Docker de chaque service sans les pousser

```
3 < on:
4   < push:
5     < branches:
6       | - main
7     < pull_request:
8       < branches:
9         | - main
10
11  < jobs:
12    < build-and-test:
13      < runs-on: ubuntu-latest
14
15    < steps:
16      < - name: Checkout code
17        < uses: actions/checkout@v3
18
19      < - name: Build Docker images
20        < run:
21          | docker build -t randomzz/devopsa-app1:latest ./app1
22          | docker build -t randomzz/devopsa-app2:latest ./app2
23          | docker build -t randomzz/devopsa-frontend:latest ./frontend
24
25      < - name: Run Tests for App1
26        < run: docker run --rm randomzz/devopsa-app1:latest pytest tests/
27
28      < - name: Run Tests for App2
29        < run: docker run --rm randomzz/devopsa-app2:latest pytest tests/
30
31      < - name: Run Tests for Frontend
32        < run: docker run --rm randomzz/devopsa-frontend:latest npm test
33
```

4. Automatisation CI/CD

4. Intégration continue (CD):

Le fichier cd.yml est déclenché après la réussite du pipeline CI. Il effectue :

- 1.Connexion à Docker Hub pour authentifier le push des images.
- 2.Build et push des images Docker de App1, App2 et Frontend vers Docker Hub.
- 3.Déploiement sur Azure en redémarrant les services pour appliquer la nouvelle version.

```
1  name: Continuous Deployment (CD)
2
3  on:
4    workflow_run:
5      workflows: ["Continuous Integration (CI)"]
6      types:
7        - completed
8
9  jobs:
10   deploy:
11     runs-on: ubuntu-latest
12     if: ${{ github.event.workflow_run.conclusion == 'success' }}
13
14   steps:
15     - name: Checkout repository
16       uses: actions/checkout@v3
17
18     - name: Log in to Docker Hub
19       uses: docker/login-action@v2
20       with:
21         username: ${{ secrets.DOCKER_USERNAME }}
22         password: ${{ secrets.DOCKER_PASSWORD }}
23
24     - name: Push Docker images
25       run:
26         docker push randamzz/devopsa-app1:latest
27         docker push randamzz/devopsa-app2:latest
28         docker push randamzz/devopsa-frontend:latest
29
30     - name: Deploy to Server (Example: AWS EC2)
31       env:
32         SSH_PRIVATE_KEY: ${{ secrets.SSH_PRIVATE_KEY }}
33         SERVER_IP: ${{ secrets.SERVER_IP }}
34         USER: ${{ secrets.USER }}
35       run:
36         echo "$SSH_PRIVATE_KEY" > private_key.pem
37         chmod 600 private_key.pem
```

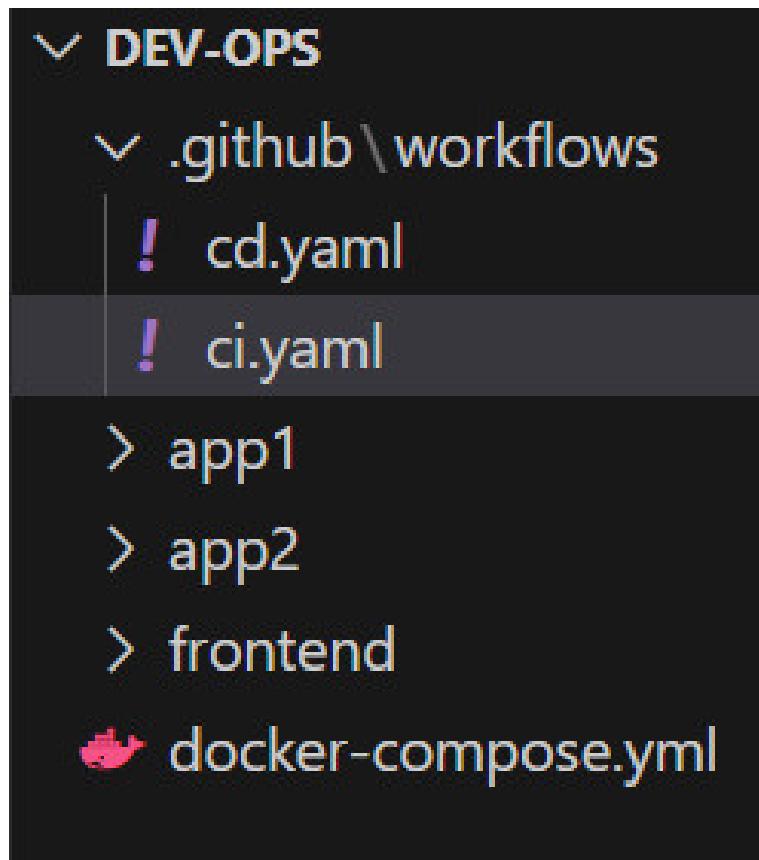
Restricted Mode ⌂ 0 ▲ 0

```
SERVER_IP: ${{ secrets.SERVER_IP }}
USER: ${{ secrets.USER }}
run:
  echo "$SSH_PRIVATE_KEY" > private_key.pem
  chmod 600 private_key.pem
  ssh -o StrictHostKeyChecking=no -i private_key.pem $USER@$SERVER_IP << 'EOF'
    docker pull randamzz/devopsa-app1:latest
    docker pull randamzz/devopsa-app2:latest
    docker pull randamzz/devopsa-frontend:latest
    docker-compose up -d
EOF
```

4. Automatisation CI/CD

5. Ajout des fichiers CI/CD .YAML:

Dans le cadre de l'amélioration de notre processus de développement et de déploiement, nous avons intégré les fichiers de pipelines d'intégration et de déploiement continus (CI/CD) au sein de l'arborescence du projet. Ces fichiers, ci.yaml et cd.yaml, sont situés dans le répertoire .github/workflows et permettent d'automatiser respectivement les tests et la validation du code (CI) ainsi que le déploiement des différentes applications (CD). Cette mise en place garantit une meilleure fiabilité du code, une détection précoce des erreurs et un déploiement plus fluide, optimisant ainsi notre workflow de développement.



4. Automatisation CI/CD

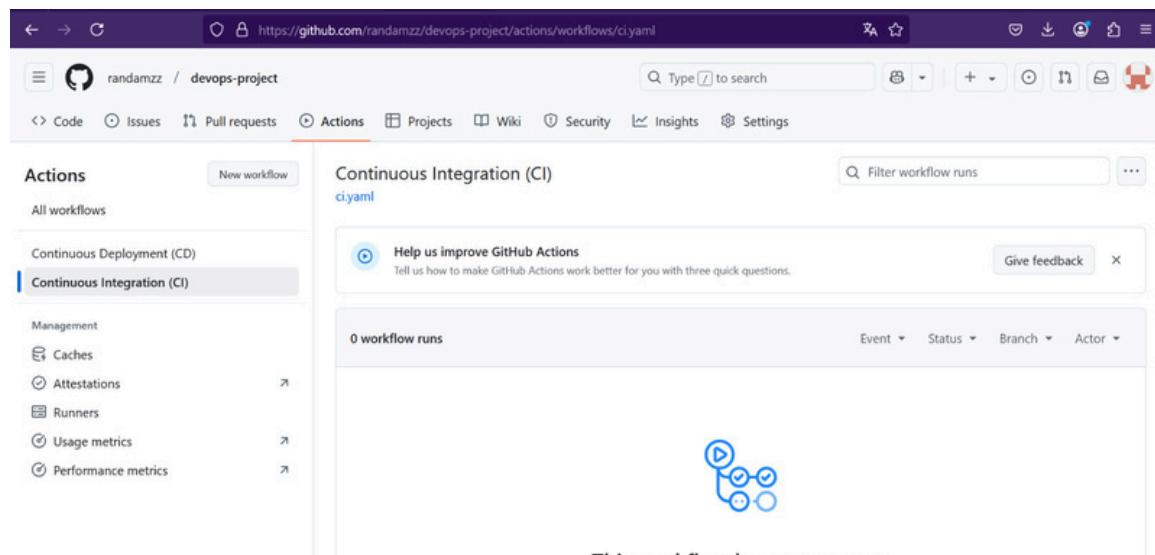
6. Ajout des Secrets GitHub pour l'Authentification Docker:

Afin de sécuriser les informations sensibles nécessaires à l'automatisation du projet, nous avons ajouté des secrets de dépôt dans les paramètres du repository. Ces secrets incluent DOCKERHUB_USERNAME et DOCKERHUB_PASSWORD, qui sont utilisés pour authentifier notre pipeline CI/CD auprès de Docker Hub. Cette configuration permet d'automatiser la connexion, la création et le push des images Docker en toute sécurité, sans exposer les identifiants dans le code. L'utilisation des GitHub Secrets renforce ainsi la sécurité et la confidentialité des informations sensibles tout en garantissant un déploiement fluide et sécurisé.

| Repository secrets | | New repository secret |
|----------------------|--------------|---|
| Name | Last updated | |
| 🔒 DOCKERHUB_PASSWORD | now |   |
| 🔒 DOCKERHUB_USERNAME | now |   |

7. Activation du Workflow d'Intégration Continue (CI) sur GitHub Actions:

Afin de sécuriser les informations sensibles nécessaires à l'automatisation du projet, nous avons ajouté des secrets de dépôt dans les paramètres du repository. Ces secrets incluent DOCKERHUB_USERNAME et DOCKERHUB_PASSWORD, qui sont utilisés pour authentifier notre pipeline CI/CD auprès de Docker Hub. Cette configuration permet d'automatiser la connexion, la création et le push des images Docker en toute sécurité, sans exposer les identifiants dans le code. L'utilisation des GitHub Secrets renforce ainsi la sécurité et la confidentialité des informations sensibles tout en garantissant un déploiement fluide et sécurisé.

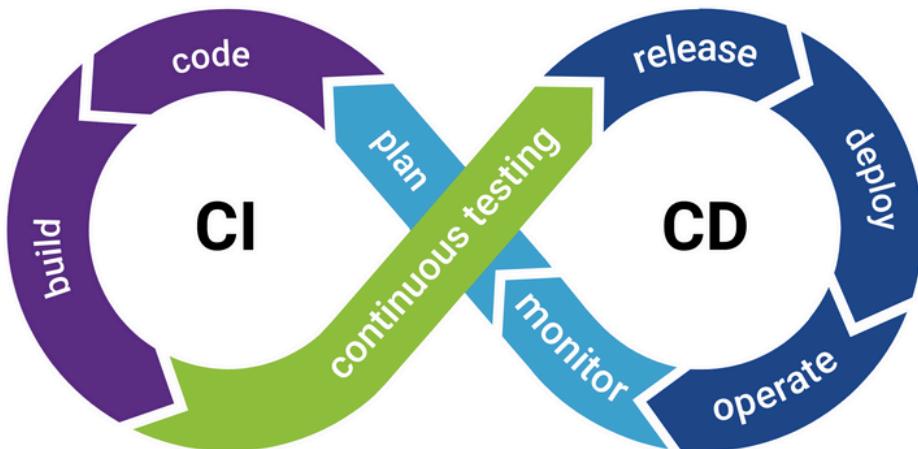


The screenshot shows the GitHub Actions interface for a repository named "devops-project". The "Actions" tab is selected. On the left, there's a sidebar with sections like "Actions", "Continuous Deployment (CD)", and "Management". Under "Management", "Continuous Integration (CI)" is selected. The main area displays the "Continuous Integration (CI) ci.yaml" workflow. It includes a "Help us improve GitHub Actions" card and a table for "0 workflow runs". A message at the bottom states "This workflow has no runs yet." There are also various navigation and search tools at the top and right side of the page.

4. Automatisation CI/CD

8. Conclusion

L'intégration des workflows CI/CD dans GitHub Actions, associée à l'utilisation des secrets pour sécuriser les informations sensibles, représente une avancée majeure dans l'automatisation et la fiabilité du projet. Grâce à ces améliorations, nous garantissons une validation continue du code, un déploiement fluide et une meilleure gestion des accès aux services externes comme Docker Hub. Cette approche optimise non seulement la productivité de l'équipe, mais aussi la qualité et la sécurité du projet, en assurant une livraison rapide et sans erreur des nouvelles fonctionnalités.



5. Collaboration et Gestion de Projet

Dans le cadre de notre projet DevOps, nous avons utilisé Trello et GitHub pour assurer une collaboration efficace et une gestion optimale des tâches. Ces outils nous ont permis d'appliquer les principes fondamentaux de collaboration et de partage dans la gestion du projet.

1. Utilisation de Trello

Trello a été utilisé pour définir et organiser les grandes phases de réalisation du projet, notamment :

- Développement des microservices
- Conteneurisation avec Docker
- Déploiement sur Azure
- Automatisation CI/CD

Notre tableau Trello était structuré en plusieurs colonnes correspondant aux différents statuts des tâches :

| Statut | Rôle |
|----------|--|
| À faire | Contient les tâches planifiées qui n'ont pas encore été commencées. |
| En cours | Regroupe les tâches actuellement en cours de réalisation. |
| Bloqué | Liste les tâches qui rencontrent des obstacles nécessitant une résolution avant de continuer. |
| Revue | Comprend les tâches terminées mais en attente de validation ou de correction après vérification. |
| Terminé | Inclut les tâches finalisées et validées. |

Chaque tâche a été assignée à un membre de l'équipe, garantissant une répartition équilibrée du travail et un suivi précis de l'avancement du projet.

La figure suivante représente notre tableau Trello .

5. Collaboration et Gestion de Projet

The screenshot shows a Trello board titled "DevOps Project". The board is organized into five columns: "À faire", "En cours", "Bloqué", "Revue", and "Terminé". Each column contains cards representing tasks. The "À faire" column has one card: "Mesures de Performance" (0/2). The "En cours" column has two cards: "Déploiement sur Azure" (0/4) and "Automatisation CI/CD" (0/3). The "Bloqué" column has one card: "+ Ajouter une carte". The "Revue" column has one card: "Conteneurisation avec Docker" (4/4). The "Terminé" column has one card: "Développen microservice" (4/4). The top navigation bar includes links for "Espaces de travail", "Récent", "Favoris", "Plus", and a search bar.

2. Gestion des sous-tâches

Pour chaque grande tâche, nous avons défini des sous-tâches afin d'assurer une meilleure organisation et un suivi détaillé.

Par exemple, dans la phase Conteneurisation avec Docker, nous avons détaillé les sous-tâches suivantes :

- Écrire un Dockerfile pour chaque microservice.
- Créer un fichier Docker Compose pour lier les services.
- Tester le build et l'exécution locale des conteneurs.
- Vérifier la bonne communication entre les conteneurs.

La figure suivante représente l'organisation des tâches dans le projet.

This screenshot shows the details of a task titled "Conteneurisation avec Docker" in a Trello board. The task is labeled as completed ("TERMINÉ"). The interface includes tabs for "Membres" (RM), "Étiquettes" (Fait, Bloqué, En cours), "Notifications" (Suivie), and a sidebar with options like "Quitter", "Membres", "Étiquettes", "Checklist", "Dates", "Pièce jointe", "Emplacement", "Image de couverture", and "Champs personnalisés". The "Sous Taches" section lists four sub-tasks: "Écrire un Dockerfile pour chaque microservice" (checked), "Créer un docker-compose.yml pour lier les services" (checked), "Tester le build des images Docker et l'exécution locale" (checked), and "Vérifier la bonne communication entre les conteneurs" (unchecked).

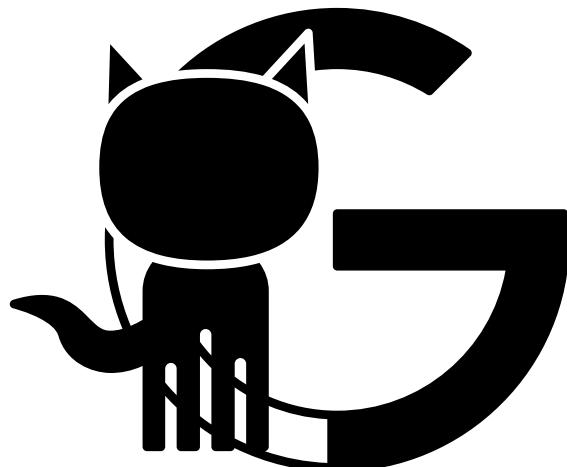
5. Collaboration et Gestion de Projet

3. Utilisation de GitHub:

GitHub a été utilisé pour :

- Héberger le code source et assurer le versioning.
- Gérer les branches pour chaque nouvelle fonctionnalité ou correction de bug.
- Effectuer des pull requests pour examiner et fusionner le code.
- Mettre en place des actions CI/CD pour l'automatisation des déploiements.

Grâce à cette organisation, nous avons pu optimiser la productivité de l'équipe et assurer un développement efficace et structuré en appliquant les principes DevOps.



Conclusion

En adoptant Trello pour la gestion des tâches et GitHub pour le versioning et l'automatisation, nous avons mis en place une organisation efficace favorisant la collaboration et la transparence au sein de l'équipe. La structuration des tâches en grandes phases et sous-tâches a permis un suivi précis de l'avancement du projet, tandis que l'utilisation des branches et des pull requests sur GitHub a garanti un développement structuré et sécurisé. Cette approche nous a permis d'optimiser notre flux de travail et d'appliquer les principes DevOps pour une meilleure intégration et livraison continue.

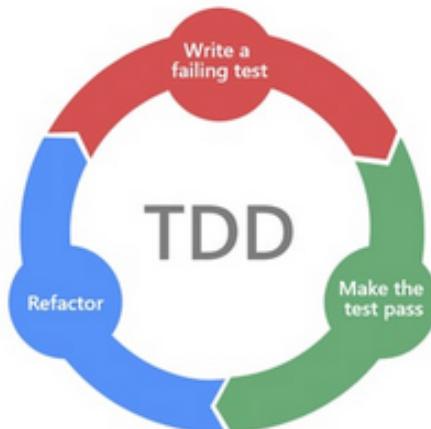
6. Tests et Mesures de Performance

1. Les Tests TDD (Jtest)

Partie 1 : Test-Driven Development (TDD)

Pourquoi utiliser TDD ?

- *Fiabilité : chaque fonctionnalité est testée dès le départ.*
- *Meilleure conception : on réfléchit à ce que doit faire le code avant de l'écrire.*
- *Moins de bugs : les erreurs sont détectées immédiatement, ce qui réduit les régressions.*
- *Facilité de maintenance : le code est plus propre et plus modulaire.*



Partie 2 : Jest et les tests en Node.js

Jest est un framework de test pour JavaScript, principalement utilisé pour les tests unitaires et d'intégration. Il est rapide, simple à configurer et offre des fonctionnalités avancées comme :

- *Tests asynchrones (utile pour les API)*
- *Mocking (simulation de dépendances)*
- *Coverage reports (analyse de la couverture des tests)*

Dans notre cas, nous utilisons Jest avec Supertest, une librairie qui permet de tester facilement les API Express.



6. Tests et Mesures de Performance

Partie 3 : Exemples de tests API avec Jest et Supertest

- **Test de récupération des produits :** Ce test vérifie que notre API renvoie bien une liste de produits et que le statut HTTP est 200 OK.

```
describe('Tests API Produits', () => {
  test('GET /produits - Doit retourner la liste des produits', async () => {
    const response = await request(app).get('/produits');

    expect(response.status).toBe(200);
    expect(response.body).toBeInstanceOf(Array);
    expect(response.body.length).toBeGreaterThan(0);
  });
});
```

- **Test d'ajout d'un produit :** Ce test simule l'ajout d'un produit et vérifie si l'API répond correctement avec un nouvel ID.

```
test('POST /produits - Doit ajouter un produit', async () => {
  const newProduct = {
    nom: 'Test Produit',
    prix: 99,
    stock: 5,
    image: '/images/test.jpg'
  };

  const response = await request(app).post('/produits').send(newProduct);

  expect(response.status).toBe(200);
  expect(response.body).toHaveProperty('id');
  expect(response.body.nom).toBe(newProduct.nom);
});
```

- **Test de validation des données :** Ce test vérifie que l'API refuse un produit si une donnée obligatoire est manquante (ici, le nom)

```
test('POST /produits - Doit refuser un produit sans nom', async () => {
  const response = await request(app).post('/produits').send({
    prix: 99,
    stock: 5,
    image: '/images/test.jpg'
  });

  expect(response.status).toBe(400);
  expect(response.body.message).toBe('Nom, prix, stock et image requis');
```

6. Tests et Mesures de Performance

Partie 4 : Analyse des résultats des tests

Première exécution des tests (Échec)

les trois tests ont échoué :

- GET /produits → L'API ne rentrait pas la liste attendue.
- POST /produits (ajout) → L'ajout de produit ne fonctionnait pas.
- POST /produits (validation) → Même une requête invalide passait, ce qui était un problème.

```
Debugger attached.  
FAIL __tests__/produits.test.js  
Tests API Produits  
  ✗ GET /produits - Doit retourner la liste des produits (38 ms)  
  ✗ POST /produits - Doit ajouter un produit (56 ms)  
  ✗ POST /produits - Doit refuser un produit sans nom (6 ms)
```

Correction et deuxième exécution des tests (Réussite)

Après avoir coder notre functions , nous avons relancé les tests et obtenu un succès total (PASS en vert) :

- GET /produits → L'API renvoie bien la liste des produits.
- POST /produits (ajout) → L'API accepte et retourne le produit ajouté.
- POST /produits (validation) → L'API rejette correctement les données invalides.

```
PASS __tests__/produits.test.js  
Tests API Produits  
  ✓ GET /produits - Doit retourner la liste des produits (147 ms)  
  ✓ POST /produits - Doit ajouter un produit (62 ms)  
  ✓ POST /produits - Doit refuser un produit sans nom (13 ms)  
  
Test Suites: 1 passed, 1 total  
Tests:       3 passed, 3 total  
at Server.log (App.js:71:11)
```

6. Tests et Mesures de Performance

1. Mesures de Performance (Prometheus , Grafana)

Partie 1: Prometheus et Grafana

Prometheus est un outil de collecte de métriques qui scrappe des données (comme les requêtes HTTP, erreurs, etc.) depuis l'application.

Grafana, quant à lui, est un outil de visualisation qui permet d'afficher ces données sous forme de graphiques pour suivre les performances en temps réel.



Pourquoi les utiliser ?

- Prometheus collecte les métriques de l'application pour analyser la performance.
- Grafana permet de visualiser ces métriques de manière claire et interactive, ce qui facilite la détection rapide de problèmes comme des erreurs ou des ralentissements.

Ces deux outils ensemble permettent de suivre la performance d'une application, d'identifier rapidement des problèmes et de prendre des mesures pour les résoudre.

Partie 2 : Configuration de Prometheus et Grafana

- **Création de l'endpoint /metrics dans l'application Node.js :** Nous avons ajouté un endpoint /metrics dans notre application Node.js qui expose des informations de performance, comme le nombre de requêtes et d'erreurs. Ces informations sont nécessaires pour que Prometheus puisse les récupérer.

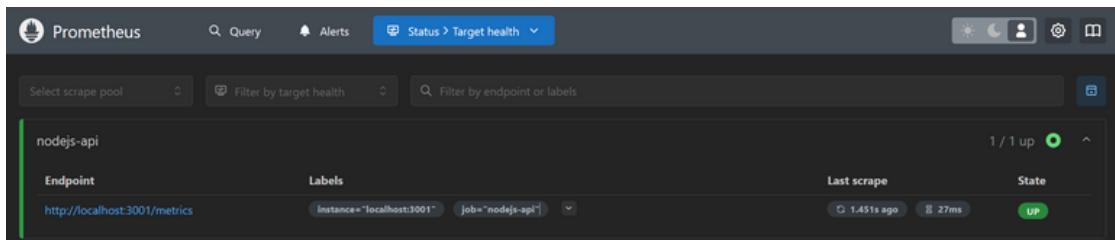
```
PS C:\Users\DELL\Downloads\devopsa\devopsa\app1> node App.js
✓ App is running on http://localhost:3001
Metrics available at http://localhost:3001/metrics
```

```
# HELP process_cpu_user_seconds_total Total user CPU time spent in seconds.
# TYPE process_cpu_user_seconds_total counter
process_cpu_user_seconds_total 0.047

# HELP process_cpu_system_seconds_total Total system CPU time spent in seconds.
# TYPE process_cpu_system_seconds_total counter
process_cpu_system_seconds_total 0.016
```

6. Tests et Mesures de Performance

- **Configuration de Prometheus pour récupérer les métriques :** Nous avons configuré et démarré Prometheus, qui a commencé à collecter les données à partir de notre application Node.js et à les stocker dans sa base de données.



- **Création de tableaux de bord dans Grafana :** Dans Grafana, nous avons créé des tableaux de bord pour visualiser les métriques, comme le nombre de requêtes HTTP, les erreurs 404, le temps de réponse, etc. Ces graphiques permettent de suivre les performances en temps réel.

Partie 3 : Surveillance des performances en temps réel

Collecte des données en temps réel

Les données collectées en temps réel (au cours des 15 dernières minutes) nous fournissent des informations essentielles sur la performance de notre application. Grâce à Prometheus, nous pouvons surveiller des métriques clés telles que :

- Le nombre total de requêtes HTTP (comme celles effectuées sur /favicon.ico et /metrics).
- Les codes de statut HTTP (200, 404, etc.), qui nous indiquent si des erreurs se produisent lors des requêtes.
- La répartition des requêtes ce qui nous permet de détecter des problèmes spécifiques, comme des erreurs récurrentes sur certaines ressources.
- Le temps de réponse pour chaque requête, ce qui nous aide à évaluer la réactivité de l'application.

Ces données sont affichées en temps réel dans Grafana sous forme de graphiques interactifs, permettant à l'équipe de réagir rapidement aux anomalies, telles que des erreurs élevées ou un temps de réponse plus long que prévu.



Conclusion

Ce projet nous a permis de concevoir une architecture logicielle optimisée et modulaire, facilitant la scalabilité et la maintenance de l'application. En structurant nos services en microservices, nous avons assuré une meilleure répartition des responsabilités et une gestion efficace des différentes fonctionnalités. Cette approche a posé les bases solides nécessaires à la conteneurisation et à l'automatisation du déploiement.

La conteneurisation avec Docker nous a offert un environnement homogène, garantissant la portabilité et la reproductibilité des services sur différentes plateformes. En intégrant Docker Compose, nous avons pu orchestrer et gérer l'ensemble des conteneurs de manière fluide, assurant ainsi une interconnexion efficace entre les composants de l'application.

Le déploiement sur Azure et l'automatisation via CI/CD ont permis d'accélérer et de fiabiliser le cycle de livraison du projet. Grâce aux pipelines de déploiement, nous avons réduit les interventions manuelles, minimisé les erreurs et amélioré la disponibilité du système. Ce processus nous a offert un gain de temps considérable tout en assurant une mise en production plus rapide et sécurisée.

Enfin, l'utilisation d'outils collaboratifs tels que Trello et GitHub a favorisé une gestion de projet efficace, garantissant une répartition claire des tâches et un suivi précis de l'évolution du développement. Les tests et les mesures de performance ont également joué un rôle clé en nous permettant d'optimiser notre application et d'assurer sa fiabilité. Ce projet nous a ainsi offert une expérience complète et enrichissante des pratiques DevOps.