Automatisation de l'intégration continue de l'API

Afin de réaliser l'automatisation, nous partons d'un unique workflow dans lequel on distinguera toutes les étapes à partir d'un name avant chaque action. Basiquement, le fichier commence par le code ci-dessous:

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

Note: les points de suspensions(...) désignent le code tapé précédement

1) Automatisation des tests unitaires

Pour réaliser les tests unitaires, on rajoute **steps** en dessous de **defaults** :

```
name: GitHub Actions Tests Unitaires
run-name: ${{ github.actor }} is testing out GitHub Actions 
on: [push]
jobs:
  build-and-test:
    name: Run unit tests
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./api
    steps:
```

à l'intérieur de **steps**,

* on commence par cloner le repository git actuel

```
...
jobs:
  steps:
    - uses: actions/checkout@v3
```

Toutes les étapes suivantes ont un name à partir de maintenant, le name

* ensuite on choisit la version de node qui doit être installée

```
jobs:
  steps:
    ...
    - name: Use Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '18.x'
```

içi, la version choisit est la version 18

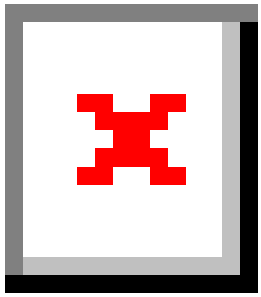
* on installe les dépendences

```
jobs:
  steps:
    ...
    - name: install dependencies
      run: yarn install
```

* on fait les tests unitaires

```
jobs:
  steps:
    ...
    - name: test unitaire
      run: yarn test
```

Lorsqu'on push ces modifications sur git, le workflow se déclenche et on obtient ce résultat qui montre que l'opération a été un succès.



2) Construction et partage de l'image Docker

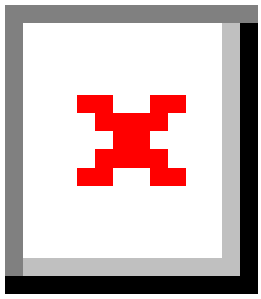
Comme indiqué dans le tp, nous avons commencé par créer un compte sur le Dockerhub et générer un token : il suffit d'aller sur *my profile/edit profile/security/Access Tokens/New Access Tokens* et de générer le token. Puis, pour éviter d'exposer dans le fichier yaml la valeur du token, on crée des secrets comme mentionné dans le tp. Les secrets, sont des variables définies sur git que github peut eploiter dans les workkflow grâce au nom donné à ces secrets. Il suffit de faire dans le repository *settings/security/secrets/Actions/New Repository secret* pour définir un nouveau secret.

* Dans fichier yaml du workflow, on ajoute trois nouvelles étapes la première permet de se connecter à docker, la seconde va chercher des métadonées(nom du tag, nom du label) sur le repository correspondant du docker hub connection à docker

```
jobs:
  steps:
    ...
    - name: connexion à Docker Hub
```

```
    uses: docker/login-action@v2
    with:
      username: ${{ secrets.DOCKER_UTILISATEUR }} #nom du secret enregistré sur github correspondant au nom d'utilisateur utilisé pour se connecter à docker
      password: ${{ secrets.DOCKER_TOKEN }} #nom du secret enregistré sur github correspondant au token génére sur le github
  - name: Extract metadata (tags, labels) for Docker
    id: meta
    uses: docker/metadata-action@98669ae865ea3cffbcbaa878cf57c20bbf1c6c38
    with:
      images: my-docker-hub-namespace/my-docker-hub-repositor #nom donné à l'image
  - name: Build and push Docker image
    uses: docker/build-push-action@ad44023a93711e3deb337508980b4b5e9bcdc5dc
    with:
      context: .
      push: true #on spécifie que l'image doit être poussée sur github si l'image est bien construite
      tags: ${{ steps.meta.outputs.tags }} #le tag extrait du dockerhub dans l'étape précédente
      labels: ${{ steps.meta.outputs.labels }} #les labels extrait du dockerhub dans l'étape précédente
```

Après push,on obtient:



En regardant de plus près, on constate que c'est l'étape du build qui a causé l'erreur: le problème venait du chemin d'accès, le volume crée lors du build dans le dossier api était introuvable lors du push. On l'a corrigé en changeant la valeur de context. Aussi,nous nous sommes rendus compte pas la suite qu'on pouvait se dispenser de l'étape de l'extraction des métadonnée alors nous l'avons enlevé.

```
jobs:
  steps:
    ...
  - name: Build and push Docker image
    uses: docker/build-push-action@ad44023a93711e3deb337508980b4b5e9bcdc5dc
    with:
      context: ./api # le chemin '.' représente la racine du conteneur, l'opération de push ne
```

considère pas le working directory spécifié précedement. Il faut donc remettre le chemin spécifié dans le working directory
      push: true
        tags: idayatnoufou/mon-api:${{ github.sha }} #le nom de l'image et du tag (repository github/nom de l'image:nom du tag) vu que la section meta donnée n'exite plus.

${{ github.sha }} est une variable existant sur github qu'on utilise comme tag pour avoir une image unique à chaque fois.

On obtient encore une erreur au niveau du build, cette fois parce que le dossier *dist* n'existe pas. Ce qui est normal car nous n'avons pas build l'api avant de créer l'image. On rajoute une nouvelle étape **build de l'api** et après push, l'opération succède. le fichier yaml à ce stade ressemble à ça:

```
...
jobs:
  ...
    steps:
      - uses: actions/checkout@v3
      - name: Use Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18.x'
      - name: install dependencies
        run: yarn install
      - name: build de l'api
        run: yarn build
      - name: test unitaire
        run: yarn test
      - name: connexion à Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKER_UTILISATEUR }}
          password: ${{ secrets.DOCKER_TOKEN }}
      - name: Build and push Docker image
        uses: docker/build-push-action@v3
        with:
          context: ./api
          push: true
          tags: idayatnoufou/mon-api:${{ github.sha }}
```

3) Ajout des tests d'intégration

Pour faire les tests d'intégration, nous avons ajouté le service mongo dans le worflow sans créer d'utilisateurs pour éviter les erreurs au début. Puis nous avons configuré la variable d'environnement du conteneur de l'api

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

```
    mongo:
      image: mongo
  steps:
    ...
    - name: test d'intégration
      run: yarn test:e2e
      env:
        MONGODB_URI: "mongodb://mongo:27017/"
```

On obtient une erreur apès push : l'api n'arrive pas se connecter au service mongo. Cela est dû au fait que **la création du service mongo est faite indépendemment du service mon-api**, les deux services ne sont pas sur le même réseau (aucun réseau n'a été crée). Pour que l'api puisse communiquer avec le service mongo, elle doit passer par l'hôte
d'où la modification de la variable d'environnement en changeant **mongo** par **localhost** et en rajoutant les ports au service mongo.

```
...
jobs:
  build-and-test:
    name: Run unit tests
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./api
    services:
      mongo:
        image: mongo
        ports: #modif içi
          - 27017:27017
    steps:
      ...
      - name: test d'intégration
        run: yarn test:e2e
        env:
          MONGODB_URI: "mongodb://localhost:27017/" #modif içi
```

4) Mise ne cache des dépendances

Pour cette partie, on modifie l'étape use node.js car c'est celle qui prend le plu de temps afin de garder en cache ce qui a été installé pour pouvoir les réutiliser.

```
...
jobs:
  ...
    steps:
      ...
      - name: Use Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18.x'
          cache: 'yarn'
          cache-dependency-path: '**/yarn.lock'
      ...
```

Bonus : Ajout de SonarCloud