Documentation CI/CD et KPIs - BobApp

Vue d'ensemble

Ce document présente l'analyse du workflow CI/CD mis en place pour l'application BobApp, incluant les étapes du workflow, les KPIs proposés, et l'analyse des métriques actuelles.

Étapes du Workflow CI/CD

1. Build and Push Docker Images (`build-and-push-images.yml`)

Objectif : Automatiser la création et la publication des images Docker sur Docker Hub

Étapes détaillées :

1. **Authentification Docker Hub**
   - **Objectif** : Se connecter au registry Docker Hub de manière sécurisée
   - **Action** : `docker/login-action@v3`
   - **Seuil critique** : Échec = arrêt du workflow
2. **Configuration Docker Buildx**
   - **Objectif** : Activer les fonctionnalités avancées de build (multi-architecture, cache)
   - **Action** : `docker/setup-buildx-action@v3`
3. **Extraction des métadonnées**
   - **Objectif** : Générer automatiquement les tags et labels pour les images
   - **Actions** : `docker/metadata-action@v5` (frontend et backend)
   - **Règles de tagging** :
     - Branches → `branch-name`
     - Pull Requests → `pr-{number}`
     - Tags → `{version}` et `{major}.{minor}`
     - Branche principale → `latest`
4. **Build et Push des images**
   - **Objectif** : Construire et publier les images Docker
   - **Action** : `docker/build-push-action@v6`
   - **Optimisations** : Cache GitHub Actions, contexte Git natif

Déclencheurs :

- Manuel (`workflow_dispatch`)
- Push sur branches `main` et `develop`
- Tags de version

2. Tests Full-Stack (`full-stack-test.yml`)

Objectif : Exécuter les tests complets Angular + Spring Boot avec génération de rapports de couverture

Étapes détaillées :

1. **Préparation de l'environnement**
   - **Objectif** : Optimiser les temps de build avec la mise en cache
   - **Cache Angular** : `front/node_modules`
   - **Cache Maven** : `~/.m2/repository`
2. **Récupération des artefacts**
   - **Objectif** : Réutiliser le JAR Spring Boot s'il existe déjà
   - **Action** : `actions/download-artifact@v4`
3. **Installation des dépendances Angular**
   - **Objectif** : Installer les dépendances Node.js de manière reproductible
   - **Commande** : `npm ci`
4. **Build Spring Boot**
   - **Objectif** : Compiler l'application Java
   - **Commande** : `mvn clean install -DskipTests`
   - **Condition** : Seulement si le JAR n'a pas été téléchargé
5. **Démarrage du backend**
   - **Objectif** : Lancer l'API Spring Boot pour les tests d'intégration
   - **Commande** : `java -jar back/target/*.jar`
6. **Exécution des tests avec couverture**

- **Tests Angular** : Karma + Jasmine avec ChromeHeadless
- **Tests Java** : JUnit avec JaCoCo pour la couverture
- **Génération des rapports** : Coverage HTML pour les deux stacks

7. **Upload des rapports**
    - **Objectif** : Conserver les rapports de couverture comme artefacts
    - **Artefacts** : `angular-coverage-report` et `java-coverage-report`

3. Analyse SonarQube (`sonar.yml`)

Objectif : Analyser la qualité du code et générer des métriques détaillées

Étapes détaillées :

1. **Démarrage SonarQube**
    - **Objectif** : Lancer une instance SonarQube Community
    - **Service** : Container Docker sur port 9000
2. **Configuration de l'environnement Java**
    - **JDK 11** : Pour le build Spring Boot
    - **JDK 17** : Pour SonarScanner (exigence technique)
3. **Installation SonarScanner CLI**
    - **Objectif** : Installer l'outil d'analyse de code
    - **Version** : 5.0.1.3006
4. **Attente et configuration SonarQube**
    - **Health check** : Vérification du statut UP
    - **Timeout** : 300 secondes maximum
    - **Création du projet** : API REST
5. **Génération du token**
    - **Objectif** : Authentification sécurisée pour l'analyse
    - **API** : `/api/user_tokens/generate`
6. **Exécution de l'analyse**
    - **Sources analysées** : `back/src` et `front/src`
    - **Binaires Java** : `back/target/classes`
    - **Configuration** : Java 11, projet multi-langage
7. **Récupération des résultats**
    - **Quality Gate** : Statut de validation
    - **Métriques** : Bugs, couverture, vulnérabilités
    - **Génération du rapport** : Summary GitHub avec tableau des issues

📊 KPIs Proposés

KPI 1 : Couverture de Code (OBLIGATOIRE)

- **Métrique** : Pourcentage de lignes de code couvertes par les tests
- **Seuil minimum** : **80%** (recommandation industrie)
- **Seuil d'alerte** : **70%** (nécessite action corrective)
- **Seuil critique** : **60%** (bloque le déploiement)
- **Mesure** :
    - Frontend : Karma coverage
    - Backend : JaCoCo coverage
    - Combinée : Moyenne pondérée

KPI 2 : Temps de Build

- **Métrique** : Durée totale du workflow CI/CD
- **Seuil optimal** : **< 10 minutes**
- **Seuil acceptable** : **< 15 minutes**
- **Seuil critique** : **> 20 minutes**
- **Mesure** : Temps total du job `build-and-push`

KPI 3 : Qualité du Code (SonarQube)

- **Métriques composées** :
    - **Bugs** : 0 (tolérance : 2 bugs mineurs max)
    - **Vulnérabilités** : 0 (aucune tolérance)
    - **Code Smells** : < 50 (seuil acceptable)
    - **Duplication** : < 5%
- **Quality Gate** : PASSED obligatoire

KPI 4 : Taux de Succès des Déploiements

- **Métrique** : Pourcentage de workflows réussis
- **Seuil optimal** : **> 95%**
- **Seuil acceptable** : **> 90%**
- **Seuil critique** : **< 85%**
- **Période de mesure** : 30 derniers jours

Tableau de Bord KPIs actuels

KPI	Valeur Cible	Statut Actuel	Tendance
Code Coverage Front	> 80%	⚠ 76.92	-
Code Coverage Back	> 80%	32%	-
Build Time	< 10 min	3m24	-
SonarQube Gate	PASSED	PASSED	-
Success Rate	> 95%	À mesurer à intervale régulier	-

Légende : Bon / ⚠ Attention / Critique / En attente