CI/CD avec GitHub Actions

Yoan Ancelly Lyazid Bahajjoub Jamil Abdel Hamid

Automatisation de l'intégration continue passing

> TP pour présenter une CI/CD avec GitHub Actions

Sommaire

Automatisation de l'intégration continue de l'API

Les étapes d'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

1.1) Installation de Node.js

Il est nécessaire d'avoir NodeJS pour exécuter les tests unitaires en local.

On peut le télécharger en utilisant nvm (Node Version Manager) :

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.2/install.sh |
bash
```

```
wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.2/install.sh |
bash
```

Puis on installe la version lts de NodeJS :

```
nvm install --lts
```

Et enfin :

```
nvm use --lts
```

1.2) Installation des dépendances

L'application se trouve dans le dossier `api`.

On se place dans ce dossier :

```
cd api
```

On installe les dépendances de l'application avec la commande :

```
yarn install
```

1.3) Exécution des tests unitaires

On exécute les tests unitaires avec la commande :

```
yarn test
```

1.4) Création du fichier de configuration de GitHub Actions

Dans le répertoire racine, création du dossier `.github/workflows` et du fichier `automate.yaml` à l'intérieur.

```
mkdir .github/workflows
touch .github/workflows/automate.yaml
```

```yaml
# automate.yaml

name: Automatisation de l'intégration continue
on: [push]
jobs:
  build-and-test:
    name: Run unit tests
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./api
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
      - name: Install dependencies
        run: yarn install
      - name: Run unit tests
        run: yarn test
```

- `name` : Nom du workflow.
- `on` : Déclencheur du workflow.
- `jobs` : Liste des jobs.
- `build-and-test` : Nom général du job.
- `name` : Nom plus explicite du job.
- `runs-on` : Plateforme d'exécution du job.
- `defaults` : Configuration par défaut du job.
- `run` : Configuration par défaut des étapes du job.
- `working-directory` : Répertoire de travail par défaut des étapes du job.
- `steps` : Liste des étapes du job.
- `uses` : Action à exécuter.
- `actions/checkout@v3` : Action qui permet de mettre le repo du code dans un workplace ($GITHUB_WORKPLACE) pour permettre au workflow d'y acceder.
- `actions/setup-node@v3` : Action permettant d'installer Node.js.
- `name` : Nom de l'étape.
- `run` : Commande à exécuter.
- `yarn install` : Installation des dépendances.
- `yarn test` : Exécution des tests unitaires.

Il est important de mettre des tirets pour le nom des étapes pour suivre une logique de telle sorte que chaque étape a son fonctionnement et de bien séparer les étapes.

Lorsque l'on push sur Github on remarque sur Github le nouveau workflow avec toutes les caractéristiques comme le nom du workflow le nom des jobs, des étapes du job et processus détaillés de chaque étape dans le fichier automate.yaml qu'on a préalablement rempli.

2) Construction et partage de l'image Docker

D'abord il faut créer un compte DockerHub.

Ensuite il faut générer un token d'accès,

Maintenant nous ajoutons les secrets suivants dans les paramètres du dépôt GitHub :

`Settings > Secrets > New repository secret`

- TP3_DOCKERHUB_SECRET : Token d'accès DockerHub.
- TP3_DOCKERHUB_USERNAME : Nom d'utilisateur DockerHub.

```yaml
# Suite du fichier automate.yaml

push-to-docker:
  name: Push on Docker Hub
  runs-on: ubuntu-latest
  defaults:
    run:
      working-directory: ./api
  steps:
    - name: Check out the repo
      uses: actions/checkout@v3

    - name: Install dependencies
      run: yarn install

    - name: Build app
      run: yarn build

    - name: Log in to Docker Hub
      uses: docker/login-action@v2
      with:
        username: ${{ secrets.TP3_DOCKERHUB_USERNAME }}
        password: ${{ secrets.TP3_DOCKERHUB_SECRET }}

    - name: Setup Docker Buildx
      uses: docker/setup-buildx-action@v2

    - name: Build and push
      uses: docker/build-push-action@v3
      with:
        context: ./api
        push: true
        tags: yoanc/tp3-github-actions:${{ github.sha }}
```

- `push-to-docker` : Nom du job.
- `docker/login-action@v2` : Action permettant de se connecter à DockerHub.
- `docker/setup-buildx-action@v2` : Action permettant de configurer Docker Buildx.
- `docker/build-push-action@v3` : Action permettant de construire et de partager une image Docker.

On utilise `${{ secrets.TP3_DOCKERHUB_USERNAME }}` et `${{ secrets.TP3_DOCKERHUB_SECRET }}` pour récupérer les secrets.

Il est nécessaire d'installer buildx avec `docker/setup-buildx-action@v2` pour pouvoir construire une image Docker multi-architecture.

Il ne faut pas oublier d'ajouter le `context` dans la configuration de l'action `docker/build-push-action@v3` pour spécifier l'emplacemment du Dockerfile. L'action `docker/build-push-action@v3` ne prend pas en compte le `working-directory` par défaut.

Il faut créer un nouveau répertoire dans Dockerhub pour pouvoir utiliser `push:true` puisqu'il nécessite un répertoire dont on met le nom dans les tags

Il suffit de rajouter `${{ github.sha }}` à la fin de l'image Docker pour que l'image soit unique.

On remarque que dans le répertoire du dockerhub on a une nouvelle image avec un tag que l'on va reporter dans le docker-compose.yaml partie image de l'api.

3) Automatisation des tests d'intégration

```
# Suite du fichier automate.yaml

build-and-test-e2e:
  name: Run integration tests
  runs-on: ubuntu-latest
  defaults:
    run:
      working-directory: ./api
  services:
    mongo:
      image: mongo
      ports:
        - '27017:27017'
      env:
        MONGO_INITDB_ROOT_USERNAME: ${MONGODB_USERNAME}
        MONGO_INITDB_ROOT_PASSWORD: ${MONGODB_PASSWORD}
  steps:
    - name: Check out the repo
      uses: actions/checkout@v3

    - uses: actions/setup-node@v3

    - name: Install dependencies
      run: yarn install

    - name: Run integration Tests
      run: yarn test:e2e
      env:
        MONGODB_URI:
mongodb://${MONGODB_USERNAME}:${MONGODB_PASSWORD}@localhost:27017
```

Les tests d'intégration nécessitent une base de données MongoDB, il faut ajouter un service MongoDB dans le job.

Il faut créer un fichier `.env` pour stocker les variables d'environnement.

```
touch .env
```

Avec les variables suivantes :

```
MONGODB_USERNAME=username
MONGODB_PASSWORD=password
```

Comme les jobs sont exécutés directement sur une `runner machine` il faut spécifier les ports à utiliser pour le mapping. L'hôte est alors `localhost` et le port de la machine est `27017`.

Pour éviter de pousser une image Docker si les tests échouent, il faut ajouter une condition needs dans le fichier `automate.yaml` à l'étape `Push on Docker Hub`.

```yaml
# automate.yaml

push-to-docker:
    name: Push on Docker Hub
    runs-on: ubuntu-latest
    needs: [build-and-test, build-and-test-e2e]
    (...)
```

4) Mise en cache des dépendances

Pour accélérer le processus d'intégration continue, il est possible d'utiliser le cache des dépendances.

Il suffit d'ajouter les étapes suivantes dans le fichier `automate.yaml` lors de l'installation des dépendances.

```yaml
# automate.yaml

- name: Setup NodeJS
    uses: actions/setup-node@v3
      with:
        cache: 'yarn'
        cache-dependency-path: '**/yarn.lock'

- name: Install dependencies
    run: yarn install --immutable
```

- `cache` : Type de cache à utiliser.
- `cache-dependency-path` : Chemin du fichier de dépendances.

Ce dernier est important car le dossier racine est utilisé par défaut et permet d'éviter l'erreur suivante :

```
yarn cache not found
```

On peut ajouter `--immutable` à la commande `yarn install` pour éviter que les dépendances ne soient modifiées dans le fichier `yarn.lock`.