Analyse continue

Florent Pruvost (modifié par Pierre Ramet)

September 20, 2026

Contents

Inspection continue de la qualité :

- Tests unitaires et badge de couverture dans GitLab
- Analyse plus poussée avec SonarQube

1 Introduction

Vérifiez que vos pipelines sont toujours fonctionnels.

Pour cela, trois prérequis à vérifier :

1. le jeton d'accès personnel (**Personal Access Token**) GitLab,
2. le service `gitlab-runner`,
3. le service `podman`.

Pour avoir accès à votre jeton et à la bonne configuration podman, vous pouvez utiliser le script setenv-podman.sh, commande à taper dans chaque nouveau terminal ou à ajouter dans votre `.bashrc` :

```
source ~/setenv-podman.sh
# rappel : un personal access token valide doit se trouver dans le fichier ~/.gitlabtoken
```

Ensuite, pour lancer le service `gitlab-runner` :

- soit le service est déjà configuré et fonctionnel

```
systemctl --user status gitlab-runner.service
# répond : active (running)
```

- soit vous le lancez manuellement dans un nouveau terminal

```
~/gitlab-runner run
```

Enfin, pour valider que les pipelines fonctionnent, lancez un pipeline. Par exemple, manuellement via la page Build -> Pipelines -> Run pipeline, en choisissant la branche 'auto' du projet **Heat** sur laquelle vous avez déjà travaillé.

Vous pouvez repartir sur votre branche 'auto' de Heat tout en commentant certains jobs dans votre fichier `.gitlab-ci.yml` commentez par exemple : *gtest*, *release*, *deploy*, *pages*. Commitez, poussez pour vérifier que votre pipeline composé **a minima** de *buildenv*, *build* et *test* fonctionne.

2 Rapports intégrés dans GitLab

2.1 Tests unitaires

La liste des tests unitaires peut être longue et il est pratique d'avoir une vue synthétique du statut des tests (succès/échec) des derniers commits de notre branche. Il existe une fonctionnalité GitLab pour visualiser le rapport de tests directement sur la page du pipeline (cf. unit test report).

Mettre en place cette technique sur votre projet Heat. Pour cela, modifiez le job **test** dans `.gitlab-ci.yml` afin d'ajouter la génération d'un rapport de tests au format JUnit. Ensuite, ajoutez le fichier `ctest.xml` aux artefacts comme démontré dans la documentation GitLab :

```
script:
  - ctest -V --test-dir build --output-junit ctest.xml
artifacts:
  when: always
  reports:
    junit: build/ctest.xml
```

Vous devriez voir `Uploading artifacts as "junit" to coordinator` dans les logs du job *test*. Rendez-vous sur la page du dernier pipeline exécuté, onglet Tests (à côté de Pipeline ou Jobs), pour voir le résultat.

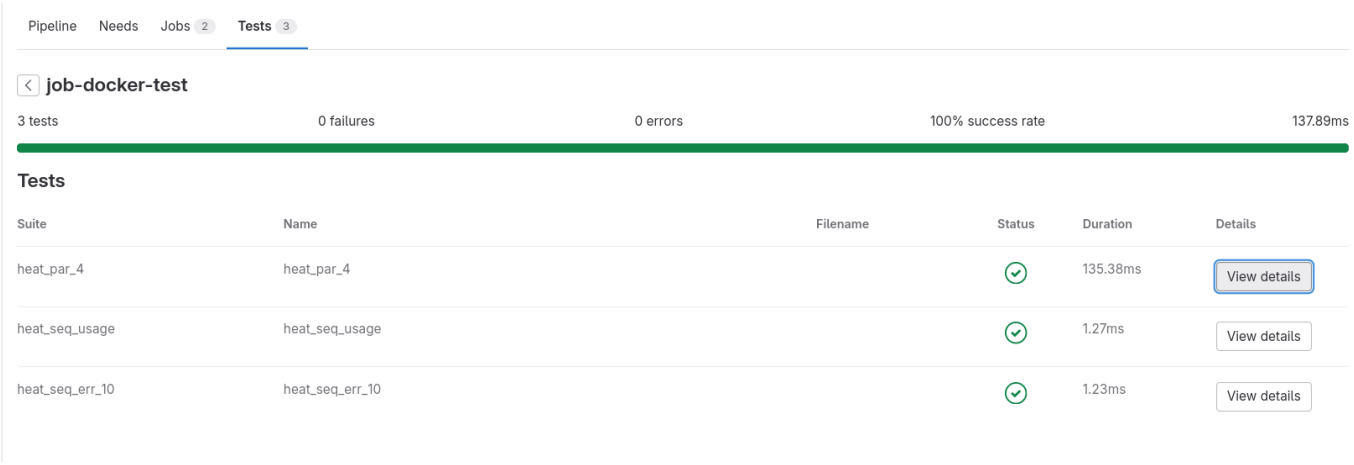


Figure 1: Rapport des tests unitaires sur GitLab

Cela vous permet de rapidement détecter les tests en échec et d'afficher leurs logs.

2.2 Couverture des tests

Les tests de couverture informent sur le pourcentage de lignes couvertes (ou % de fonctions, ou % d'embranchements tels que les if, switch) par les tests unitaires. C'est une bonne pratique de développement de vérifier que les nouvelles lignes de code sont bien couvertes par des tests.

GitLab permet de récupérer une métrique de couverture calculée pendant l'un des jobs (cf. test coverage). De plus, avec ce rapport, les relecteurs d'une "merge request" pourront vérifier si les nouvelles lignes de code proposées sont couvertes ou non (liseré vert ou orange sur le côté gauche du code source dans l'onglet Changes).

Ajoutez au job *test* le rapport de couverture. Pour cela, il faut ajouter le paquet **gcovr** dans l'image docker de test, forcer sa reconstruction en enlevant les éventuelles conditions affectant le job *buildenv*.

Modifiez le job *build* : compiler l'application Heat avec l'option de couverture de code `-DCMAKE_C_FLAGS="--coverage"`.

Ensuite, ajoutez le nécessaire au job *test* :

- génération du rapport de couverture au format *cobertura* avec **gcovr**,
- ajouter `coverage: /^\s*lines:\s*\d+.\d+\%/` pour indiquer à GitLab (expression régulière sur les logs) comment récupérer le pourcentage de couverture global sur le code source,
- ajouter le rapport Cobertura dans les artefacts.

```
script:
  - ctest -V --test-dir build --output-junit ctest.xml
  - cd build/ && gcovr --xml-pretty --exclude-unreachable-branches --print-summary -o coverage.xml --ro
coverage: /^\s*lines:\s*\d+.\d+\%/
artifacts:
  when: always
  reports:
    junit: build/ctest.xml
    coverage_report:
      coverage_format: cobertura
      path: build/coverage.xml
```

Vous devriez observer le pourcentage sur la page du job *test* dans les métadonnées sur la droite.

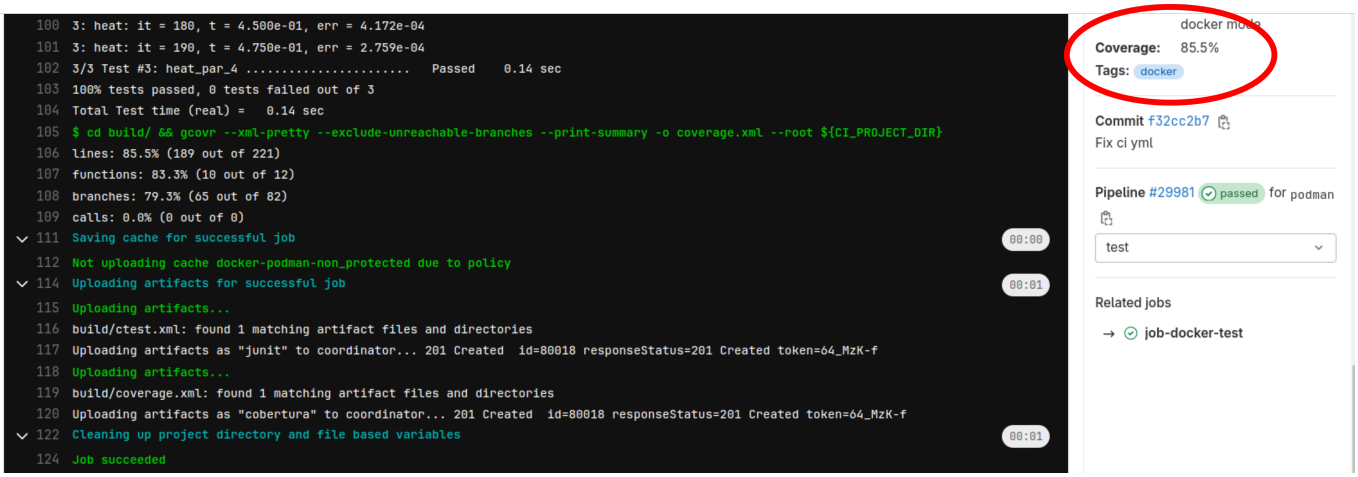


Figure 2: Pourcentage de couverture

2.3 Exploitation des rapports dans les Merge Requests (Widgets et Code Quality)

L'intérêt majeur de générer des rapports standardisés (JUnit, Cobertura, Code Climate) au niveau de GitLab CI réside dans leur intégration native au sein de l'interface des Merge Requests (MR).

Lorsqu'un développeur soumet une branche pour relecture, GitLab compare automatiquement les rapports générés sur sa branche avec ceux de la branche de destination (généralement `master`) et affiche des widgets interactifs d'aide à la revue de code.

2.3.1 Scénario appliqué au projet Heat : suivi des tests et de la couverture

Imaginons qu'un développeur travaille sur une branche de fonctionnalité nommée `feature/heat-opt` dans laquelle il cherche à optimiser le calcul numérique dans `heat_seq.c` :

1. Détection d'une régression avec le widget « Test summary » Supposons que lors de la refactorisation de la boucle de calcul, une mauvaise gestion d'indice provoque l'échec de la vérification des conditions aux limites :

   - Le job *test* s'exécute et publie le rapport JUnit `build/ctest.xml`.
   - Dans la page de la Merge Request, GitLab affiche automatiquement un encart dédié :

     ```
     Test summary: 1 failed test, 2 passed (1 new failure)
     Failed: ctest: heat_seq_test (new)
     ```

   - En cliquant sur l'échec directement dans le widget, le relecteur (ou l'auteur) accède au nom exact du test et à sa trace d'erreur, sans avoir à naviguer dans les logs bruts du runner.

2. Contrôle de la couverture avec le widget « Code coverage » et le diff en ligne Supposons maintenant que le développeur ajoute une nouvelle fonction de validation des paramètres d'entrée dans `heat_seq.c`, mais omet d'ajouter le test unitaire correspondant :

   - **Dans le widget de la MR** : GitLab calcule la différence de couverture globale :

     ```
     Test coverage 52.3% (-3.8% compared to master 56.1%)
     ```

     Le relecteur est immédiatement alerté que la contribution dégrade la qualité globale des tests.
   - **Directement dans l'onglet Changes (diff du code)** : GitLab exploite le rapport Cobertura `build/coverage.` pour colorer la marge gauche du code source :
     - **Liseré vert** : la ligne de code ajoutée ou modifiée a été exécutée par la suite de tests.
     - **Liseré orange ou rouge** : la ligne n'a jamais été traversée lors de l'exécution des tests.

     Le relecteur peut ainsi commenter directement : *« Merci d'ajouter un test pour couvrir cette nouvelle branche conditionnelle. »*. Dès que le développeur ajoute le test adéquat, le liseré bascule au vert et la métrique de couverture remonte.

2.3.2 Intégration du rapport « Code Quality » sur le projet Heat

GitLab prend en charge nativement le format de rapport Code Quality standardisé par Code Climate. Cela permet aux analyseurs statiques de signaler les défauts de conception, problèmes de portée de variable ou fuites mémoire directement sous forme d'annotations dans les Merge Requests.

1. Format JSON attendu par GitLab GitLab s'attend à un tableau d'objets JSON au format suivant :

   ```
   [
     {
       "description": "The scope of the variable 'i' can be reduced.",
       "check_name": "variableScope",
       "fingerprint": "9a0f4a8...",
       "severity": "minor",
       "location": {
         "path": "heat_seq.c",
         "lines": {
   "begin": 45
         }
       }
     }
   ]
   ```

   Les sévérités reconnues par GitLab sont : `info`, `minor`, `major`, `critical`, `blocker`.

2. Conversion du rapport Cppcheck vers le format Code Quality Comme nous disposons déjà de l'outil **Cppcheck** qui analyse le code C de Heat et génère un rapport XML (`heat-cppcheck.xml`), il suffit de convertir ce XML au format JSON attendu par GitLab.

   Vous pouvez créer un script léger `tools/cppcheck_to_codequality.py` :

   ```
   #!/usr/bin/env python3
   import sys
   import json
   import hashlib
   ```

```
import xml.etree.ElementTree as ET

if len(sys.argv) < 3:
    print(f"Usage: {sys.argv[0]} <cppcheck.xml> <gl-code-quality-report.json>")
    sys.exit(1)

xml_file, json_file = sys.argv[1], sys.argv[2]
tree = ET.parse(xml_file)
root = tree.getroot()

severity_map = {
    "error": "critical",
    "warning": "major",
    "style": "minor",
    "performance": "minor",
    "portability": "minor",
    "information": "info"
}

issues = []
for error in root.findall(".//error"):
    error_id = error.get("id", "cppcheck")
    msg = error.get("msg", "")
    severity = severity_map.get(error.get("severity", "style"), "minor")

    for location in error.findall("location"):
file_path = location.get("file", "")
line = int(location.get("line", 1))

# Génération d'une empreinte unique pour le problème
fingerprint = hashlib.md5(f"{file_path}:{line}:{error_id}".encode()).hexdigest()

issues.append({
    "description": msg,
    "check_name": error_id,
    "fingerprint": fingerprint,
    "severity": severity,
    "location": {
"path": file_path,
"lines": {
    "begin": line
}
    }
})

with open(json_file, "w", encoding="utf-8") as f:
    json.dump(issues, f, indent=2)
print(f"Exporté {len(issues)} problèmes vers {json_file}")
```

(Alternative sans script) : Vous pouvez également installer, dans votre environment Python (et/ou dans l'image Docker), le paquet officiel :

```
pip install cppcheck-codequality
cppcheck-codequality --input heat-cppcheck.xml --output gl-code-quality-report.json
```

3. Configuration dans `.gitlab-ci.yml` Ajoutez la génération et l'exportation du rapport Code Quality dans votre pipeline :

```
code_quality:
  stage: test
```

```
image: gitlab-ce.iut.u-bordeaux.fr:5050/but-devops/automatisation/analyse
script:
  - cppcheck -v -f --language=c --enable=all --xml --xml-version=2 --suppress=missingIncludeSyst
  - python3 tools/cppcheck_to_codequality.py heat-cppcheck.xml gl-code-quality-report.json
artifacts:
  reports:
    codequality: gl-code-quality-report.json
```

4. Bénéfice visuel dans la Merge Request Grâce à cet artefact :
   - Le widget Code Quality de la MR affiche : *« Code Quality: 2 issues found (1 major, 1 minor) »*.
   - Dans l'onglet **Changes**, une alerte s'affiche directement sous la ligne incriminée :

     `heat_seq.c:L42 - Variable 'u_tmp' is allocated but never freed [memleak] (major)`
   - Dès que le développeur corrige la fuite et pousse son commit, le widget se met à jour : *« Code Quality: Fixed 2 issues »* !

3 Analyse plus poussée avec SonarQube

3.1 Généralités

SonarQube est un outil d'analyse de la qualité d'un logiciel. Un programme nommé `sonar-scanner` est lancé sur la machine cible permettant de faire tourner le logiciel, à la racine du code source afin de réaliser l'analyse, ensuite les rapports de bugs et diverses métriques sont téléversés vers un serveur où un tableau de bord web permet de visualiser et partager la qualité et la maturité du logiciel.

Plusieurs éléments peuvent être rapportés : taille du code source, détection de duplication, problèmes basés sur des analyses de code statiques (recherche de bugs potentiels sur la connaissance du code source) et dynamiques (exécution du binaire).

Plusieurs versions de SonarQube sont distribuées : une libre (open source, gratuite) "Community" avec une limitation sur certaines fonctionnalités, et d'autres payantes (voir Plans & Pricing). SonarCloud est aussi une instance disponible en ligne gratuitement pour les projets libres.

Langages supportés : C, C++, C#, CSS, Flex, Go, HTML, Java, JavaScript, Kotlin, PHP, Python, Ruby, Scala, Swift, XML, etc., voir list of supported languages. Certaines fonctionnalités additionnelles peuvent être obtenues via un système de plugins.

3.2 Analyse avec SonarQube

3.2.1 Image docker de SonarQube

L'application SonarQube peut être instanciée à partir d'une image docker. Ce qui suit est basé sur le guide officiel pour un système Linux : installing sonarqube from docker.

Récupérer l'image sonarqube:lts-community qui a été copiée de docker.io sur le registre de l'Université de Bordeaux et démarrer l'application :

```
podman run -d --name sonarqube \
       -p 9000:9000 \
       -v sonarqube_data:/opt/sonarqube/data \
       -v sonarqube_extensions:/opt/sonarqube/extensions \
       -v sonarqube_logs:/opt/sonarqube/logs \
       registry.u-bordeaux.fr/vhub/sonarqube:lts-community
```

Attention : normalement, les volumes créés devraient être conservés entre 2 sessions, mais ici ce n'est pas le cas, car ils sont montés par défaut dans *tmp/containers (cf.* ~.config/containers/storage.conf) pour des raisons d'optimisation des quotas utilisateurs. Généralement, les volumes sont montés dans ~/.local/share/containers/storage, un dossier utilisateur persistant.

Cela implique que toute la configuration qui suit ne sera utilisable que le temps de votre session Ubuntu actuelle. Si vous quittez la session, il faudra recommencer la configuration. Ou alors, vous créez une sauvegarde de vos données SonarQube **juste avant de fermer votre session Linux** :

```
mkdir -p ~/sonarqube_data
podman volume export sonarqube_data -o ~/sonarqube_data/data.tar
podman volume export sonarqube_extensions -o ~/sonarqube_data/extensions.tar
podman volume export sonarqube_logs -o ~/sonarqube_data/logs.tar
```

afin de les réimporter lors d'une prochaine connexion :

```
podman volume create sonarqube_data
podman volume create sonarqube_extensions
podman volume create sonarqube_logs

podman volume import sonarqube_data ~/sonarqube_data/data.tar
podman volume import sonarqube_extensions ~/sonarqube_data/extensions.tar
podman volume import sonarqube_logs ~/sonarqube_data/logs.tar

podman run -d --name sonarqube -p 9000:9000 -v sonarqube_data:/opt/sonarqube/data -v sonarqube_ext
```

Ouvrir une page internet à l'adresse : `http://localhost:9000/`. Il faut changer le mot de passe, celui par défaut est admin:admin.

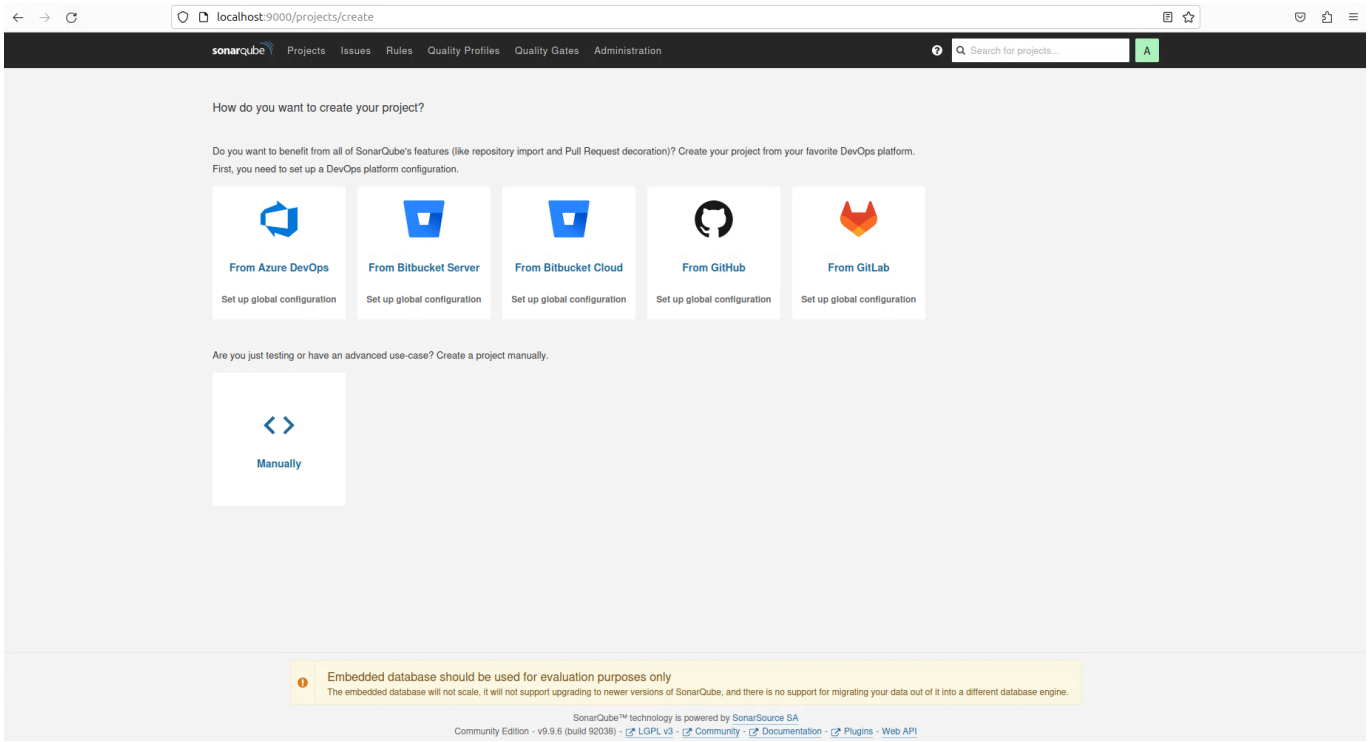


Figure 3: Page d'accueil

Il faut créer un jeton côté SonarQube pour permettre d'importer les résultats d'analyse de code avec le programme `sonar-scanner`. Créer un jeton personnel (User) "heat", voir **My Account -> Security -> Generate Tokens** en haut à droite, le sauver dans un fichier et une variable :

```
echo "squ_blablabla-1234567890" > $HOME/.sonartoken
export SONAR_TOKEN=‘cat $HOME/.sonartoken‘
```

Vous pouvez aussi ajouter cet export dans votre script `~/setenv-podman.sh`.

Par la suite, nous allons analyser du code écrit en C/C++. Par défaut, la version "Community" de SonarQube ne prend pas en charge ce langage. Il faut installer le plugin sonar-cxx (version 2.1.1 compatible avec SonarQube 9.9 LTA) :

```
podman exec sonarqube bash -c 'wget https://gitlab-ce.iut.u-bordeaux.fr/api/v4/projects/14291/packages/
# source du plugin sur github : https://github.com/SonarOpenCommunity/sonar-cxx/releases/download/cxx-2
podman restart sonarqube
```

De retour sur `http://localhost:9000/`, se connecter, puis cliquer sur "I understand the risk".

Si vous voulez voir les données des volumes :

```
podman exec sonarqube bash -c 'ls "$SONARQUBE_HOME"/data/*'
podman exec sonarqube bash -c 'ls "$SONARQUBE_HOME"/extensions/*'
podman exec sonarqube bash -c 'ls "$SONARQUBE_HOME"/logs/*'
```

3.2.2 Environnement d'analyse

Pour l'analyse statique et dynamique, on va avoir besoin d'un certain nombre d'outils. Plutôt que de les installer directement dans l'environnement natif, on va se servir d'une image docker. La recette est ici : dockerfile-sonar-scanner.

Télécharger l'image préconstruite (bien être authentifié sur le registre GitLab cf. setenv-podman.sh) :

```
podman pull gitlab-ce.iut.u-bordeaux.fr:5050/but-devops/automatisation/analyse
podman tag gitlab-ce.iut.u-bordeaux.fr:5050/but-devops/automatisation/analyse scanner
```

À présent, vous pouvez instancier un conteneur avec tous les outils nécessaires à l'analyse SonarQube (sonar-scanner, cppcheck, clang, gcov, valgrind, etc.) :

```
podman run -it --name scanner --network="host" -e GITLAB_USER=$USER -e GITLAB_TOKEN=$TOKEN -e SONAR_TOK
```

3.2.3 Hello World SonarQube

Dans le conteneur précédemment instancié, exécuter une analyse très simple, un "Hello World" en Python.

Créer un script Python comme ceci :

```
cd /builds
mkdir -p hello/
cd hello/
echo 'print("Hello, world!")' > hello_world.py
```

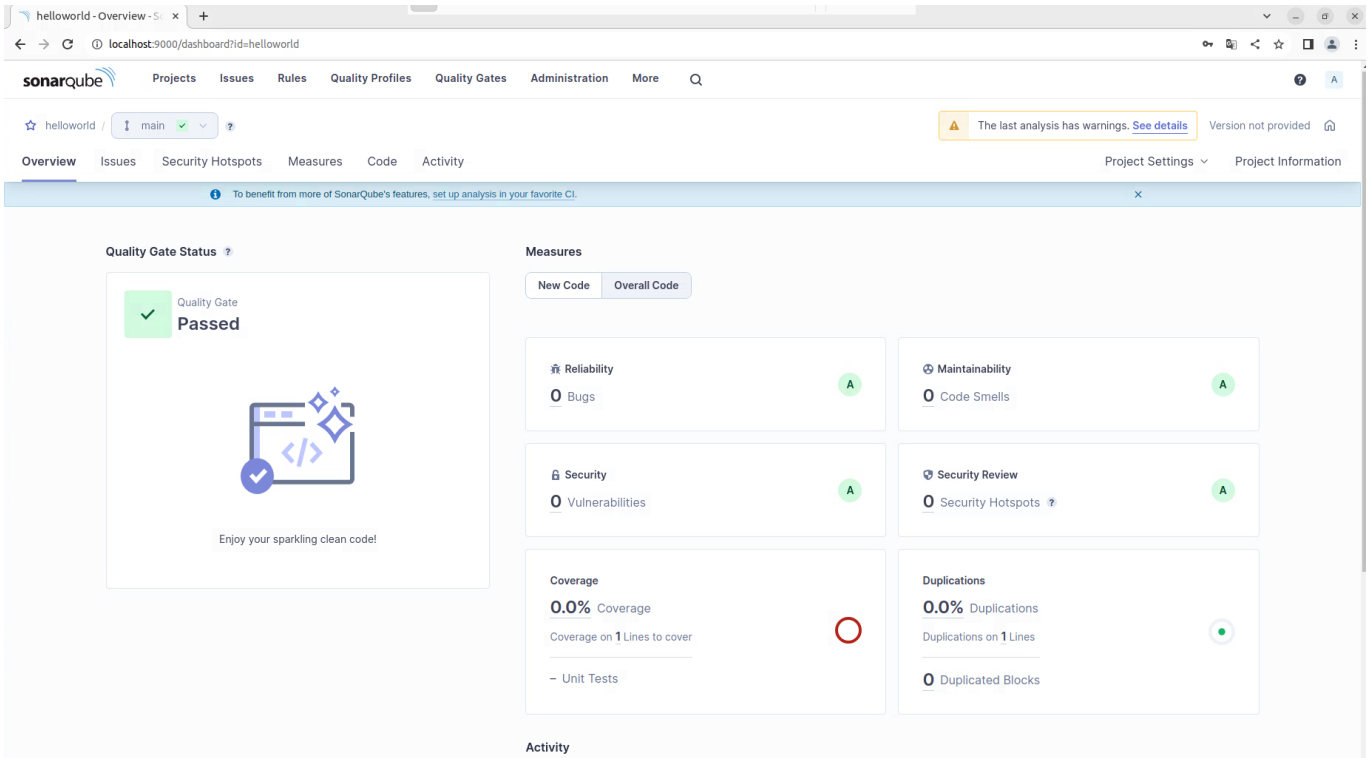
Tester l'exécution :

```
python hello_world.py
```

Maintenant, utilisons **sonar-scanner** pour analyser ce code source :

```
sonar-scanner -D"sonar.host.url=http://localhost:9000" -D"sonar.login=$SONAR_TOKEN" -D"sonar.projectKey
```

Vous devriez voir `INFO: EXECUTION SUCCESS`. Visitez la page SonarQube projects et explorez le projet soumis "helloworld".



Remarquez les deux panneaux "New Code" et "Overall Code". La partie New Code souligne les mesures sur les nouvelles lignes de code par rapport à un état précédent, qui peut être défini de plusieurs manières :

- soit par rapport à la toute **première analyse**,
- ou par rapport au **paramètre** `sonar.projectVersion`,
- ou par rapport à l'état du code sur une **autre branche git**.

Vous pouvez parcourir les différents onglets, notamment Code qui donne la liste des fichiers analysés et quelques métriques associées.

sonar-scanner est utilisé dans un terminal via une interface en ligne de commande (CLI). On peut lui passer des paramètres en ligne de commande ou les enregistrer dans un fichier de configuration `sonar-project.properties` :

```
sonar.host.url=http://localhost:9000
sonar.login=squ_blablabla-1234567890
sonar.projectKey=helloworld
sonar.sources=./hello_world.py
```

Ainsi, vous pouvez lancer l'analyse simplement comme ceci :

```
sonar-scanner
```

Notez que les valeurs par défaut sont celles définies dans le fichier, celles passées à la commande complètent ou remplacent celles du fichier (si la même variable est renseignée deux fois).

3.3 Scan complet du programme Heat (C/C++) avec des analyseurs externes

Expérimentons l'analyse statique et dynamique sur le code Heat écrit en C, qui fournit une implémentation basique pour résoudre l'équation de la chaleur.

L'analyse statique tend à trouver des problèmes dans le code source simplement basée sur les fichiers texte et révèle de possibles erreurs de syntaxe, variables non initialisées lues, déréférencements de pointeurs, fuites de mémoire, duplications, réduction de portée, non-respect de conventions, etc.

L'analyse dynamique permet de vérifier un programme en exécutant les binaires. Les tests unitaires et d'intégration font partie de l'analyse dynamique, tout comme les mesures de couverture, la vérification mémoire, la vérification des threads, etc.

3.3.1 Télécharger Heat avec git, compiler Heat et lancer le script d'analyse

Cloner le code Heat dans votre conteneur :

```
cd /builds
git clone https://$GITLAB_USER:$GITLAB_TOKEN@gitlab-ce.iut.u-bordeaux.fr/$GITLAB_USER/heat.git
cd heat/
```

Lancer l'analyse complète :

```
HEAT_ROOT=$PWD ./tools/build-test.sh
./tools/analysis.sh
```

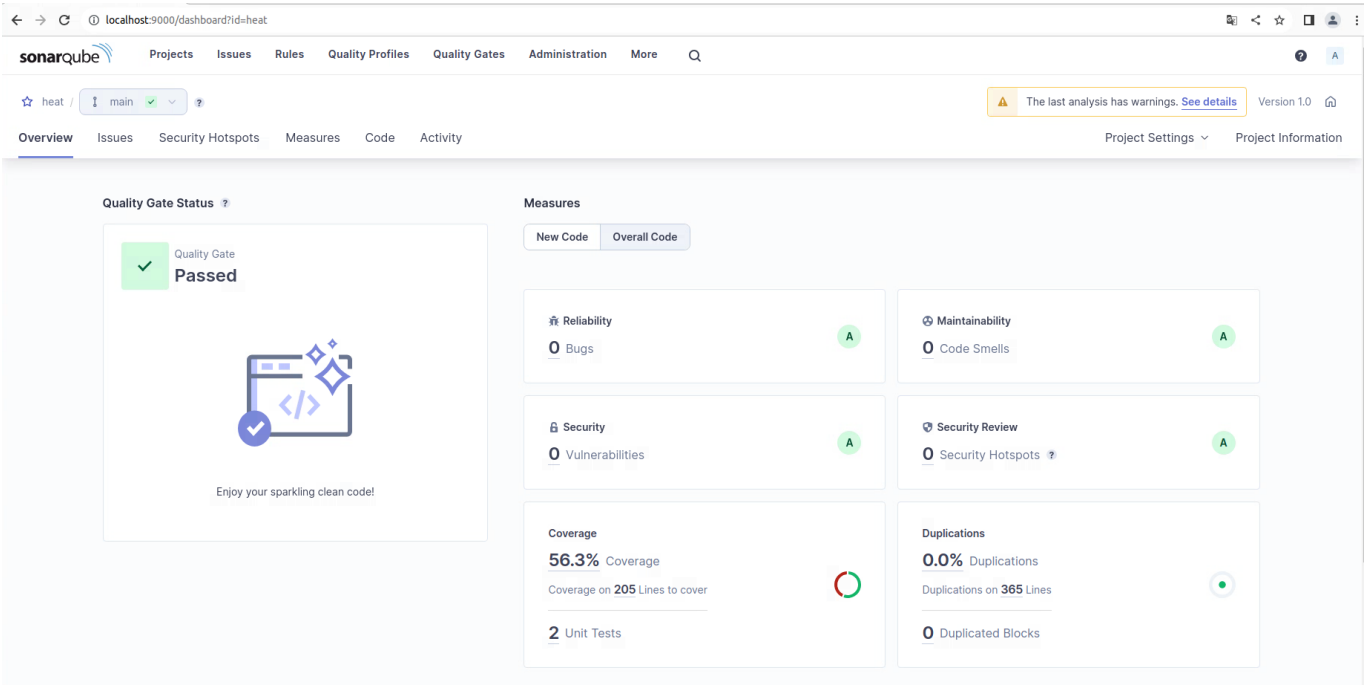
Un certain nombre de rapports sont générés :

- build/heat-build.log : gcc warnings
- build/analyzer_reports/ : clang-sa
- clang-tidy-report : clang-tidy
- heat-cppcheck.xml : cppcheck
- heat-rats.xml : rats
- heat-vera.xml : vera++
- heat-valgrind.xml : valgrind
- heat-junit.xml : xunit report for C/C++
- heat-coverage.xml : coverage C/C++
- plot_junit.xml : xunit report for Python
- plot_coverage.xml : coverage Python

Transmettre les rapports d'analyse sur le serveur SonarQube :

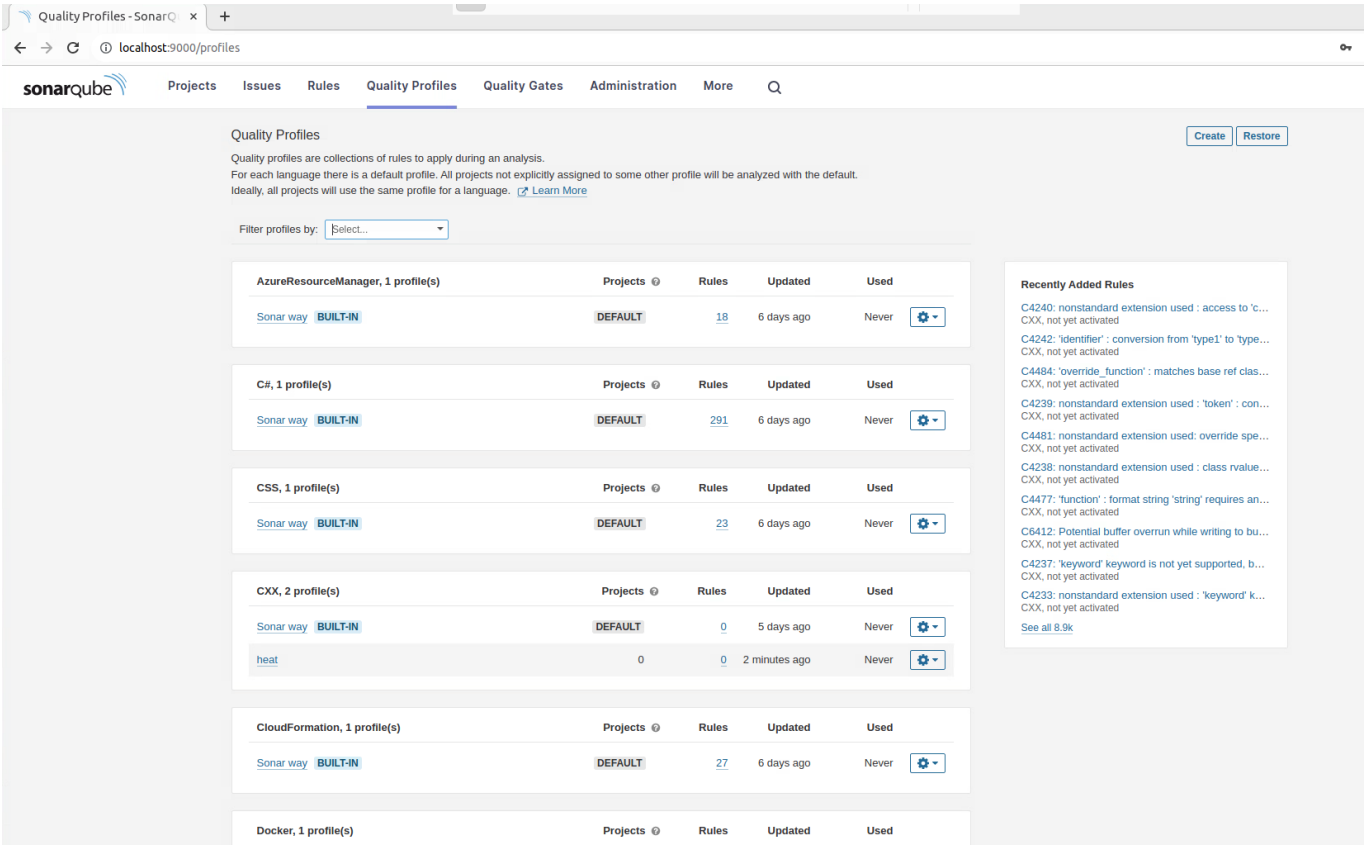
```
sonar-scanner -X -Dsonar.host.url=http://localhost:9000 -Dsonar.login=$SONAR_TOKEN 2>&1 |tee sonar.log
```

Pour le moment peu de problèmes sont visibles côté SonarQube, car par défaut les règles C/C++ ne sont pas actives. Nous allons activer les règles à surveiller dans un Quality Profile pour Heat dans la section suivante.

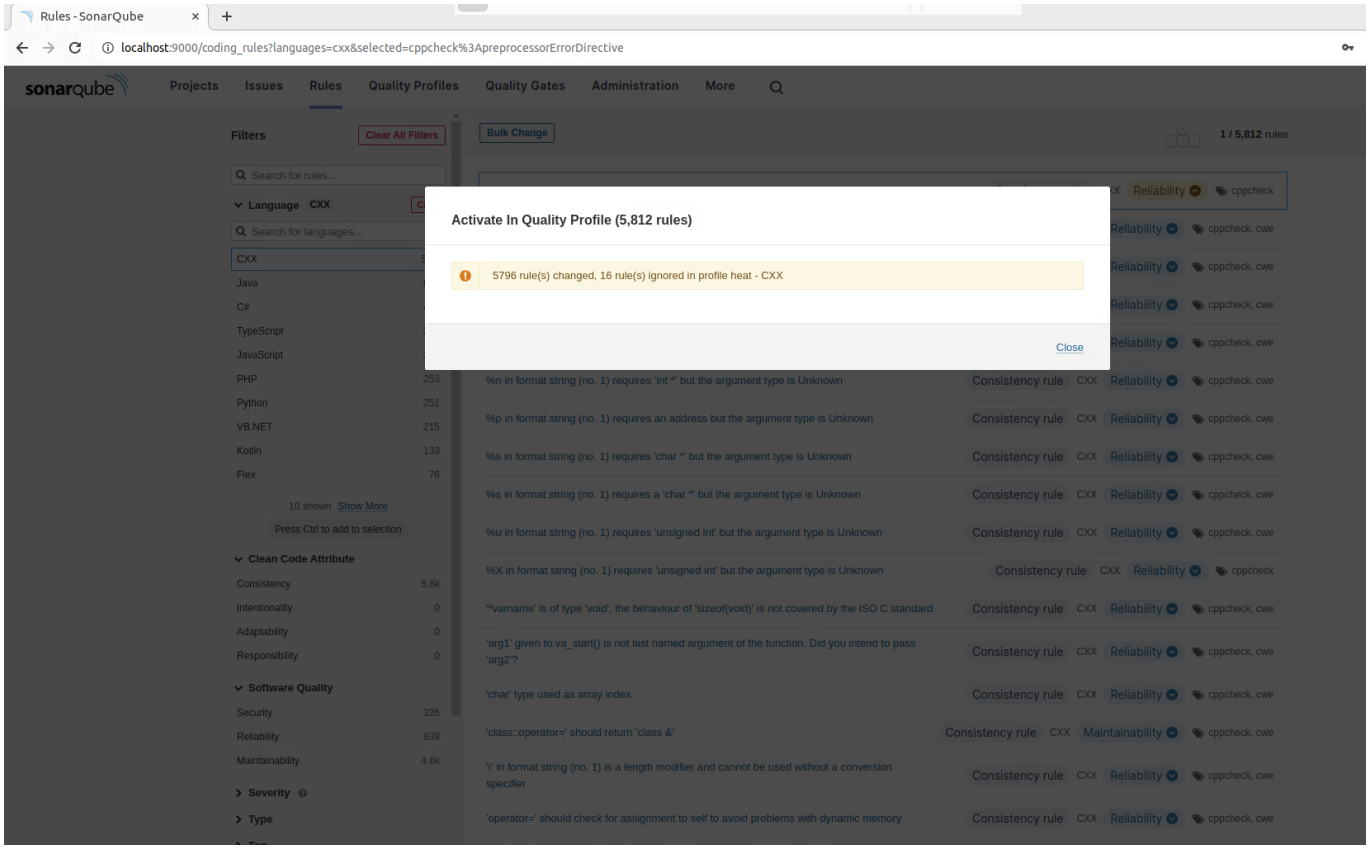


3.3.2 Créer un Quality Profile

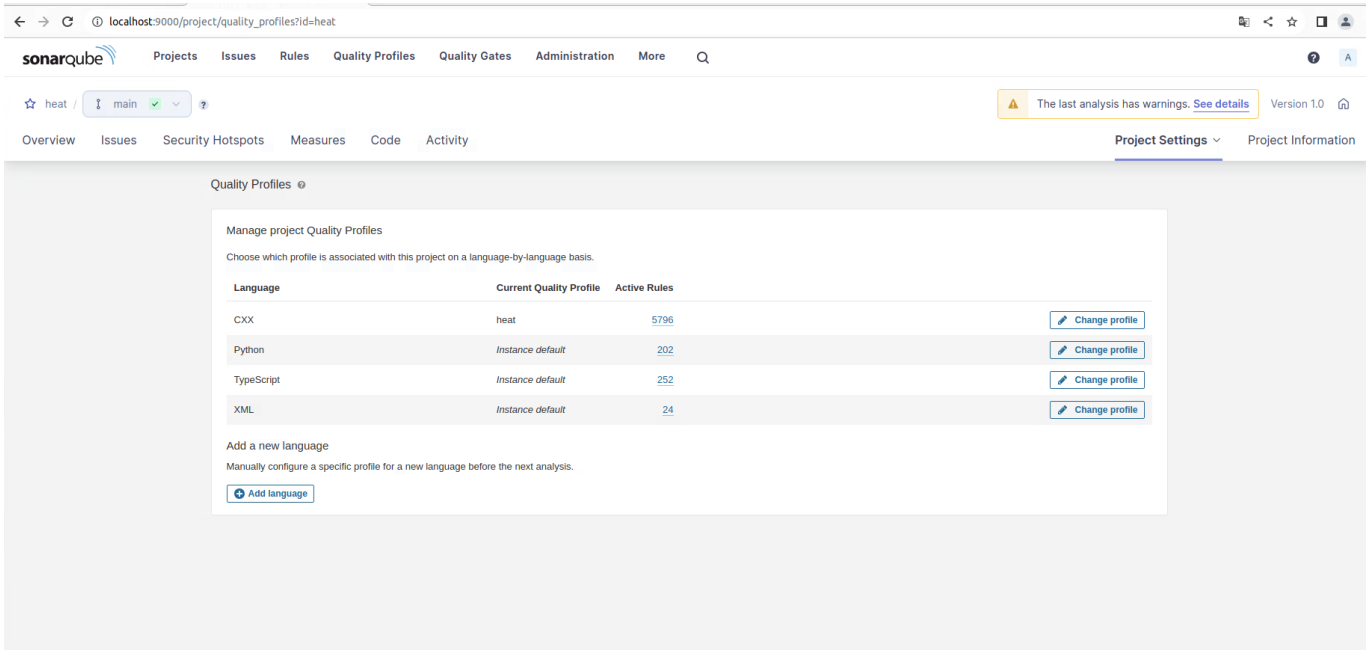
Nous avons besoin de choisir les règles à activer. Créer un **Quality Profile** (QP). Aller sur la page **Quality Profiles**, chercher le profil existant dans le langage CXX "Sonar way" et le copier en tant que "heat".



Dans ce nouveau QP, cliquer sur **Activate More**, enlever les filtres existants, sélectionner le langage CXX dans les filtres, et cliquer sur **Bulk Change -> Activate In ... -> heat - CXX**. Cela va activer l'ensemble des règles disponibles.



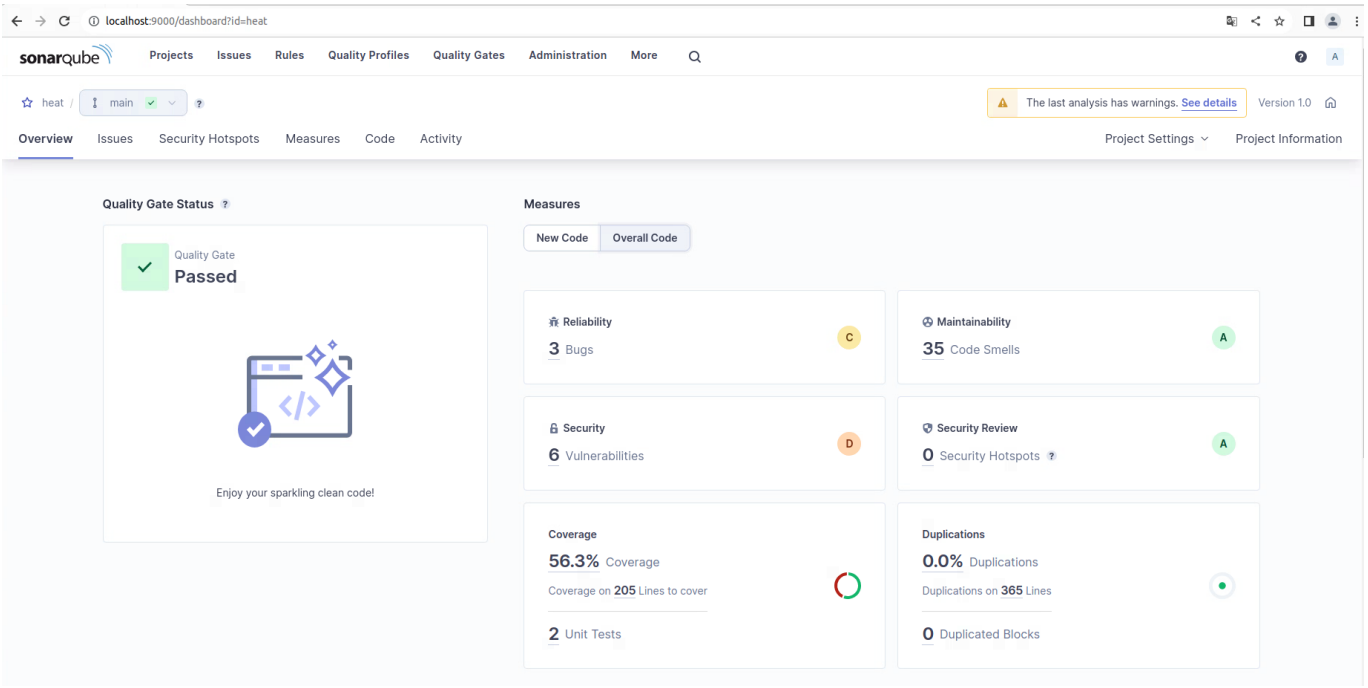
Par défaut, votre projet utilise le Quality Profile "Sonar way". Changez cela en allant dans les "Project Settings" du projet Heat et modifiez le QP à utiliser pour le langage CXX : "Sonar way" -> "heat".



Retransmettre les rapports d'analyse sur le serveur SonarQube :

```
sonar-scanner -X -Dsonar.host.url=http://localhost:9000 -Dsonar.login=$SONAR_TOKEN 2>&1 |tee sonar.log
```

Vous devriez voir apparaître des problèmes dans l'onglet Overview->Overall Code.



Parcourir les issues de type Bugs par exemple.

3.4 Étude qualitative du code source en utilisant SonarQube

La documentation SonarQube.

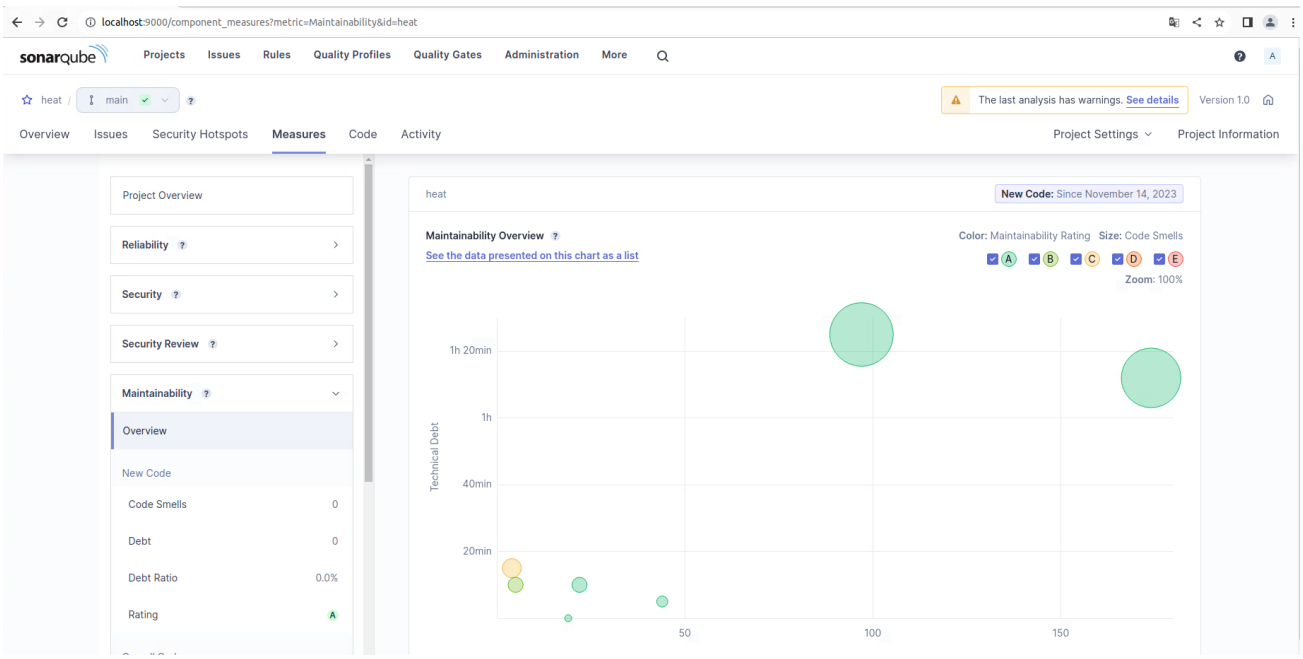
3.4.1 Overview

Une première question à se poser est la question de la visibilité du projet. Est-ce que le projet est libre et donc ouvert, ou privé, avec une licence ? Est-ce que vous souhaitez partager les mesures de SonarQube avec votre communauté ? Si oui, allez dans Project Settings -> Permissions et positionnez le projet sur "Public". Ainsi, le projet pourra être examiné par tous, sans authentification. Bien sûr, au sein d'une entreprise, le serveur pourrait être accessible aux employés connectés au réseau d'entreprise, mais fermé au reste du monde.

Ensuite, vous avez deux onglets **New Code** et **Overall Code** qui présentent les principales métriques telles que le nombre de Bugs (robustesse), Vulnerabilities (aspects sécurité), Code Smells (encouragent le respect des standards), Unit tests, Coverage, Duplications.

Chaque mesure est cliquable, on peut explorer la liste des :

- **issues**
- **measures** : métriques et graphes résumant la situation
  - cercles en haut à droite : fichiers à prioriser et corriger
  - cercles en bas à gauche : plutôt sains



- **activity** : tendances historiques de l'évolution des mesures après plusieurs analyses

La Quality Gate (QG) informe de façon binaire (OK, Not OK) si la qualité du code est pire qu'avant. Par exemple, si de nouveaux bugs apparaissent entre deux analyses sur du nouveau code ou si la couverture de code devient inférieure à 80 %, alors la QG devient rouge et les développeurs sont notifiés pour corriger le nécessaire et faire en sorte de faire repasser la QG en vert.

La notion de nouveau code (New Code) sert à pouvoir mesurer l'évolution de la qualité du logiciel. Elle peut être définie de différentes manières : comparaison entre deux branches, comparaison par rapport à la dernière release. On peut indiquer à SonarQube le numéro de version en cours afin de définir le nouveau code avec *sonar.projectVersion*. Si on passe de 1.0 à 2.0 alors, pour les analyses suivantes, le nouveau code considéré sera celui injecté depuis la release 1.0. Voir defining new code.

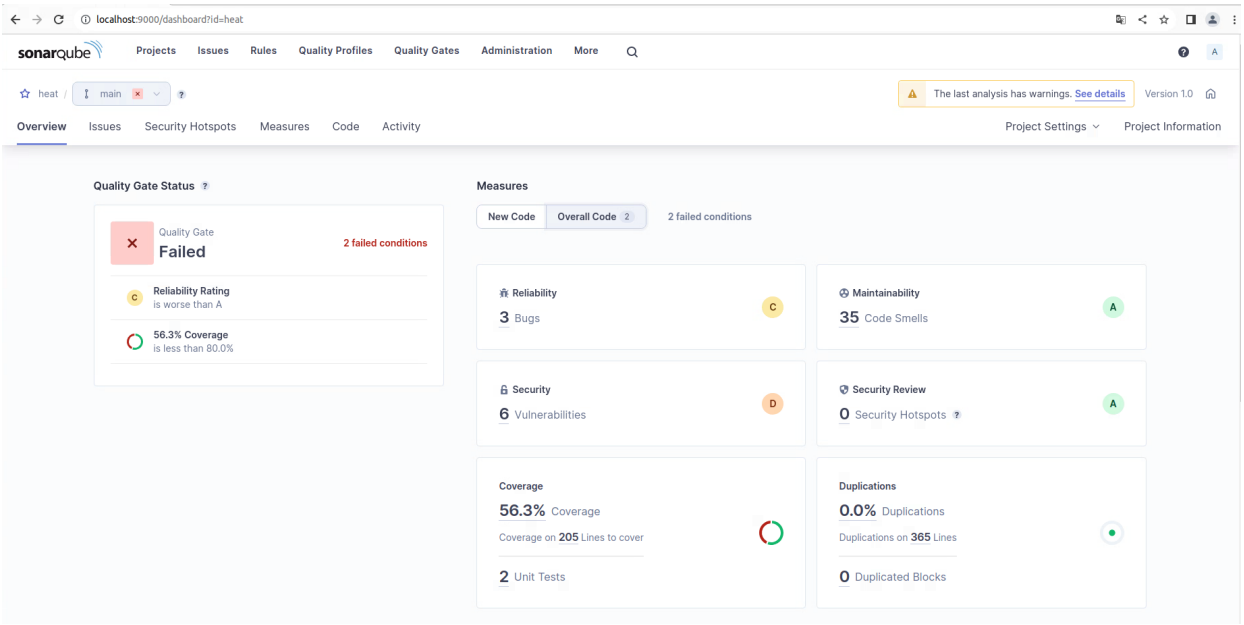
1. Exercices

   (a) Quelles sont les notes sur l'ensemble du code pour la robustesse, la sécurité et la maintenabilité ?

   (b) Créez une nouvelle Quality Gate "heat" avec en condition sur l'"Overall Code" :

   - *Coverage is less than 80%*,
   - *Reliability Rating is worse than A*.

   Appliquez cette nouvelle QG sur le projet Heat et réexécutez le scan : le statut de la Quality Gate est Failed, car la couverture est aux alentours de 50 % et puisqu'il y a des bugs.



3.4.2 Measures

La section mesures permet d'avoir une vue synthétique de la qualité du code fichier par fichier avec des graphes et différentes vues dossiers/sous-dossiers (bugs, vulnérabilités, couverture, duplications, tailles, complexité). Voir aussi metric definitions.

1. Exercices

   (a) Sur le graphe Project Overview, quelle est la signification de la couleur et de la taille des cercles pour chaque fichier ?

   (b) Quel fichier devrait être considéré en premier pour améliorer la maintenabilité ?

   (c) Quel fichier important semble ne pas être couvert par les tests unitaires (voir Coverage -> Overall -> View as: Treemap) ?

   (d) Quel langage est principalement analysé (Size -> Lines of code), et combien de lignes de code pour chaque langage ?

   (e) La densité de duplication vous semble-t-elle correcte (saine) ?

3.4.3 Issues

En cliquant sur Issues, on obtient une liste de tous les problèmes relevés (bugs, vulnerabilities, code smells). Le panneau de gauche permet de filtrer par type d'issue.

Cette page permet de vérifier les potentiels problèmes détectés par les différents outils (sonar-scanner, avertissements GCC, clang-sa, clang-tidy, cppcheck, valgrind, etc.) et de les gérer. Faut-il corriger ces problèmes ? Vérifier les faux positifs ? Par défaut, les nouveaux problèmes ont un statut *Open*. Le but est de réduire le nombre de problèmes ouverts. La première chose à faire dans l'interface est de choisir quelles erreurs corriger dans un futur proche ou non :

1. Les erreurs à corriger rapidement sont étiquetées *Confirmed*. Si dans la prochaine analyse l'erreur n'est plus visible dans les rapports, alors SonarQube va automatiquement la basculer en *Fixed* et *Closed*.

2. Les autres erreurs peuvent être classées différemment. Pourquoi ne pas corriger rapidement ces erreurs ? Est-ce parce que :

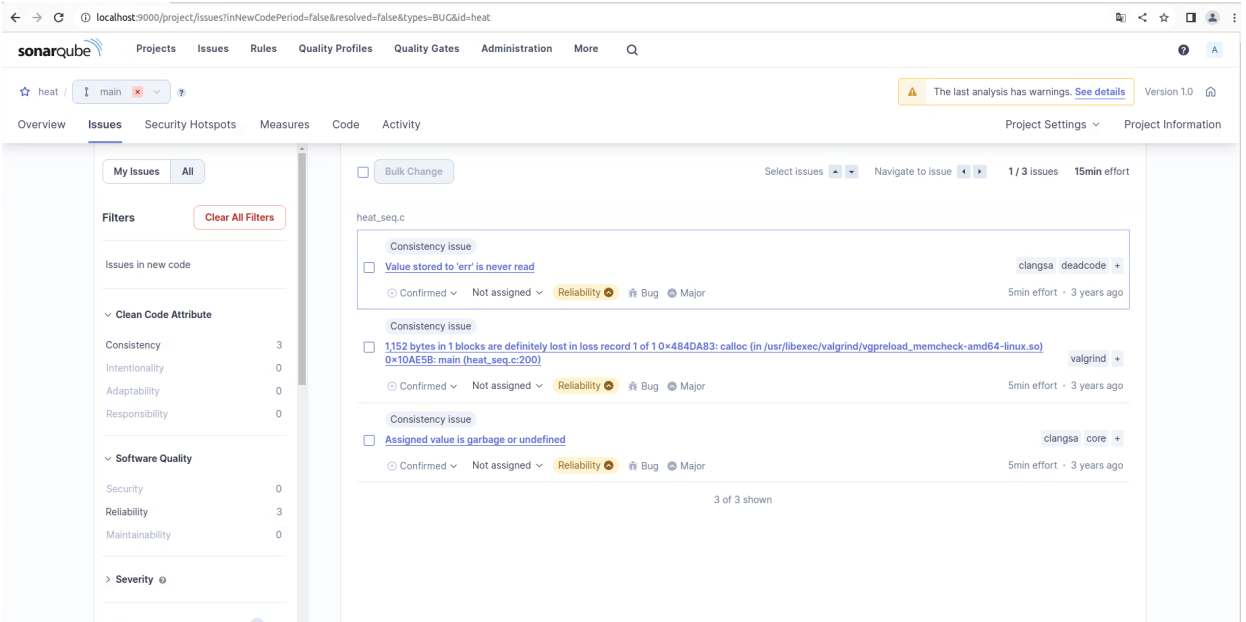
   - c'est un faux positif, à basculer dans la catégorie *false positive*
   - l'erreur est connue, mais on souhaite conserver le code tel quel, car on a de bonnes raisons, à étiqueter *won't fix*

Remarque : les erreurs *Closed* sont gardées 30 jours dans la base de données.
Il y a de nombreuses façons de filtrer les erreurs via le panneau de gauche :

- Type : Bug, Vulnerabilities, Code Smells
- Resolution and Status : Unresolved, Fixed, Open, Closed, ...
- Rule : nom de la règle
- Directory, File
- Assignee, Author
- Language

1. Exercices Le but est de corriger quelques problèmes dans Heat. Vous allez devoir éditer et modifier des fichiers et réexécuter l'analyse.

   (a) Filtrez les issues de type Bugs (voir Software Quality -> Reliability) et étiquetez-les **Confirm**. Ensuite, commencez par corriger celles que vous pouvez parmi les trois avant de soumettre une nouvelle analyse.

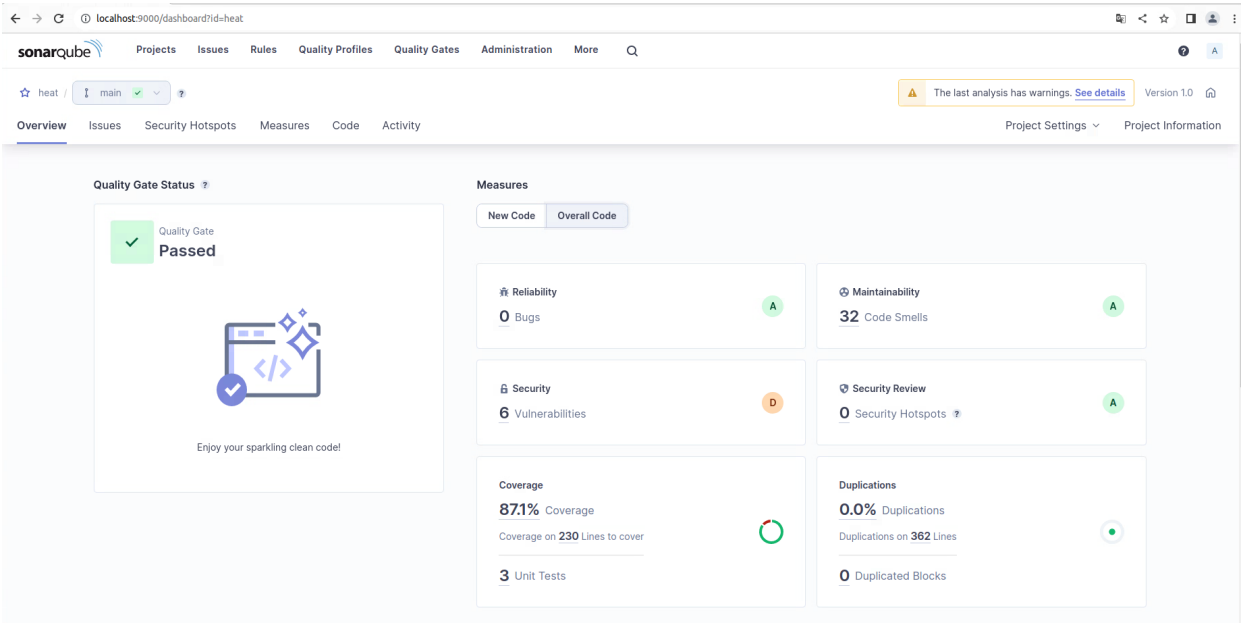


   - Corriger l'issue "Value stored to 'err' is never read"
   - Corriger l'issue "Assigned value is garbage or undefined"
   - Enlever aussi l'allocation qui n'est jamais libérée `xalloc (u_tmp, ...)`
   - Corriger la macro `xalloc`

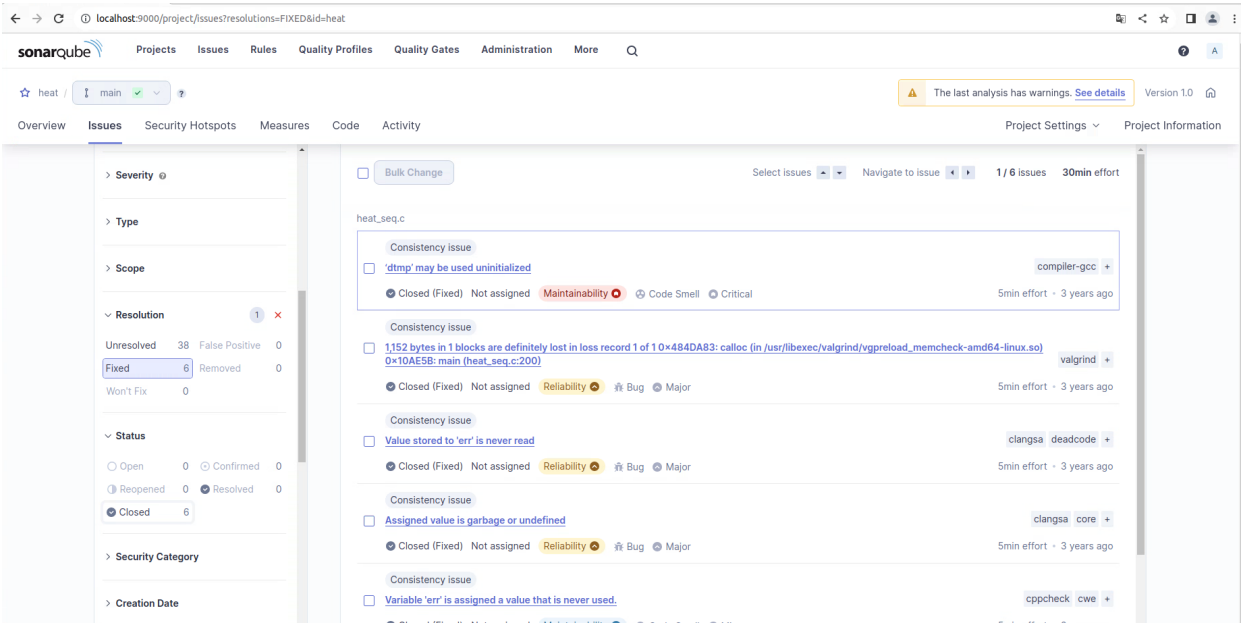
     Note : l'éditeur de texte **vim** est installé dans le conteneur. Sinon, commitez vos modifications dans une branche (ex. : `auto`) et faites `git pull` dans le conteneur *scanner*. Vous pouvez utiliser les scripts existants pour refaire toute l'analyse sur le nouveau code source :

```
HEAT_ROOT=$PWD ./tools/build-test.sh
./tools/analysis.sh
sonar-scanner -X -Dsonar.host.url=http://localhost:9000 -Dsonar.login=$SONAR_TOKEN 2>&1 |te
```

(b) Essayez maintenant de corriger votre Quality Gate. Pour cela, il faut une couverture > 80 %. Vous avez sûrement remarqué que le fichier `heat_par.c` n'est pas couvert par les tests. Pour activer cette partie du code, il faut activer l'option MPI : éditez le fichier `tools/build-test.sh` et activez MPI (cf. `-DHEAT_USE_MPI=ON`). Après une nouvelle analyse complète, la QG devrait être validée.



(c) Essayez de filtrer par "Rule" les erreurs "Warn when a variable is unused" ou "The scope of the variable can be reduced" (cppcheck) et réglez-les. Une fois corrigées, vous devriez les voir comme *Fixed* dans les Issues avec le filtre *Resolution=Fixed*.



3.4.4 Code

L'onglet Code permet de naviguer dans les répertoires et de voir des métriques fichier par fichier : taille, nombre d'issues, pourcentage de couverture et duplications.

3.4.5 Activity

L'onglet Activity permet d'avoir une vue sur l'évolution des métriques (bugs, vulnerabilities, code smells) dans le temps entre les versions (releases par exemple).

3.5 Automatiser l'analyse SonarQube dans votre pipeline GitLab CI

Dans une démarche d'amélioration continue de la qualité, vous pouvez maintenant automatiser ce processus d'analyse dans votre pipeline GitLab CI.

Dans votre fichier `.gitlab-ci.yml`, commentez les jobs existants *build* et *test*, ajoutez un stage **sonar** après **test**, et ajoutez le job suivant :

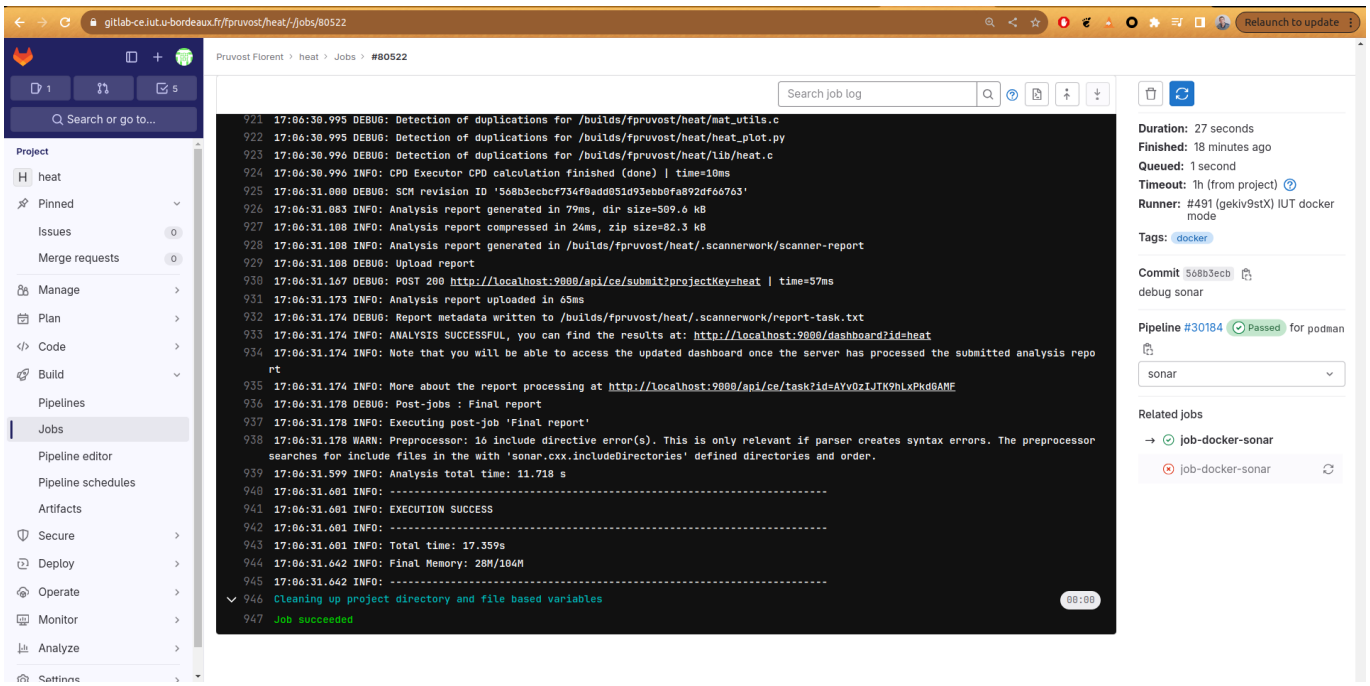
```
sonar:
  stage: sonar
  image: gitlab-ce.iut.u-bordeaux.fr:5050/but-devops/automatisation/analyse
  script:
    - HEAT_ROOT=$PWD ./tools/build-test.sh
    - ./tools/analysis.sh
    - sonar-scanner -X -Dsonar.host.url=http://localhost:9000 -Dsonar.login=$SONAR_TOKEN 2>&1 |tee sona
```

Il y a deux configurations préalables à faire avant de pousser ce nouveau job sur GitLab :

1. Pour que votre conteneur **podman** ait accès au serveur SonarQube local (`http://localhost:9000`), il faut ajouter `network_mode = "host"` dans votre fichier `$HOME/.gitlab-runner/config.toml` dans la section `[runners.docker]`. Redémarrez le runner avec `./gitlab-runner run` (ou `systemctl --user restart gitlab-runner`).

2. Il faut ajouter une variable d'environnement contenant le token SonarQube dans votre pipeline GitLab. Pour cela, allez dans la section Settings -> CI/CD -> Variables de votre projet GitLab Heat. Cliquez sur "Add variable", renseignez Key=SONAR_TOKEN et dans Value collez votre jeton SonarQube. Vous pouvez cocher *Masked* afin de masquer la valeur dans les journaux d'exécution du pipeline.

Vous pouvez commiter et pousser votre branche « auto » sur GitLab et vérifier que votre job d'analyse se lance avec succès.



1. Corrigez les issues SonarQube avec commit + push sur votre branche « auto ».

3.6 Bonus

3.6.1 Notifications par e-mail

Vous pouvez vous inscrire aux notifications par e-mail qui vous informent des nouveaux résultats d'analyse et donnent un lien vers les nouvelles issues du projet.

Allez dans "My Account" (coin supérieur droit) -> Notifications -> Notifications per project, écrivez le nom de votre projet dans la barre de recherche, sélectionnez-le puis cochez les cases.

3.6.2 Badges

Des badges peuvent être ajoutés dans certaines pages web afin d'afficher les mesures de qualité du code.

3.6.3 3D view

Consultez l'onglet More -> SoftVis3D Viewer si le plugin a été installé.

3.6.4 Serveur public SonarQube pour les logiciels libres et open source

L'analyseur C/C++ de SonarQube est un service payant (cf. SonarCFamily plugin). Ce plugin ne peut pas être utilisé dans la version communautaire que nous avons instanciée ici. Avec ce plugin, nous avons accès à quelques issues SonarQube (lors de l'analyse avec sonar-scanner) et nous pouvons importer de nombreux autres rapports provenant d'analyseurs externes. C'est bien, mais vous n'avez pas accès à l'analyseur « officiel » SonarQube C/C++ qui est sûrement intéressant lui aussi.

La bonne nouvelle : il existe un serveur SonarQube public, SonarCloud, gratuit pour les projets open source et qui donne accès au plugin SonarCFamily.

Si vous avez un projet open source à analyser et que vous voulez voir ce que donne l'analyse sonar-scanner avec le plugin SonarCFamily, vous pouvez consulter la documentation SonarSource.

Voir par exemple le projet SimGrid.

3.6.5 L'alternative Coverity

Coverity est un autre analyseur qui est normalement un service payant, mais qui donne accès à un serveur public pour les projets open source, voir :

- page principale
- connexion

Téléchargez l'application cov-build.

```
COVERITY_TOKEN="XEJaJ1cAnqW-9M_zkmxd7w"
wget --no-check-certificate https://scan.coverity.com/download/linux64 --post-data "token=$COVERITY_TOK
tar xvf coverity_tool.tgz
sudo ln -s -f $PWD/cov-analysis-linux64-*/bin/cov-build /usr/local/bin/cov-build
```

Construire et analyser le projet Heat avec la commande cov-build :

```
git clone https://gitlab-ce.iut.u-bordeaux.fr/but-devops/heat.git
cd heat/
mkdir -p build
cd build
cmake ..
make clean
cov-build --dir cov-int make -j 4
```

Créer une archive compressée des résultats :

```
tar czvf heat.tgz cov-int
```

Transmettre le rapport d'analyse sur le serveur Coverity :

```
curl --form token=$COVERITY_TOKEN --form email=florent.pruvost@inria.fr --form file=@heat.tgz --form ve
```

Consulter les résultats sur le dashboard Coverity, suivre le lien "View Defects".
Fonctionnalités :

- Service payant si le projet n'est pas open source
- Langages : C/C++, C#, Java, Javascript, PHP, Python, .NET Core, ASP.NET, Objective-C, Go, JSP, Ruby, Swift, Fortran, Scala, VB.NET, iOS, TypeScript
- Bien intégré dans les IDE, prend en charge la plupart des compilateurs
- Détection fine
  - des bugs (analyse statique)
  - des problèmes de concurrence (accès simultané aux données, interblocages, race conditions)
  - des problèmes de sécurité ...

4 Nettoyage

```
# sauvegarde des volumes de données sonarqube
mkdir -p ~/sonarqube_data
podman volume export sonarqube_data -o ~/sonarqube_data/data.tar
podman volume export sonarqube_extensions -o ~/sonarqube_data/extensions.tar
podman volume export sonarqube_logs -o ~/sonarqube_data/logs.tar

# arrêter le conteneur
podman stop sonarqube

# supprimer le conteneur
podman rm sonarqube

# supprimer l'image
podman rmi sonarqube

# nettoyer les volumes (efface la configuration courante de sonarqube: password, plugins, quality profi
# podman volume prune
```