CI/CD avec GitHub Actions

Introduction

Le but de ce TP va être d'automatiser le cycle d'intégration de l'application conteneurisé dans le TP précédent, afin de rendre accessible la release, dans ce cas une image Docker accessible afin d'être déployé lors d'un prochain TP dans un environnement de production.

Pour cela, vous allez utilisez GitHub Actions comme étudié dans le cours.

Prérequis

Ce TP nécessite d'avoir :

- Un environnement Linux/Unix. Si vous utilisez Windows, assurez-vous d'avoir WSL2 d'installé et configuré.
- Docker installé dans un environnement Linux/Unix. Si vous utilisez Windows, assurez-vous d'avoir WSL2 d'installé et configuré pour fonctionner avec Docker Desktop.
- Git installé dans l'environnement Linux/Unix.
- Un compte GitHub. Si vous n'en avez pas, il faudra vous en créer un sur GitHub.

Pour la suite de ce TP, vous utiliserez exclusivement votre environnement Linux/Unix préalablement configuré. **Veillez à ne pas utiliser votre environnement Windows ainsi que Powershell**

Rappel sur Git

Afin de réaliser ce TP il est nécessaire au préalable d'avoir une connaissance du fonctionnement de Git. Voici un rappel sur l'utilisation de Git :

- `git clone` permet de cloner un repository distant sur votre environnement local.
- `git add .` permet d'ajouter les fichiers modifiés dans l'index.
- `git commit -m "message"` permet persister les fichiers situés dans l'index dans le repository.
- `git push` permet de pousser les changement du repository local vers le repository distant.
- `git pull` permet de mettre à jour les changement du repository distant dans le repository local.

Fork du repository Git.

Afin de faciliter la mise en place de ce TP, le plus simple consiste à fork ce repository Git. Pour cela, connectez vous avec votre compte sur GitHub, rendez vous sur le repository de ce TP et cliquez sur **"Fork"** en haut à droite de l'interface de GitHub. Sélectionnez votre espace personnel comme destination.

Vous pouvez ainsi cloner le repository forké et travailler à partir de ce répo là.

Votre premier workflow GitHub

Afin de vous familiariser avec GitHub Actions, créer un dossier `.github/workflows` à la racine du repo Git:

```
mkdir -p .github/workflows
```

Créer ensuite le fichier `my-first-workflow.yaml` dans ce dossier avec le contenu suivant dedans :

```
name: GitHub Actions Demo
run-name: ${{ github.actor }} is testing out GitHub Actions 🚀
on: [push]
jobs:
  Explore-GitHub-Actions:
    runs-on: ubuntu-latest
    steps:
      - run: echo "🎉 The job was automatically triggered by a ${{
github.event_name }} event."
      - run: echo "🐧 This job is now running on a ${{ runner.os }} server
hosted by GitHub!"
      - run: echo "🔎 The name of your branch is ${{ github.ref }} and
your repository is ${{ github.repository }}."
      - name: Check out repository code
        uses: actions/checkout@v3

      - run: echo "💡 The ${{ github.repository }} repository has been
cloned to the runner."
      - run: echo "🖥️ The workflow is now ready to test your code on the
runner."
      - name: List files in the repository
        run: |
          ls ${{ github.workspace }}
      - run: echo "🍏 This job's status is ${{ job.status }}."
```

A partir du cours, essayez de comprendre ce que va permettre de réaliser ce workflow GitHub.

Commitez et pousser ce nouveau fichier dans votre repo Github avec les commandes suivantes :

```
git add .
git commit -a -m"Add a new workflow"
git push
```

Rendez vous ensuite sur votre repo GitHub qui devrait avoir l'adresse suivante : `https://github.com/<YOUR_GITHUB_USER>/TP3-Github-Actions` et allez dans l'onglet `Actions`. Un nouveau workflow a normalement du se lancer. Cliquez dessus et inspectez le. N'hésitez pas à dérouler les logs de chaque commande dans le workflow pour bien comprendre le fonctionnement. **Il est primordial de comprendre le fonctionnement afin de réaliser la suite du TP**

Essayez maintenant de comprendre le lien qu'il y a entre le contenu du fichier `my-first-workflow.yaml` et le workflow sur GitHub.

Automatisation de l'intégration continue de l'API.

Dans la suite de ce TP, vous allez créer un nouveau workflow GitHub qui va permettre d'automatiser l'ensemble des étapes de l'intégration continue de l'API.

Voici les étapes de l'intégration continue de l'API :

- Installation des dépendances : `yarn install`
- Exécution des tests unitaires : `yarn test`
- Exécution des tests d'intégration : `yarn test:e2e`
- Build de l'application : `yarn build`
- Construction de l'image Docker.
- Partage de l'image Docker sur le DockerHub.

1) Automatisation des tests unitaires

Créer un nouveau workflow en vous inspirant de celui précédemment créé, qui va permettre d'exécuter les tests unitaires de l'API.

Afin d'exécuter les tests unitaires, il est nécessaire d'avoir NodeJS d'installé ainsi que les dépendances. Pensez-y !!!

N'oubliez pas le code de l'API est localisée dans le dossier `api`, c'est important car par défaut les scripts du workflow sont exécutés depuis le répertoire racine. Afin de palier ce problème, vous pouvez ajouter cette configuration dans le job afin de modifier le répertoire par défaut d'exécution des scripts :

```
jobs:
  build-and-test:
    name: Run unit tests
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./api
```

Voici un lien de la documentation de GitHub qui pourra vous aider:

- Tester et build une application NodeJS

2) Construction et partage de l'image Docker

Ajouter une étape qui va permettre de construire l'image Docker de l'API et de la partager sur le DockerHub.

Pour cela il sera nécessaire de :

- Créer un compte sur le DockerHub
- Générer un `token` à partir de votre compte DockerHub qui sera utilisé pour vous authentifier sur le DockerHub et pousser l'image créée depuis GitHub Actions. Suivez cette documentation pour créer votre token.

Pensez à utiliser une valeur unique (par exemple le `SHA du commit`) dans le tag de l'image Docker, afin de rendre chaque image unique. Pour chaque job, des informations de contextes sont générées et accessibles à partir du workflow. Il faudra utiliser ce contexte afin de récupérer le `SHA du commit`.

Il vous faudra également configurer un `secret` afin de ne ma exposer en clair la valeur du token dans le workflow.

Utilisez la documentation de GitHub afin de vous aider, en suivant ces liens:

- Créer un secret dans GitHub Actions.
- Construire et publier une image Docker avec GitHub Actions.
- Utiliser le contexte pour récupérer la valeur du SHA du commit.

Une fois réalisé, assurez que l'image a été poussée sur le DockerHub. Assurez vous également que celle-ci est fonctionnelle en mettant à jour le fichier `docker-compose.yaml` avec le nom de votre nouvelle image Docker et en lançant la commande `docker compose up`.

Assurez que l'API est fonctionnelle en interagissant avec celle-ci :

```
# Ajouter un utilisateur
curl --location --request POST 'http://localhost:3000/users' \
--header 'Content-Type: application/json' \
--data-raw '{"email":"alexis.bel@ynov.com", "firstName": "Alexis",
"lastName": "Bel"}'

# Lister les utilisateurs
curl 'http://localhost:3000/users'
```

3) Ajout des tests d'intégration

Ajoutez une nouvelle étape dans le workflow qui va exécuter les tests d'intégration. Les tests d'intégration, contrairement aux tests unitaires, permettent de tester l'intégration avec le reste du système, **à savoir MongoDB**. Durant l'exécution de ces tests, l'API va donc devoir interagir avec MongoDB.

Il vous faudra :

- Configurer un serveur MongoDB dans le workflow : Configurer des services externes pour les tests d'intégration
- Passer la variable d'environnement `MONGODB_URI` nécessaire pour la connection à MongoDB pendant l'exécution des tests d'intégration (qui se lancent avec la commande `yarn test:e2e`) : Configurer des variables d'environnement dans le workflow

Si les tests échouent, l'image Docker ne doit pas être poussée sur le DockerHub.

4) Mise ne cache des dépendances

Une des étapes les plus longues dans le workflow est le téléchargement des dépendances avec `yarn install`.

Les dépendances ne change pas à chaque nouveau build, par conséquent il serait intéressant de les garder en cache lorsque les dépendances ne changent pas.

Trouvez un moyen de mettre en place ce cache en suivant ces liens :

- Mise en cache des dépendances dans GitHub Actions
- Mise en cache des dépendances NodeJS avec Yarn

Bonus : Intégrer SonarCloud dans le workflow GitHub.

SonarCloud est un outils de scan de code qui permet de vérifier la qualité du code.

Ajouter une étape qui permettra d'intégrer un scan du code avec SonarCloud en suivant cette documentation.