COMPTE RENDU DES MODIFICATIONS

2024

8 AVRIL

SERRE Loïc

Aix*Marseille université

Table des matières

Introduction

Contexte

Ce projet est un proof of concept (POC) pour la mise en place d'une application multi-plateforme (web et desktop dans mon cas, mais peut être mobile, embarqué, etc…).
Le but de mon application est de permettre à un utilisateur de gérer une cartographie de port commercial.
L'utilisateur pourra alors modifier les emplacements des conteneurs, les déplacer, les supprimer, les ajouter, etc...

Tests d’intégration et unitaires de l’application

Description

Au cours du semestre 6 de la SAÉ, j'ai réalisé une série de tests d'intégration et unitaires pour évaluer la robustesse, la fiabilité et la qualité de l'application développée. Ces tests ont été conçus pour vérifier le bon fonctionnement des différentes fonctionnalités de l'application, en mettant l'accent sur la cohérence de l'interface utilisateur, la précision des fonctionnalités métier et la stabilité du code.

Méthodologie

La réalisation des tests d'intégration et unitaires a suivi une approche méthodique, basée sur les bonnes pratiques de test logiciel. Voici les étapes clés de la méthodologie employée :

1. Analyse des exigences : Avant de commencer les tests, j'ai effectué une analyse approfondie des exigences fonctionnelles et techniques de l'application, en identifiant les cas d'utilisation principaux et les scénarios de test associés.

2. Conception des tests : En fonction des exigences identifiées, j'ai conçu une suite de tests couvrant les différentes parties de l'application, en se concentrant à la fois sur les tests unitaires pour les composants individuels et sur les tests d'intégration pour les interactions entre les composants.

3. Implémentation des tests : Les tests ont été implémentés en utilisant des frameworks de test appropriés, tels que Jest et @testing-library/react pour les tests d'intégration de l'interface utilisateur, et Mocha et Chai pour les tests unitaires des services et des API.

4. Exécution des tests : Les tests ont été exécutés de manière régulière tout au long du processus de développement, permettant de détecter les erreurs et les anomalies dès leur apparition et de les corriger rapidement.

5. Analyse des résultats : Les résultats des tests ont été analysés de manière critique pour identifier les zones à risque et les problèmes potentiels, et pour guider les efforts de correction et d'amélioration du code.

Exemple de test unitaire

Voici un exemple de test unitaire que j'ai développé pour évaluer le bon fonctionnement des opérations CRUD sur les entités « rôles » de l'API REST :

```
describe('RolesSqliteDAO', () => {
    let rolesSqliteDAO;
    let idtestrole;

    beforeEach(() => {
        rolesSqliteDAO = new RolesSqliteDAO();
    });

    describe('getRoles', () => {
        it('should retrieve roles based on the specified filters', async () => {
            const filters = { id: 1 };

            const roles = await rolesSqliteDAO.getRoles(filters);

            expect(roles).to.be.an('array');
            expect(roles).to.have.lengthOf(1);
            expect(roles[0]).to.have.property('role_id', 1);
            expect(roles[0]).to.have.property('name', 'user');
        });
    });

    describe('createRole', () => {
        it('should create a new role', async () => {
            const role = { name: 'test' };

            const newRole = await rolesSqliteDAO.createRole(role);
            idtestrole = newRole.lastID;

            expect(newRole).to.be.an('object');
            expect(newRole).to.have.property('lastID');
        });
    });

    describe('updateRole', () => {
        it('should update a role', async () => {
            const role = { name: 'test2' };

            const updatedRole = await rolesSqliteDAO.updateRole(idtestrole, role);

            expect(updatedRole).to.be.an('object');
            expect(updatedRole).to.have.property('changes');
        });
    });

    describe('deleteRole', () => {
        it('should delete a role', async () => {
            const deletedrole = await rolesSqliteDAO.deleteRole(idtestrole);

            expect(deletedrole).to.be.an('object');
            expect(deletedrole).to.have.property('changes');
        });
    });
});
```

Exemple de test d’intégration

Voici un exemple de test d'intégration que j'ai développé pour évaluer le bon rendu d'une page de création de zone dans l’application React :

```
describe('CreateZoneModal', () => {
    test('renders create zone modal', () => {
        render(
            <Router>
                <CreateZoneModal isOpen={true} onRequestClose={() => {}} />
            </Router>
        );

        // Check if the modal title is rendered
        expect(screen.getByText('Créer une nouvelle zone')).toBeInTheDocument();

        // Check if the form elements are rendered
        expect(screen.getByLabelText('Nom :')).toBeInTheDocument();
        expect(screen.getByLabelText('Coordonnée X :')).toBeInTheDocument();
        expect(screen.getByLabelText('Coordonnée Y :')).toBeInTheDocument();
        expect(screen.getByLabelText('Hauteur :')).toBeInTheDocument();
        expect(screen.getByLabelText('Largeur :')).toBeInTheDocument();
        expect(screen.getByLabelText('Nombre de lignes :')).toBeInTheDocument();
        expect(screen.getByLabelText('Nombre de colonnes :')).toBeInTheDocument();

        // Check if the buttons are rendered
        expect(screen.getByRole('button', { name: 'Créer' })).toBeInTheDocument();
        expect(screen.getByRole('button', { name: 'Annuler' })).toBeInTheDocument();
    });

    test('updates state on input change', () => {
        render(
            <Router>
                <CreateZoneModal isOpen={true} onRequestClose={() => {}} />
            </Router>
        );

        // Change the input values
        fireEvent.change(screen.getByLabelText('Nom :'), { target: { value: 'Test Zone' } });
        fireEvent.change(screen.getByLabelText('Coordonnée X :'), { target: { value: '10' } });
        fireEvent.change(screen.getByLabelText('Coordonnée Y :'), { target: { value: '20' } });
        fireEvent.change(screen.getByLabelText('Hauteur :'), { target: { value: '100' } });
        fireEvent.change(screen.getByLabelText('Largeur :'), { target: { value: '200' } });
        fireEvent.change(screen.getByLabelText('Nombre de lignes :'), { target: { value: '5' } });
        fireEvent.change(screen.getByLabelText('Nombre de colonnes :'), { target: { value: '8' } });

        // Check if the state values are updated
        expect(screen.getByLabelText('Nom :').value).toBe('Test Zone');
        expect(screen.getByLabelText('Coordonnée X :').value).toBe('10');
        expect(screen.getByLabelText('Coordonnée Y :').value).toBe('20');
        expect(screen.getByLabelText('Hauteur :').value).toBe('100');
        expect(screen.getByLabelText('Largeur :').value).toBe('200');
        expect(screen.getByLabelText('Nombre de lignes :').value).toBe('5');
        expect(screen.getByLabelText('Nombre de colonnes :').value).toBe('8');
    });
});
```

La réalisation des tests d'intégration et unitaires a été un élément essentiel du processus de développement de l'application, contribuant à assurer sa qualité, sa fiabilité et sa stabilité. En suivant une approche méthodique et en utilisant des outils appropriés, j'ai pu valider efficacement les différentes fonctionnalités de l'application, garantissant ainsi une expérience utilisateur optimale.

Réalisation du CI/CD

Description

J'ai également mis en place un processus de CI/CD (Continuous Integration/Continuous Deployment) pour assurer un déploiement efficace et automatisé de l'application développée. Cette démarche a été cruciale pour garantir une gestion efficace du cycle de vie du logiciel, en automatisant les phases de construction, de test et de déploiement de l'application.

Méthodologie

La réalisation du CI/CD s'est déroulée en plusieurs étapes, comme décrit ci-dessous :

1. Création des Dockerfiles : Dans un premier temps, j'ai créé des Dockerfiles pour chaque composant de l'application, notamment l'API, l'application web et la documentation associée. Cette étape a permis de containeriser les différents services de l'application, facilitant ainsi leur déploiement et leur gestion.

2. Docker Compose : Un fichier Docker Compose à été fait pour orchestrer les différents conteneurs de l'application. Ce fichier a permis de définir les services nécessaires et leurs dépendances, facilitant ainsi le déploiement et la gestion de l'ensemble de l'application.

3. Mise en place du fichier .gitlab-ci.yml : Ensuite, j'ai configuré un fichier .gitlab-ci.yml pour mettre en place le processus de CI/CD sur l'environnement de développement utilisé, à savoir Etulab. Ce fichier a défini les différentes étapes du pipeline CI/CD, telles que la construction, les tests et le déploiement de l'application.

4. Intégration des tests dans le CI/CD : J'ai intégré les tests unitaires et d'intégration dans le pipeline CI/CD afin de garantir la qualité du code à chaque étape du processus de développement. Ces tests ont été exécutés automatiquement à chaque modification du code source, permettant ainsi de détecter rapidement les éventuelles erreurs ou anomalies.

5. Configuration du serveur pour le CI/CD : En parallèle, j'ai configuré les dossiers, fichiers et paramètres nécessaires sur le serveur de déploiement pour permettre l'intégration continue et le déploiement automatique de l'application. Cette configuration a assuré une gestion efficace du processus de CI/CD dans un environnement de production.

6. Déploiement automatique avec CI/CD : Enfin, j'ai mis en place la copie des fichiers vers le serveur de déploiement à travers le pipeline CI/CD. Cela a permis de lancer automatiquement le docker-compose et de déployer l'application dès la fin du processus de construction et de test.

La mise en place du CI/CD a été un élément clé du processus de développement de l'application, permettant d'automatiser les tâches répétitives, d'améliorer la qualité du code et d'accélérer le cycle de vie du logiciel. En adoptant cette approche, j'ai pu garantir une livraison continue et fiable de l'application, répondant ainsi aux exigences de qualité et de performance attendues.

Exemples de réalisation du CI/CD

Création d'un dockerfile pour l'API

Voici le code d'un Dockerfile qui permet de dockeriser l'API.

```
FROM node:18.17.1
WORKDIR /api
COPY . .
RUN npm install
EXPOSE 3000
CMD [ "npm", "start" ]
```

Le Dockerfile fourni permet de créer une image Docker pour l'API de notre application. Il utilise une image de base Node.js, copie le code source de l'API dans le conteneur, installe les dépendances avec npm, expose le port 3000 et définit la commande de démarrage. Ainsi, il encapsule l'API dans un environnement isolé et portable, simplifiant ainsi son déploiement.
Bien évidemment un Dockerfile est fait pour chaque service de l'application.

Création d'un fichier Docker Compose

Voici la création du Docker Compose qui permet de réunir et de lancer chaque service correctement.

```yaml
services:
  api:
    build:
      context: ./API
      dockerfile: sae-api.Dockerfile
    image: sae_api
    container_name: sae_api
    ports:
      - "3000:3000"
    restart: always

  webapp:
    build:
      context: ./webapp
      dockerfile: sae-webapp.Dockerfile
    image: sae_webapp
    container_name: sae_webapp
    ports:
      - "3001:3001"
    restart: always

  api_doc:
    build:
      context: ./API/doc
      dockerfile: sae-api-doc.Dockerfile
    image: sae_api_doc
    container_name: sae_api_doc
    ports:
      - "3002:80"
    restart: always

  webapp_doc:
    build:
      context: ./webapp/doc
      dockerfile: sae-webapp-doc.Dockerfile
    image: sae_webapp_doc
    container_name: sae_webapp_doc
    ports:
      - "3003:80"
    restart: always
```

Le fichier de configuration Docker Compose définit plusieurs services pour notre application. Chaque service correspond à une composante spécifique de l'application : API, application web, documentation de l'API et documentation de

l'application web. Chaque service est configuré pour construire une image Docker à partir de son propre Dockerfile, spécifier des paramètres tels que le nom du conteneur, le port d'exposition et redémarrer automatiquement en cas d'échec. Cette approche permet de déployer facilement l'ensemble de l'application ainsi que sa documentation, en les isolant dans des conteneurs distincts et en les reliant selon les besoins.

Mise en place du fichier .gitlab-ci.yml pour le pipeline CI/CD

Voici un extrait (pour l'application web et le déploiement) du fichier permettant de faire chaque étape de la construction de l'application et le déploiement sur le serveur à la fin.

Le fichier de configuration GitLab CI/CD ci-dessous définit plusieurs étapes pour la construction, les tests et le déploiement de l'application web. Chaque étape est définie par un job qui spécifie les commandes à exécuter.

- Le job `webapp` installe les dépendances nécessaires pour l'application web en utilisant npm.
- Le job `webapp_build` construit l'application web en exécutant les scripts de construction spécifiques.
- Le job `webapp_test` exécute les tests de l'application web en utilisant Mocha et Chai.
- Ensuite, le job `deploy_job` déploie l'application en copiant les fichiers vers le serveur distant via SSH et en lançant le docker-compose pour démarrer l'application.

Ce fichier de configuration permet d'automatiser le processus de construction, de test et de déploiement de l'application web à chaque modification du code source, assurant ainsi une intégration continue et un déploiement automatique cohérents.

```yaml
  # ---------------------------------- Webapp Install/Build/Test ----------------------------------

webapp:
  stage: install
  script:
    - cd webapp
    - npm install
    - npm install --save react-scripts
  only:
    changes:
      - webapp/**/*

webapp_build:
  stage: build
  script:
    - cd webapp
    - npm uninstall rollup-plugin-terser
    - npm install @rollup/plugin-terser --save-dev
    - npm install react-dom @babel/plugin-proposal-private-property-in-object --save-dev
    - npx react-scripts build
  only:
    changes:
      - webapp/**/*

webapp_test:
  stage: tests
  script:
    - cd webapp
    - npm install --save-dev mocha
    - npm install --save-dev chai
    - npm test
  only:
    changes:
      - webapp/**/*

  # ---------------------------------- API and webapp hosting ----------------------------------
before_script:
  - 'which ssh-agent || ( apt-get update -y && apt-get install openssh-client -y )'
  - 'which rsync || ( apt-get update -y && apt-get install rsync -y )'
  - eval $(ssh-agent -s)
  - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add - > /dev/null
  - mkdir -p ~/.ssh
  - chmod 700 ~/.ssh
  - '[[ -f /.dockerenv ]] && echo -e "Host *\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config'

deploy_job:
  stage: deploy
  script:
    - echo "Copie des fichiers vers le serveur"
    - rsync -avz --delete ./ lolo@next-vertices.com:/home/lolo/sae
    - echo "Lancement du docker compose sur le serveur distant..."
    - ssh lolo@next-vertices.com 'cd /home/lolo/sae && docker compose up -d --build -d'
    - echo "Application déployée vers https://carto.next-vertices.com"
  rules:
    - changes:
        - "*"
      when: always
```

Conclusion

Ce compte rendu des modifications a mis en lumière l'importance des tests d'intégration et unitaires ainsi que la mise en place d'un processus de CI/CD. Ces éléments sont cruciaux pour garantir la qualité, la fiabilité et la stabilité de l'application tout au long de son cycle de vie. En suivant une approche méthodique et en utilisant des outils appropriés, j'ai pu valider efficacement les différentes fonctionnalités de l'application, assurant ainsi une expérience utilisateur optimale. De plus, la mise en place du CI/CD a permis d'automatiser les tâches répétitives de construction, de test et de déploiement, accélérant ainsi le cycle de vie du logiciel et garantissant une livraison continue et fiable de l'application. En adoptant ces bonnes pratiques de développement logiciel, j'ai pu répondre aux exigences de qualité et de performance attendues pour ce proof of concept (POC) d'une application de gestion de cartographie de port commercial.