Document Explicatif

# Contexte

BobApp est une application initialement réalisée et maintenue par un unique développeur. Suite à l’agrandissement du panel d’utilisateurs de son application, Bob décide de mettre en open source son projet. Malheureusement, en l’état, l’applicatif possède de nombreux bogues que Bob a du mal à identifier et à corriger.

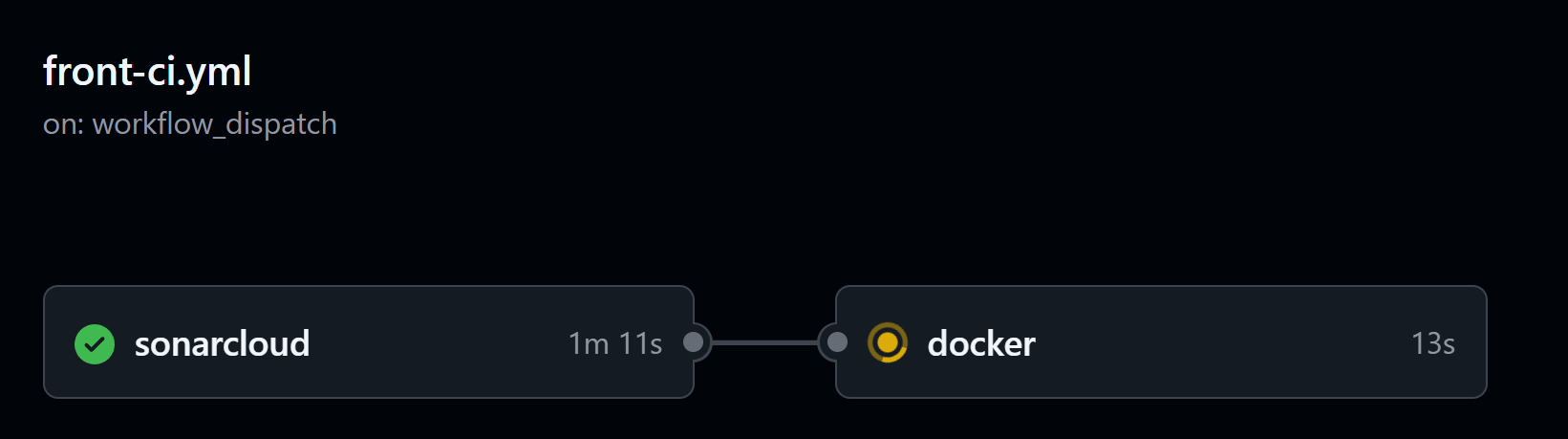
Pour l’aider, la mise en place d’un pipeline CI est décidée. L’objectif est ici de permettre de réaliser les tests unitaires, de contrôler la qualité du code et de réaliser un dépôt des images Docker sur un registre de conteneur de façon automatisée.

L’objectif est ainsi d’offrir un meilleur suivi des incréments de sorte à permettre à Bob et à sa future équipe de travailler plus efficacement et d’éviter les problèmes de lenteur dans les correctifs actuels et futurs.

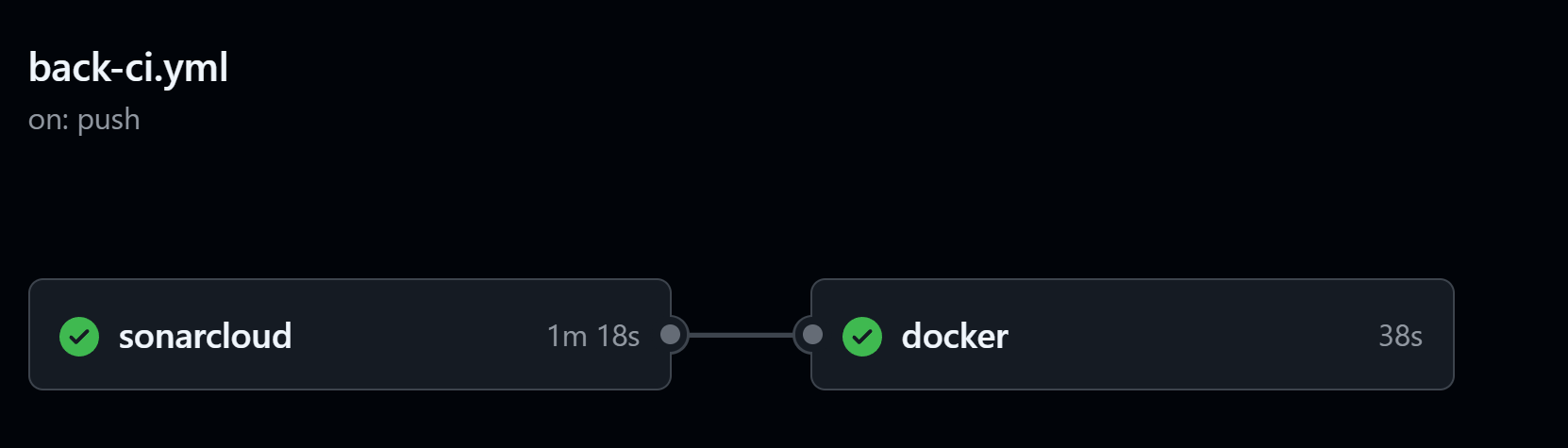
# Workflow

Les GitHub Actions ont été séparées en deux pipelines distinctes. Ceci est normal au vu de la structure du projet, ce dernier possédant une partie frontend et une partie backend. En procédant de la sorte, on profite de la parallélisation des tâches au niveau des runners et on peut isoler plus aisément les différentes sections du projet. La séparation en deux fichiers rend également la lecture des deux pipelines plus digeste.

Les pipelines GitHub Actions actuellement mis en place dans le projet ont pour rôle de réaliser les tâches suivantes :



* Dans la partie frontend
  + Récupération du code source présent dans le dossier **apps/front**
  + Installation et mise en place des variables d’environnement compatible avec un environnement Node.js dont la version est la même que celle prévue par la suite pour l’image Docker.
  + Installation des dépendances de notre application via l’utilisation de Node Package Manager (NPM)
  + Lancement des tests via l’utilisation de la commande prévue dans le fichier **package.json,** celle-ci causant la génération d’un fichier de couverture du code par l’ensemble des tests
  + Lancement d’une analyse du code via l’utilisation de SonarQube (dans un environnement cloud, soit SonarQube Cloud). On doit fournir au client Sonar l’ensemble des arguments en lien avec notre plateforme SonarQube Cloud, les crédentials étant disponible dans les secrets du dépôt Git. Il faut également lui fournir l’emplacement des fichier à scanner et les fichiers de couverture du code.
  + On réalise ensuite, si tous les tests ont été concluants, le build de l’application Angular de sorte à en vérifier la capacité à compiler correctement
  + Donc réalise ensuite le packaging de notre application dans une image docker. Pour ce faire, on va se baser sur le fichier Dockerfile présent à la racine du projet frontend.
  + Pour pouvoir pousser l’image sur le registre d’image de conteneur qu’est DockerHub, il faut dans un premier temps s’y connecter. On réalise cette étape via encore une fois les crédentials présent dans les secrets de notre dépôt Git.
  + Enfin, on pousse l’image sur le registre d’image de conteneur qu’est Dockerhub.



* Dans la partie backend
  + On récupère le code source de l’applicatif présent dans apps/back
  + On va installer l’environnement Java permettant la poursuite des tâches dans le runner sur GitHub. Pour cela, on va installer la version 11 de Java et les dépendances vis-à-vis de Maven
  + On va réaliser les tests unitaires de l’applicatif via JUnit et JaCoCo dans le but d’ensuite se voir offrir un score de couverture du code et un fichier récapitulatif.
  + On va envoyer l’ensemble de nos classes et notre fichier de couverture du code à SonarQube (encore une fois dans un environnement cloud). Pour cela, il va falloir fournir des arguments à SonarClient, dont des crédentials. Ces derniers seront dans les secrets de notre dépôt Git.
  + On va compiler l’applicatif de sorte à vérifier qu’il n’y a pas de soucis de build potentiel en amont de son packaging en image.
  + Si tout se passe bien, on réalise l’image Docker de notre applicatif via le fichier Dockerfile présent à la racine de notre dossier apps/back
  + On se connecte à DockerHub de sorte à pouvoir ensuite y déposer la dernière version de notre image
  + On pousse notre image Docker sur le registre d’image de conteneur.

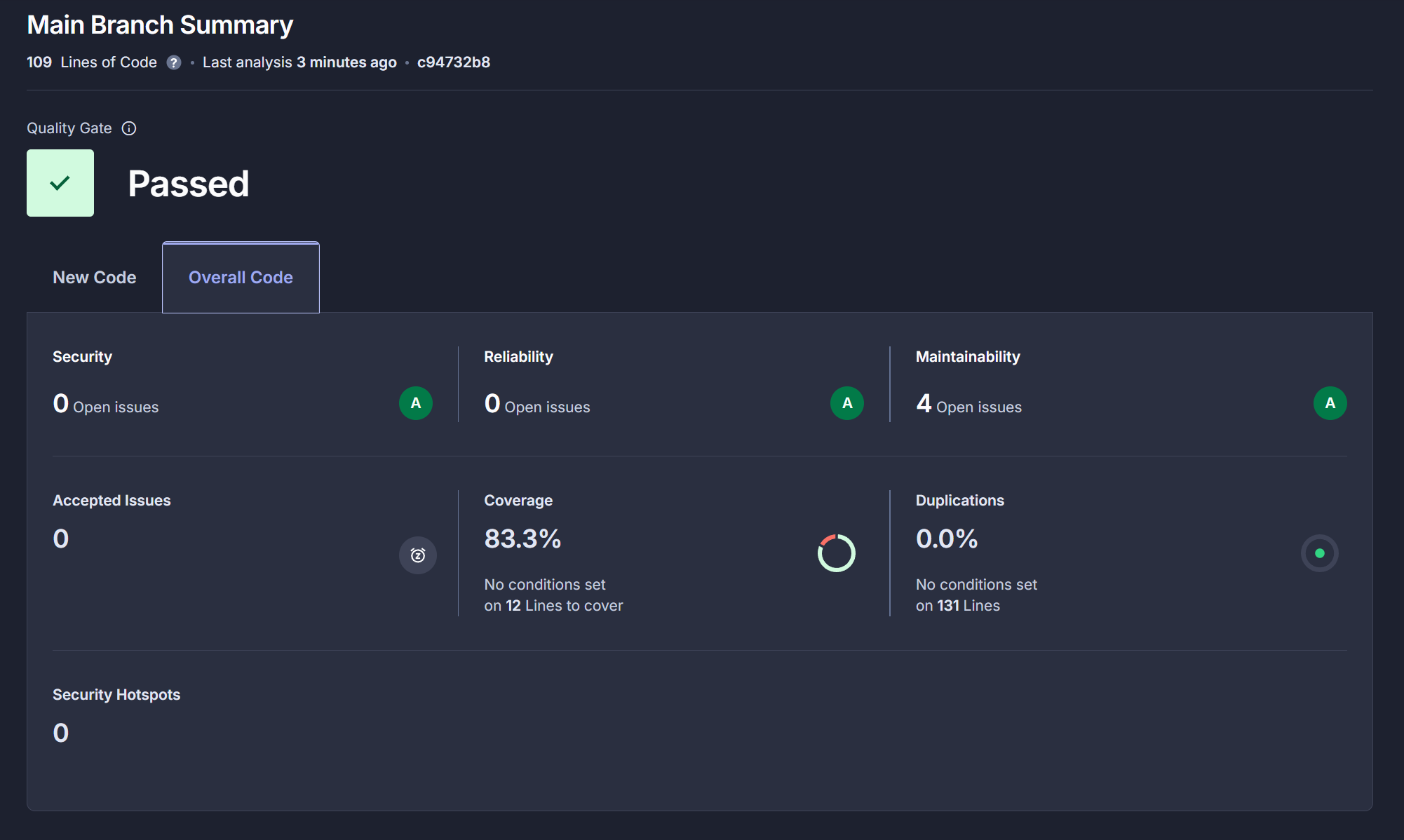
Au niveau du fonctionnement de notre pipeline, l’ensemble des tests est réalisé en cas de push sur n’importe quelle branche, **qu’il s’agisse de main ou pas**. Ceci est prévu de sorte à pouvoir informer un potentiel code reviewer de la réalisation de tests et lui permettre de consulter le score de couverture du code ainsi que les retour de Sonarkube cloud en amont de la validation de la pull request. Si celle-ci se voit validée par le code reviewer, alors les étapes suivantes auront lieu à la suite du rapatriement du code sur la branche principale (génération d’un commit, et donc **push sur main** dans ce cas-là). Dans ce cas de figure, les images Docker seront créées et poussées sur le registre d’image de conteneur. L’utilisation du hash du commit est également prévue de sorte à pouvoir, en plus de l’utilisation du tag **latest,** conserver une trace des différentes versions de notre application.

# Seuils Proposés

* Au niveau de la couverture du code, le score idéal qu’il est courant de devoir respecter dans un environnement professionnel oscille entre 60 et 95%. Il serait, pour cet applicatif, idéal d’atteindre au moins un score de couverture du code de **80%**. C’est d’ailleurs ce score qui est de base prévu au niveau de Sonar.
* Par rapport à SonarQube, ce dernier fourni un seuil de base se basant sur **SonarWay**. Suite à l’analyse du code par SonarQube, des métriques telles que le **score de maintenabilité** (estimé avec une lettre allant de A vers E de façon décroissante), le **nombre de failles de sécurités**, le nombres de **duplication de lignes de code**, sont disponibles. On peut se baser sur ces données pour créer des seuils qu’il faudrait ne pas dépasser. Par exemple, il est possible de demander de ne pas être en dessous d’un score de couverture du code de 80% par les tests unitaires, de ne pas avoir plus de 10% de duplication du code, etc. Pour suivre l’ensemble de ces métriques, il est possible d’aller sur la page SonarQube Cloud du projet. Dans le pipeline CI/CD, chaque partie de l’applicatif s’est vu relié à un projet SonarQube Cloud. On peut donc y observer l’ensemble des métriques indépendamment pour la partie front et la partie back de l’application.
* Une autre métrique intéressante au niveau de SonarQube est celle de la dette technique. Cette donnée présente le potentiel temps qu’il faudrait en théorie passer si l’on avait besoin de corriger l’ensemble des problèmes actuellement présents dans le code. Ne pas dépasser une dette technique de plusieurs heures et se voir confronté à devoir passer plusieurs journées à corriger des erreurs est important si l’on veut pouvoir réagir rapidement aux retours des utilisateurs.

# Analyse des Métriques

* Pour le backend :
  + A l’heure actuelle, le score de couverture du code est bien trop bas pour l’applicatif backend. Il faudrait augmenter le nombre de tests unitaires dans le but de pouvoir, à chaque modification future de l’applicatif, suivre l’évolution des fonctionnalités. Ce score de couverture bas est sans aucun doute l’une des causes principales de la non-détection des bogues éventuels que les clients ont relevé dans leurs commentaires. En améliorant la couverture du code, on pourra identifier des problèmes en amont d’un éventuel déploiement et donc intercepter les bogues avant qu’ils n’arrivent sur l’applicatif en production. Les clients se verront ainsi à coup sûr moins souvent mis en contact avec des bogues.
  + L’autre métrique qu’il peut être intéressant de regarder est le score de fiabilité global de l’application. Ce score nous indique, encore une fois dans l’applicatif backend, de la présence d’un souci pouvant causer un bogue. Avis Utilisateurs
  + Parmi les points positifs, on peut par contre remarquer qu’il y a un taux de duplication du code nul. Ce score nous indique en gros que l’applicatif sera plus simple à relire et que la collaboration sera plus aisée dans le cadre du passage à l’open-source.



* Pour le frontend :
  + Au niveau des retours de Sonar, l’applicatif frontend semble avoir des métriques plus que satisfaisante. La couverture de code dépasse le seuil prévu par le scanner (80%) et les score de duplication de code, de fiabilité de l’applicatif ainsi que de sécurité sont au vert. Que du bon !

Pour garantir un niveau de qualité constant sur le projet BobApp et faciliter la détection précoce des défauts, nous proposons d’utiliser les quality gates standards de SonarCloud, en les adaptant légèrement au contexte spécifique de l’application.

* Le premier indicateur clé à suivre est le taux de couverture du code par les tests automatisés. Nous recommandons de le fixer à un minimum de 80 %, afin d’assurer qu’une large partie du code soit testée, ce qui diminue fortement les risques de régressions lors des évolutions futures.
* Un second indicateur crucial est le nombre de nouveaux bugs bloquants (« new blocker issues »), qui doit impérativement rester à zéro. Cela évite l’introduction de défauts critiques en production et renforce la fiabilité de l’application dès sa phase de développement.

Par ailleurs, nous suggérons également de surveiller d’autres indicateurs déjà présents dans le quality gate par défaut de SonarCloud, tels que :

* Le taux de duplication du code, idéalement inférieur ou égal à 3 %, afin de limiter la dette technique ;
* La maintenabilité, évaluée via les code smells, dont le seuil acceptable varie selon la taille du projet mais doit rester faible pour préserver la clarté du code ;
* Et enfin, les hotspots de sécurité, qui doivent être absents ou entièrement corrigés pour éviter toute faille non détectée.

Ces KPIs sont automatiquement contrôlés par SonarCloud à chaque exécution de la pipeline, et le quality gate bloque toute validation en cas de non-respect. Ainsi, Bob dispose d’une visibilité immédiate sur la santé du code, peut se concentrer sur les améliorations prioritaires, et sécurise les contributions futures de la communauté open source.

# Avis utilisateurs



Les avis utilisateurs sont pour la majorité en lien avec une frustration venant d’une absence de correction des bogues présents dans l’application. Cette absence de rapidité de réaction face à cette situation a causé pour les utilisateurs une frustration pouvant, pour certains, causer la recherche d’une alternative. En améliorant notre temps de réaction et de correction des bogues, nous devrions pouvoir améliorer grandement la note globale de l’applicatif. Les fonctionnalités actuellement à corriger sont :

* L’ajout d’une nouvelle blague à l’applicatif (l’envoi de la requête semble tourner en boucler et la cause probable est une absence de communication entre le frontend et le backend)
* Problème lors du post d’une vidéo. Ceci est peut-être dû au format du fichier à envoyer ou du type de données envoyées lors de la requête entre le frontend et le backend, ces dernières n’utilisant peut-être pas les mêmes DTOs.
* L’un des utilisateurs ne reçoit plus aucune blague, sans doute suite à un soucis vis-à-vis de la génération de contenu ou via un bogue dans la notification des utilisateurs ?