TQS: Quality Assurance manual

***Alexandre Rodrigues [92951]*, *Diogo Bento [93391]*, *Pedro Laranjinha [93179]*, Pedro Silva [93011]**
v2021-06-23

1 Project management

1.1 Team and roles

DevOps	Diogo Bento NMEC 93391
Product Owner	Alexandre Rodrigues NMEC 92951
QA Engineer	Pedro Silva NMEC 93011
Team Lead	Pedro Casimiro NMEC 93179

1.2 Agile backlog management and work assignment

In order to organize our project according to agile method practices we utilize a **Pivotal Tracker** project for backlog management of user stories coupled with a **GitHub** extension in order to connect pull requests to user stories (and by doing this connect issues to user stories), for each user story a pull request is done.
Each user story is given a priority and story points and is put on the current iteration when work is being done on it.
After a pull request is done and closed, it is merged into our development branch.
Pivotal Tracker Project

2 Code quality management

2.1 Guidelines for contributors (coding style)

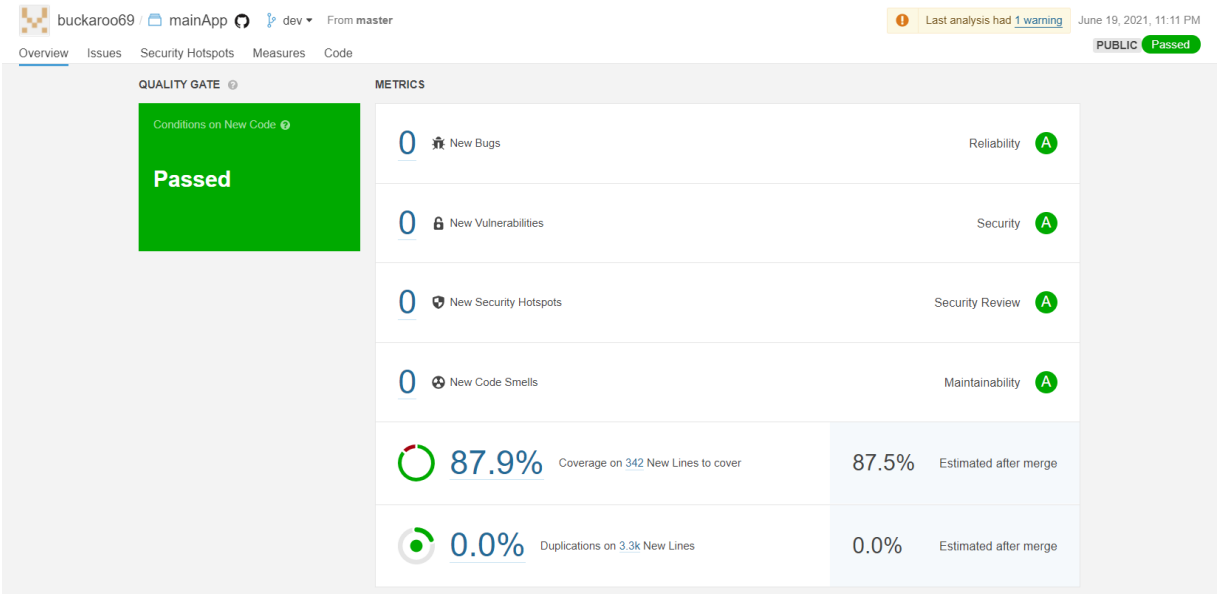
The coding style that was adopted will take into account the suggestions given by the **SonarQube** tool which allows us to analyze the style of the code and see its evolution over time.
All commits made to any branch that are put up to pull request must pass SonarQube quality gate requirements, which includes tests.

Other coding rules that should be followed (not following will end up with any pull request for review rejected):

- Don't ignore Exceptions, always use try catch constructs or throws statements in method used, in case of the second, justify in comments.
- Write short methods, any method above 40/50 lines of code will have to be well justified in its scope in comments, the LoC limit is not a hard limit and you may exceed if it implies you would need to make very small methods to compensate.
- Always describe what any method does, preferably do it by describing input, output, what's used and what Exception it may throw.
- Use TODO comments when needed.
- Use CamelCase for methods and classes.

2.2 Code quality metrics

Thanks to GitHub integration both projects are built and tested by using SonarQube when a commit happens, on top of that Docker images are also automatically generated when a commit to the "dev" or "master" branch happens.
We used Sonar Way, as our quality gate, followed by a variant that expects a coverage of at least 65%.
Code smells are to be removed as soon as they are found, and never be pushed to main.



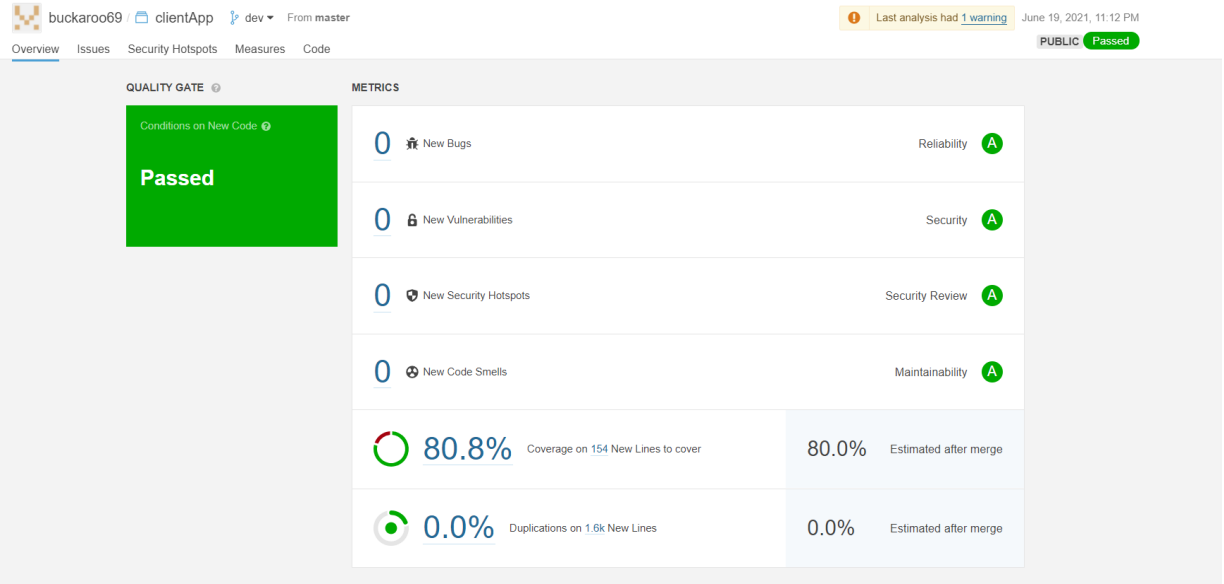


Fig 1,2: Quality Dashboards

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

We are using both a “dev” and a “master” branch, with other branches being related to user stories. Branches are merged into **dev** rather than **master**, which will be merged into whenever a major feature is implemented in a clean, sufficiently tested manner that matches our definition of done.
For a user story to be considered done it must be approved by the team, implemented cleanly, with all major behaviors being tested for and passing successfully, both in unit tests and in integration tests, and it must’ve been rolled out to the production environment without negative side effects.

3.2 CI/CD pipeline and tools

We used using Docker Hub to automate image updates whenever new code is posted to **master** or **dev.**
Docker Hub’s automatic building feature was officially paywalled very late into this project’s development, somehow we have suffered no repercussions and are still able to build our images.
If Docker Hub’s automatic building had not been available for this project we would have built our images via Git Action.

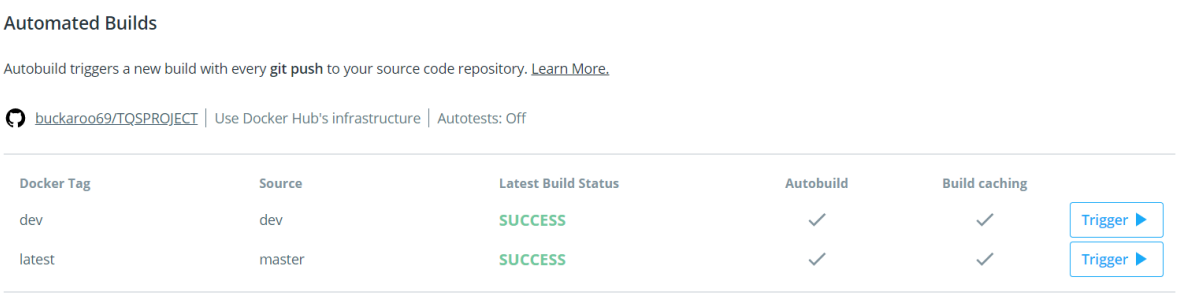


Fig 3: Docker Hub Automated Builds

All resources are packaged inside the containers, meaning nothing else is required to continuously integrate new code.

BUILD LOGS DOCKERFILE README

```
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ clientApp ---
[
INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to /workdir/server/target/classes
[INFO]
[INFO] --- maven-resources-plugin:3.2.0:testResources (default-testResources) @ clientApp ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Using 'UTF-8' encoding to copy filtered properties files.
[INFO] skip non existing resourceDirectory /workdir/server/src/test/resources
[INFO]
[INFO] ---
maven-compiler-plugin:3.8.1:testCompile (default-testCompile) @ clientApp ---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-surefire-plugin:2.22.2:test (default-test) @ clientApp ---
[INFO] No tests to run.
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.5:report (report) @ clientApp ---
[INFO] Skipping JaCoCo execution due to missing execution data file.
[INFO]
[INFO] --- maven-jar-plugin:3.2.0:jar (default-jar) @ clientApp ---
[INFO] Building jar: /workdir/server/target/clientApp-0.0.1-SNAPSHOT.jar
```

BUILD LOGS DOCKERFILE README

```
FROM maven:3.6.3-jdk-11 AS builder
WORKDIR /workdir/server
COPY clientApp/pom.xml /workdir/server/pom.xml
RUN mvn dependency:go-offline

COPY clientApp/src /workdir/server/src
RUN mvn install
RUN mkdir  -p target/dependency
WORKDIR /workdir/server/target/dependency
RUN jar -xf ../*.jar

FROM openjdk:11-jre-slim

EXPOSE 8000
VOLUME /tmp
ARG DEPENDENCY=/workdir/server/target/dependency
COPY --from=builder ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY --from=builder ${DEPENDENCY}/META-INF /app/META-INF
COPY --from=builder ${DEPENDENCY}/BOOT-INF/classes /app
ENTRYPOINT ["java","-cp","app:app/lib/*","marchingfood.tqs.ua.MainAppApplication"]
```

Fig 4,5: Docker Hub Logs/ Online Dockerfile visualization

Every 1500 seconds a Watchtower instance running on the virtual machine we were provided updates running containers of our components to the newest version, meaning that our solution is continuously deployed.

```
version: "3.8"
services:
  backend:
    #build: .
    image: dioben/tqs-client-server:dev
    ports:
      - 8000:8000
    depends_on:
      db:
        condition: service_healthy
  db:
    image: postgres:13
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U root -h localhost -d postgres -p 5433"]
      interval: 5s
      timeout: 5s
      retries: 5
    ports:
      - 5433:5433
    command: -p 5433
    environment:
      - POSTGRES_USER=root
      - POSTGRES_DB=postgres
      - POSTGRES_PASSWORD=postgres
  backendmain:
    #build: .
    image: dioben/tqs-provider-server:dev
    ports:
      - 8080:8080
    depends_on:
      db_main:
        condition: service_healthy

  db_main:
    image: postgres:13
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U root -h localhost -d postgres"]
      interval: 5s
      timeout: 5s
      retries: 5
    ports:
      - 5432:5432
    environment:
      - POSTGRES_USER=root
      - POSTGRES_DB=postgres
      - POSTGRES_PASSWORD=postgres
  watchtower:
    image: containrrr/watchtower
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    command: --interval 3000
```

Fig 6: Continuous deployment strategy via Watchtower

3.3 Artifacts repository

Artifacts and resources were automatically packaged into images available at public docker repositories.
Logistics Component
Service Component

4 Software testing

4.1 Overall strategy for testing

In order to test code for both the frontend and backend of both components of the project we used a TDD approach, by initially sketching out what type of functionality is expected and from then devising tests. The tests made thereafter were unit tests and integration tests.
While some set up in terms of programming was done beforehand (mostly frontend prototypes and the setup of each component's database) most feature development was made after tests for said feature had been implemented. After a MVC prototype of each project was seen as 'complete' additional functional tests were thought and done for each component.
Furthermore in order to make these tests we utilized the JUnit 5 test framework alongside Mockito in order to mock various components during integration tests, MockMVC to mock API requests and Selenium for functional tests with the Frontend of both components.

4.2 Functional testing/acceptance

For writing Functional Tests, after most of the frontend had been established, we utilized Selenium library, taking into account what elements should be included in each page. Each test was done following the perspective of a potential user going through a use story. While it should be attested this is not utilizing the TDD approach, we opted to do it instead due to the way Selenium works, as the

frontend of the website might require quick change and that would lead to increasing redoing of Selenium tests to accommodate to new developments.
All tests were written using ChromeDriver, furthermore steps were taken to include a headless Chrome Driver running on our pipeline.

4.3 Unit tests

Most of our Unit tests revolve testing the functionality of our API from outside perspective by testing our Controller classes, in this, we mock our service classes and make sure inputs are being provided correctly, that the return value is valid, and in case of error that we return an HTTP status error code.
Besides those, we also have unit tests for our service classes in which we tested the interactions between these and the Controllers and Repositories classes, making sure that methods are called correctly and are returning the correct response for every situation.
Some unit tests for Repositories were also considered, but as there was no need for any complex queries we thought of these as redundant.