

APPLICATION BOBAPP



Document explicatif

Sommaire

1. Contexte et objectifs.....	2
2. Mise en place des Workflows.....	2
2.1. Workflow CI (Backend & Frontend).....	2
2.1.1. Analyse du workflow CI Frontend.....	3
2.1.2. Analyse du workflow CI Backend.....	3
2.2 Workflow CD (Backend & Frontend).....	4
2.2.1. Analyse du workflow CD Frontend.....	4
2.2.2. Analyse du workflow CD Backend.....	4
3. Proposition de KPI et seuils à respecter.....	5
3.1. KPI appliqués dans le Quality Gate de SonarCloud.....	5
3.1.1. Couverture du code (Coverage).....	5
3.1.2. Lignes dupliquées (Duplicated Lines %).....	6
3.1.3. Note de Maintenabilité (Maintainability Rating).....	6
3.1.4. Problèmes bloquants (Blocker Issues).....	6
3.1.5. Note de Fiabilité (Reliability Rating).....	6
3.1.6. Note de Sécurité (Security Rating).....	6
3.2. Paramétrage des Quality Gates dans SonarCloud.....	7
3.3. Présentation des Quality Gates de BobApp.....	7
4. Analyse des métriques.....	8
4.1. Analyse du Backend.....	8
4.2. Analyse du Frontend.....	9
4.3. Analyse métrique via SonarCloud.....	10
5. Les avis actuels.....	11
5.1. Absence de bouton pour poster une blague.....	11
5.2. Absence de bouton pour poster une vidéo.....	11
5.3. Problème de communication avec l'utilisateur.....	12
5.4. Insatisfaction générale et désinstallation de l'application.....	12
6. Conclusion.....	12

1. Contexte et objectifs

BobApp est une application open-source permettant de lire et partager des blagues. Bien que populaire, l'application rencontre des problèmes liés à son développement, à la gestion des bugs et aux déploiements manuels, qui sont chronophages et sujets aux erreurs. Actuellement, Bob, travaillant sur ce projet seul, peine à corriger rapidement les problèmes et à ajouter de nouvelles fonctionnalités.

Pour résoudre ces défis, une démarche d'intégration continue (CI) et de déploiement continu (CD) automatisée a été mise en place avec GitHub Actions, SonarCloud et Docker Hub. Cette approche vise à automatiser les processus, améliorer la qualité du code et simplifier les déploiements.

Les principaux objectifs sont :

- Automatiser la validation des pull-requests et l'exécution des tests ;
- Vérifier la qualité du code et générer des rapports de couverture ;
- Construire et pousser des images Docker pour le frontend et le backend ;
- Analyser la qualité du code avec SonarCloud, en définissant des KPI comme la couverture minimale et l'absence de « new blocker issues » ;
- Automatiser et fiabiliser le processus de déploiement, réduisant ainsi le temps et l'effort nécessaires.

2. Mise en place des Workflows

Afin d'automatiser et d'optimiser les processus de développement et de déploiement de BobApp, quatre workflows GitHub Actions ont été mis en place pour gérer l'intégration continue (CI) et le déploiement continu (CD) du backend et du frontend.

Ces workflows sont conçus pour garantir une haute qualité de code, accélérer les tests et déploiements, et minimiser les erreurs humaines dans la gestion des versions.

2.1. Workflow CI (Backend & Frontend)

Le but des workflows CI pour le backend et le frontend est de valider et tester automatiquement chaque changement de code avant sa fusion dans la branche principale. Cela permet de garantir que chaque modification respecte les standards de qualité et n'introduit pas de régressions.

2.1.1. Analyse du workflow CI Frontend

Le workflow CI du frontend se déclenche sur chaque **push** ou **pull request** vers la branche **main**, ciblant les modifications dans le dossier **front/**. Les étapes sont les suivantes :

1. Vérification du code source: Le code du dépôt est récupéré avec **actions/checkout@v4**.
2. Configuration de l'environnement Node.js: Utilisation d'**actions/setup-node@v4** pour installer la version **20.15.1** de Node.js.
3. Configuration de JDK 17: Nécessaire pour l'analyse SonarQube, le JDK 17 est configuré via **actions/setup-java@v4**.
4. Mise en cache des modules Node.js: Les dépendances sont mises en cache (**front/node_modules**) pour accélérer les futures exécutions.
 - La clé de cache est générée en fonction du fichier **package-lock.json**, qui liste les dépendances.
5. Installation des dépendances: Les dépendances du projet sont installées avec **npm ci**, une commande qui garantit une installation cohérente basée sur le fichier **package-lock.json**.
6. Configuration de Xvfb: Xvfb est installé pour permettre l'exécution de tests en mode headless (*sans interface graphique*).
7. Lancement des tests: Les tests sont exécutés avec **npm run test:prod -- --browsers=ChromeHeadless**.
8. Analyse de la qualité du code avec SonarQube: L'analyse est effectuée à l'aide de **npx sonar-scanner** pour vérifier la qualité du code et la couverture des tests.
9. Vérification des fichiers de couverture: Affiche la liste des fichiers dans le dossier **coverage/bobapp** pour vérifier la présence du rapport de couverture (**lcov.info**).
10. Téléchargement des résultats de tests: Le rapport de couverture est téléchargé en tant qu'artefact pour consultation future.

2.1.2. Analyse du workflow CI Backend

Le workflow CI backend suit une logique similaire, adaptée aux technologies Java et Maven.

1. Vérification du code source: Le code est récupéré avec **actions/checkout@v4**.
2. Configuration de JDK 17: Le JDK 17 est configuré via **actions/setup-java@v4** pour la compilation et l'analyse du code.

3. Mise en cache des packages SonarCloud et Maven : Les caches sont utilisés pour les fichiers de SonarCloud et les dépendances Maven (`~/.m2`), ce qui réduit le temps de compilation.
4. Compilation et analyse du code : Le projet est compilé avec Maven (`mvn -B verify`), et l'analyse SonarQube est lancée pour évaluer la qualité du code.
5. Téléchargement du rapport JaCoCo : Le rapport de couverture généré par JaCoCo est téléchargé pour permettre une analyse approfondie des tests.

2.2 Workflow CD (Backend & Frontend)

Les workflows CD sont déclenchés une fois que les workflows CI ont validé les changements de code. Leur rôle est de déployer automatiquement les applications sur les environnements de staging ou de production.

2.2.1. Analyse du workflow CD Frontend

Le workflow CD frontend s'exécute après la complétion du workflow CI frontend. Il comprend les étapes suivantes :

1. Vérification du code source : Le code est récupéré avec `actions/checkout@v4`.
2. Configuration de Node.js : Utilisation d'`actions/setup-node@v4` pour configurer la version `20.15.1`.
3. Installation des dépendances : Les dépendances sont installées avec `npm install` dans le dossier `front/`.
4. Construction du projet : Le projet est construit avec la commande `npm run build`.
5. Connexion à Docker Hub : Authentification sur Docker Hub en utilisant `docker/login-action@v3`, avec les identifiants stockés dans les secrets GitHub (`DOCKER_USERNAME` et `DOCKER_PASSWORD`).
6. Construction et push de l'image Docker : L'image Docker est construite et poussée sur Docker Hub via `docker/build-push-action@v6.9.0`.
 - Le contexte de build est le dossier `front/`, et le fichier Docker utilisé est `front/Dockerfile`.
 - L'image est taggée avec `latest` et poussée sous le nom `username/frontend:latest`.

2.2.2. Analyse du workflow CD Backend

Le workflow CD backend suit un processus similaire, adapté aux technologies backend.

1. Vérification du code source : Le code est récupéré avec `actions/checkout@v4`.

2. Configuration de JDK 17 : Configuration du JDK via `actions/setup-java@v4`.
3. Construction du projet avec Maven : Le projet est construit avec la commande `mvn clean install -DskipTests`.
4. Connexion à Docker Hub : Authentification sur Docker Hub avec `docker/login-action@v3`.
5. Construction et push de l'image Docker : L'image Docker est construite et poussée sur Docker Hub.
 - Le contexte de build est le dossier `back/`, et le fichier Docker utilisé est `back/Dockerfile`.
 - L'image est taggée avec `latest` et poussée sous le nom `username/backend:latest`.
6. Vérification des images Docker : La commande `docker images` affiche la liste des images construites pour vérification.

<https://hub.docker.com/repository/docker/chrysalb/frontend/general>

<https://hub.docker.com/repository/docker/chrysalb/backend/general>

3. Proposition de KPI et seuils à respecter

Afin de garantir un code de qualité pour BobApp, il est essentiel de définir des KPI (indicateurs de performance clé). Ces KPI sont intégrés dans les Quality Gates de SonarCloud et permettent de s'assurer que chaque modification de code respecte des critères de qualité avant sa validation et son déploiement.

Voici les KPI principaux appliqués à BobApp, avec leurs seuils de qualité définis dans le Quality Gate de SonarCloud.

3.1. KPI appliqués dans le Quality Gate de SonarCloud

3.1.1. Couverture du code (Coverage)

Seuil : Couverture du code $\leq 80 \%$

Explication : La couverture de code mesure le pourcentage de lignes de code testées par des tests unitaires. Un seuil de 80 % est nécessaire pour garantir une bonne couverture des tests, ce qui permet de détecter les régressions et les défauts plus tôt dans le cycle de développement

Condition : Si la couverture du code est inférieure à 80 %, le Quality Gate échoue

3.1.2. Lignes dupliquées (Duplicated Lines %)

Seuil : Lignes dupliquées > 3.0 %

Explication : Ce KPI mesure le pourcentage de code dupliqué dans le projet. Un taux de duplication trop élevé peut entraîner une mauvaise maintenance du code et augmenter les risques de bugs

Condition : Si le pourcentage de lignes dupliquées est supérieur à 3.0 %, le Quality Gate échoue

3.1.3. Note de Maintenabilité (Maintainability Rating)

Seuil : Note de maintenabilité \leq A

Explication : La note de maintenabilité évalue la facilité avec laquelle le code peut être modifié ou corrigé. Une note inférieure à A indique que le code pourrait être difficile à maintenir, ce qui augmente les risques à long terme

Condition : Si la note de maintenabilité est inférieure à A, le Quality Gate échoue

3.1.4. Problèmes bloquants (Blocker Issues)

Seuil : Problèmes bloquants > 0

Explication : Un problème bloquant est un bug ou une vulnérabilité grave qui empêche le bon fonctionnement de l'application. Avoir des problèmes bloquants dans le code est inacceptable car cela pourrait rendre l'application instable ou vulnérable

Condition : Si des problèmes bloquants sont détectés, le Quality Gate échoue

3.1.5. Note de Fiabilité (Reliability Rating)

Seuil : Note de maintenabilité \leq A

Explication : La note de fiabilité évalue le nombre et la gravité des défauts qui affectent la stabilité et le bon fonctionnement de l'application. Un code fiable est essentiel pour éviter des erreurs en production

Condition : Si la note de fiabilité est inférieure à A, le Quality Gate échoue

3.1.6. Note de Sécurité (Security Rating)

Seuil : Note de maintenabilité \leq A

Explication : La note de sécurité évalue les risques de sécurité dans le code, notamment les vulnérabilités qui pourraient être exploitées par des attaquants. Assurer une sécurité maximale est essentiel pour protéger l'application

Condition : Si la note de fiabilité est inférieure à A, le Quality Gate échoue

3.2. Paramétrage des Quality Gates dans SonarCloud

Les KPI mentionnés ci-dessus sont configurés dans le Quality Gate de SonarCloud pour BobApp. Voici comment il est possible de les vérifier et les ajuster dans SonarCloud :

1. Accéder aux Quality Gates

- Connectez-vous à SonarCloud et sélectionnez votre projet BobApp
- Allez dans l'onglet « Quality Gates » dans le menu de votre projet

2. Modifier ou Créer un Quality Gate

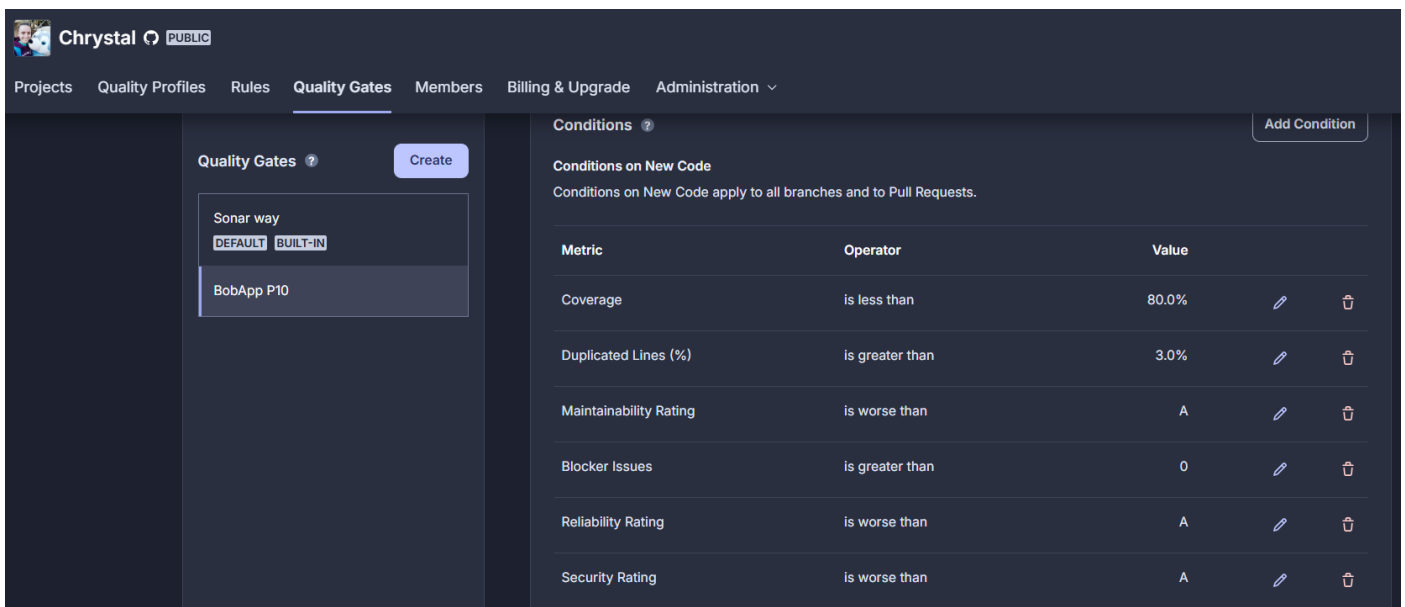
- Si vous souhaitez modifier un Quality Gate existant, sélectionnez le Quality Gate actuel (par défaut « Sonar way ») ou créez un nouveau Quality Gate en cliquant sur le bouton « Create » et en le nommant
- Dans la partie droite de l'écran, choisissez le metric à modifier en cliquant sur l'icône « stylo » à droite de la ligne. Sinon, vous pouvez cliquer sur « Add Condition » pour en ajouter une nouvelle

3. Sauvegarde des modifications

- La sauvegarde est automatique après chaque modification

3.3. Présentation des Quality Gates de BobApp

Voici la capture d'écran des Quality Gates configurés pour BobApp dans SonarCloud :



The screenshot shows the SonarCloud interface for the 'BobApp' project. The 'Quality Gates' tab is selected. On the left, there is a list of quality gates: 'Sonar way' (with 'DEFAULT' and 'BUILT-IN' tags) and 'BobApp P10'. A 'Create' button is visible. On the right, the 'Conditions' section is expanded, showing a table of conditions for 'Conditions on New Code'. The table has columns for Metric, Operator, and Value. The conditions listed are:

Metric	Operator	Value
Coverage	is less than	80.0%
Duplicated Lines (%)	is greater than	3.0%
Maintainability Rating	is worse than	A
Blocker Issues	is greater than	0
Reliability Rating	is worse than	A
Security Rating	is worse than	A

Les conditions et seuils appliqués garantissent un code de qualité avant chaque déploiement.

4. Analyse des métriques


4.1. Analyse du Backend


Le rapport de couverture du Back-End est généré via JaCoCo et présente les éléments suivants :

- Colonnes :
 - Element : Nom du package ou module
 - Missed Instructions : Instructions non couvertes par les tests
 - Cov. (Couverture) : Pourcentage de couverture des instructions
 - Missed Branches : Branches à couvrir et non couvertes par les tests
 - Cov. (Branches) : Pourcentage de couverture des branches
 - Missed : Nombre de lignes ou de tests non couverts dans différentes catégories
 - Cxty (Complexité) : Complexité des méthodes ou classes testées
 - Missed Methods : Méthodes à couvrir et non couvertes
 - Missed Classes : Classes à couvrir et classes non couvertes
 - Lines, Missed : Lignes de code à couvrir et ligne non couvertes






Le rapport de couverture du Back-End révèle que les résultats varient selon les éléments du projet (**model**, **data**, **service**, **controller**, etc.).

Le taux de couverture global du Back-End, qui est de 32 % ce qui reste insuffisant pour répondre aux attentes du Quality Gate. Cela inclut la couverture des tests pour les Instructions, les Branches, les Méthodes et les Classes. Ce faible taux souligne la nécessité de renforcer les tests unitaires sur cette partie de l'application.

 bobapp

 Sessions

bobapp

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
 com.openclassrooms.bobapp.model	<div><div></div></div>	0 %		n/a	7	7	13	13	7	7	1	1
 com.openclassrooms.bobapp.data	<div><div></div></div>	49 %	<div><div></div></div>	50 %	5	7	8	18	3	5	1	2
 com.openclassrooms.bobapp.service	<div><div></div></div>	25 %		n/a	1	2	4	7	1	2	0	1
 com.openclassrooms.bobapp.controller	<div><div></div></div>	54 %		n/a	1	2	1	4	1	2	0	1
 com.openclassrooms.bobapp	<div><div></div></div>	37 %		n/a	1	2	2	3	1	2	0	1
Total	90 of 134	32 %	2 of 4	50 %	15	20	28	45	13	18	2	6

Created with JaCoCo 0.8.5.201910111838

4.2. Analyse du Frontend

Le rapport de couverture du Front-End est généré via lcov et présente les indicateurs suivants :

- Taux de couverture global :
 - Statements : 76,92 %
 - Branches : 100 %
 - Fonctions : 57,14 %
 - Lines : 83,33 %

Le rapport de couverture détaille aussi les colonnes suivantes :

- File : Fichier du code
- Statements : Nombre de déclarations couvertes par les tests
- Branches : Couverture des conditions (branches) dans le code
- Fonctions : Pourcentage de fonctions couvertes par les tests
- Lines : Lignes de code couvertes par les tests

Le Front-End présente une couverture de statements de 76,92 % et de fonctions de 57,14 %. Bien que la couverture des statements soit relativement bonne, la couverture des fonctions pourrait être améliorée pour garantir un test plus complet du code. Certaines parties du code restent non couvertes, ce qui pourrait entraîner des risques de régressions dans des sections critiques de l'application.

All files

76.92% Statements 10/13 100% Branches 0/0 57.14% Functions 4/7 83.33% Lines 10/12

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

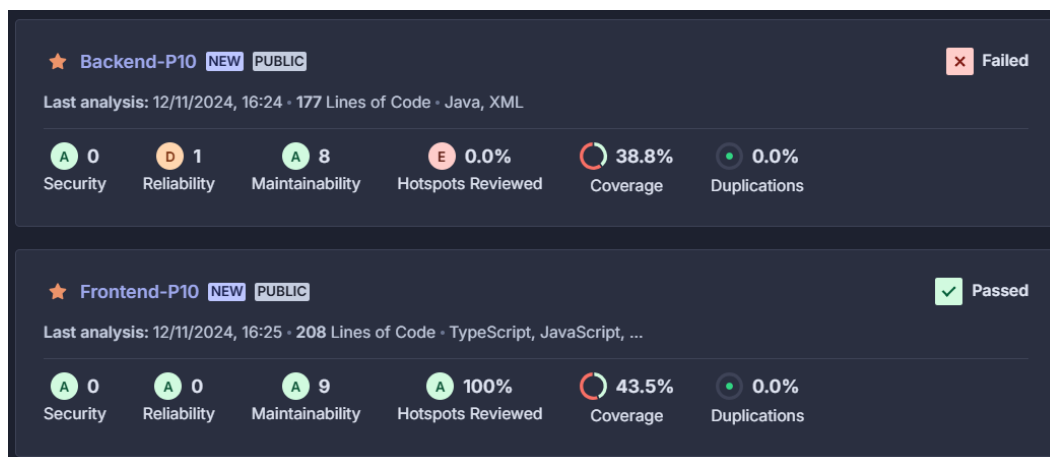
File		Statements	Branches	Fonctions	Lines
app	<div><div></div></div>	60%	3/5	0/0	33.33%
app/services	<div><div></div></div>	87.5%	7/8	100%	75%

4.3. Analyse métrique via SonarCloud

Les indicateurs de SonarCloud, qui sont intégrés au Quality Gate « BobApp P10 », fournissent un aperçu global de la qualité du code. Ces métriques incluent des éléments comme :

- Couverture du code (Coverage)
- Lignes dupliquées (Duplications)
- Notes de Sécurité (Security), Fiabilité (Reliability) et Maintainabilité (Maintainability)

D'après SonarCloud, le Front-End affiche une qualité relativement bonne, avec des problèmes principalement au niveau des tests de Fonctions. En revanche, le Back-End a plusieurs points faibles, notamment une couverture de code insuffisante et une faible fiabilité.



Le Front-End présente une bonne qualité générale, bien que le Code Coverage soit encore insuffisant (43,5 %). En revanche, le Back-End est le point faible de l'application, avec un score de fiabilité (*reliability*) faible, huit points de maintainability et un Code Coverage nettement insuffisant (38,8 %).

Note : Les valeurs de Code Coverage peuvent varier entre les tests unitaires et SonarCloud, chaque outil utilisant une méthode de calcul différente.

5. Les avis actuels

Lors de l'analyse des retours utilisateurs, plusieurs problèmes ont été identifiés. Voici une présentation des problèmes rencontrés ainsi que les solutions proposées pour chacun :

5.1. Absence de bouton pour poster une blague



Je mets une étoile car je ne peux pas en mettre zéro ! Impossible de poster une suggestion de blague, le bouton tourne et fait planter mon navigateur !

Contexte : Actuellement, l'application ne permet pas aux utilisateurs de publier des blagues, ce qui limite l'interactivité et l'engagement des utilisateurs.

Solution proposée : Ajouter un bouton permettant aux utilisateurs de poster leurs propres blagues. Cette fonctionnalité sera développée avec une attention particulière à la qualité du code et à son intégration fluide dans l'application. Grâce au pipeline CI/CD, chaque modification sera testée de manière automatique à travers des tests unitaires et d'intégration, permettant ainsi de valider la fonctionnalité avant le déploiement. Cela garantit également une mise à jour rapide et sans impact pour les utilisateurs.

5.2. Absence de bouton pour poster une vidéo

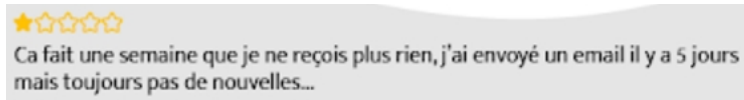


#BobApp j'ai remonté un bug sur le post de vidéo il y a deux semaines et il est encore présent ! Les devs vous faites quoi ????

Contexte : L'application ne permet pas aux utilisateurs de télécharger ou de publier des vidéos, ce qui limite son interactivité et son attrait pour un certain type d'audience.

Solution proposée : Développer un bouton et ses fonctionnalités permettant aux utilisateurs de télécharger et de publier des vidéos. Une fois développée, la fonctionnalité sera soumise à des tests unitaires, des tests d'intégration et des tests end-to-end automatisés dans le pipeline CI/CD. Ce processus permettra de garantir que la nouvelle fonctionnalité fonctionne de manière fluide et ne génère pas de régressions dans d'autres parties de l'application. Le pipeline CI/CD assurera également une gestion efficace des versions et un déploiement sans heurts.

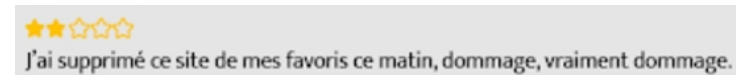
5.3. Problème de communication avec l'utilisateur



Contexte : Les utilisateurs expriment des difficultés à obtenir des réponses à leurs questions. Un manque de communication efficace nuit à l'expérience utilisateur.

Solution proposée : Mettre en place un service client intégré à l'application, sous la forme d'un chat en direct ou d'une messagerie, permettant aux utilisateurs de poser leurs questions et de recevoir des réponses instantanées. L'intégration de ce service dans l'application sera suivie de tests automatisés dans le pipeline CI/CD, garantissant son bon fonctionnement et sa performance sur tous les appareils. Cela permettra une communication fluide avec les utilisateurs et des mises à jour fréquentes pour améliorer constamment l'expérience.

5.4. Insatisfaction générale et désinstallation de l'application



Contexte : Un utilisateur a exprimé sa déception quant à l'application sans spécifier les raisons, ce qui a conduit à son retrait des favoris.

Solution proposée : Mettre en place un formulaire intégré à l'application permettant aux utilisateurs de signaler les bugs et les problèmes rencontrés. Ce formulaire devrait offrir des options pour décrire les erreurs de manière détaillée, ce qui permettra de mieux comprendre les causes de la déception et d'y répondre rapidement. L'utilisation de ce formulaire pourra être automatisée dans le processus CI/CD pour faciliter la remontée des problèmes en production, garantissant une résolution rapide des incidents et améliorant ainsi la satisfaction des utilisateurs.

6. Conclusion

En résumé, pour garantir une meilleure qualité de l'application BobApp, il est crucial de se concentrer sur l'amélioration du Code Coverage et la résolution rapide des bugs, afin de répondre aux attentes des utilisateurs et de renforcer la fiabilité de l'application.