TQS: Quality Assurance manual

Diogo Silva [89348], Pedro Oliveira [89156], Pedro Escaleira [88821], Rafael Simões [88984]

v2020-05-143

Project management **1**
Team and roles 1
Agile backlog management and work assignment 2
Code quality management **5**
Guidelines for contributors (coding style) 5
Code quality metrics 6
Continuous delivery pipeline (CI/CD) **8**
Development workflow 8
CI pipeline and tools 10
CD pipeline and tools 11
Pipeline Monitoring 12
Software testing **12**
Overall strategy for testing 12
Functional testing/acceptance 13
Unit Tests 13
System and Integration Testing 14
Performance Testing 15
Production Analysis 15

1 Project management

1.1 Team and roles

The team is composed of highly capable developers, both in terms of Backend Development, with shared experience in all technologies utilized, as well as in the usage of the JavaScript frameworks for the Frontend development. We're all used to the Git Workflow from previous projects, i.e. working in separate branches for each feature that we were assigned and creating merge requests before pushing code to the main branch.

As requested by our professor, we organized the team members in 4 roles. In the following points, we can consult the division made:

- **Team leader**

    - We decided by absolute majority that **Diogo Silva** would be the perfect one for this position. His work is to decide which tasks to do next, by whom, and in what time. He also makes meetings with all the project members to discuss new work approaches and the efforts required in the following weekly sprint. Therefore, he also acts as the project's Scrum Master.
- **Product owner**
    - Our product owner is **Pedro Oliveira**: he will be the one to represent our clients' wants and needs; he is responsible for knowing what requirements are most important, and which should be done first; he will also be the one to present ideas and further features that should be implemented in the project.
- **DevOps master**
    - Because of prior experience, we decided that **Pedro Escaleira** would be the best person to fit this role. As such, he was responsible for the set-up of the repository, it«s **CI** and **CD pipelines**, as well as deploying all of our service's modules in the server.
- **QA Engineer**
    - Finally,, we attributed the role of QA engineer to **Rafael Simões**. He'll be the one in charge to check if the source code is according to the standards of our code style, and check if the tests the rest of the team implemented for their respective feature are thorough enough, warning us, as the developers, if we missed a use case that must be tested.
- **Developer**
    - As requested by the subject professor requested, we are all project developers. In this role, we created two subroles:
        - **Frontend developer:** Diogo Silva and Rafael Simões, which are responsible for creating all the web and mobile applications.
        - **Backend developer:** Pedro Escaleira and Pedro Oliveira, responsible to create the REST API which is supporting all the end user applications.

Our team's channel for communication was with the Slack app, having introduced several channels for different conversation topics, such as a deploy channel with a Bot connected to GitHub to tell us about the deploy stages and an Issue channel with a Bot connected to the Github Issues so that we are notified the moment an issue was assigned to us.

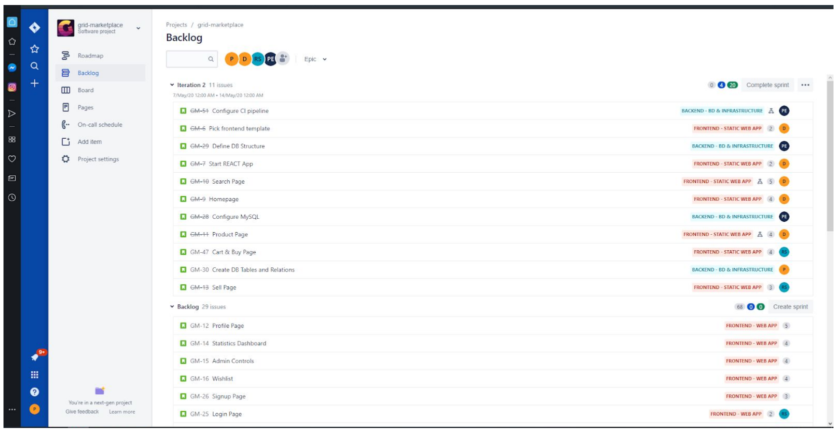
1.2 Agile backlog management and work assignment

Our development focused around a mix between the **Scrum** and **Kanban** strategies: where we prepare a backlog of tasks that have to be done for the project, and divide them into **To Do, In Progress,** and **Done**.

We have also adopted the use of Sprints: we start a **meeting** by each saying what has been done in the past Sprint, and move on to explain what needs to be done in the following one.

Each Sprint, in our strategy, will last one week, so that we all remain updated with the progress of the project, be it the frontend or the backend, while also retaining a fairly large window to work on the project and the assigned tasks.

In order to keep track of all this, we have been using **Jira**. Our team leader created tasks in the project's backlog, and assigned each one to a developer. He also divided all those tasks into **Epics**, representative of the project's milestones, wrapping several **subtasks**.. After being assigned to a task, the person is in charge of creating child issues, since the main task may prove to be too vague or too all-encompassing, not quite representing the work that needs to be done in a simple task.



Projects / grid-marketplace

Backlog

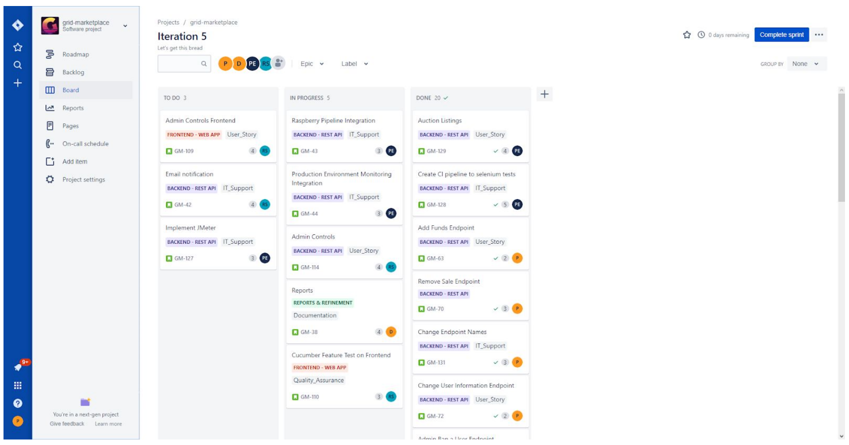
Iteration 2 11 issues 0 4 20
7/May/20 12:00 AM • 14/May/20 12:00 AM

~~GM-51~~	Configure CI pipeline	BACKEND - BD & INFRASTRUCTURE		PE
~~GM-6~~	Pick frontend template	FRONTEND - STATIC WEB APP	2	D
~~GM-29~~	Define DB Structure	BACKEND - BD & INFRASTRUCTURE		PE
~~GM-7~~	Start REACT App	FRONTEND - STATIC WEB APP	2	D
~~GM-10~~	Search Page	FRONTEND - STATIC WEB APP	5	D
~~GM-9~~	Homepage	FRONTEND - STATIC WEB APP	4	D
~~GM-28~~	Configure MySQL	BACKEND - BD & INFRASTRUCTURE		PE
~~GM-11~~	Product Page	FRONTEND - STATIC WEB APP	4	D
GM-47	Cart & Buy Page	FRONTEND - STATIC WEB APP	4	RS
GM-30	Create DB Tables and Relations	BACKEND - BD & INFRASTRUCTURE		P
~~GM-13~~	Sell Page	FRONTEND - STATIC WEB APP	3	RS

Backlog 29 issues 68 0 0

GM-12	Profile Page	FRONTEND - WEB APP	5	
GM-14	Statistics Dashboard	FRONTEND - WEB APP	4	
GM-15	Admin Controls	FRONTEND - WEB APP	4	
GM-16	Wishlist	FRONTEND - WEB APP	4	
GM-26	Signup Page	FRONTEND - WEB APP	3	
GM-25	Login Page	FRONTEND - WEB APP	2	RS

Img 1.1 - An image representative of our Jira Backlog



Projects / grid-marketplace

Iteration 5

Let's get this bread

0 days remaining

TO DO 3

- Admin Controls Frontend — FRONTEND - WEB APP, User_Story — GM-109 — 4 — RS
- Email notification — BACKEND - REST API, IT_Support — GM-42 — 4 — RS
- Implement JMeter — BACKEND - REST API, IT_Support — GM-127 — 3 — PE

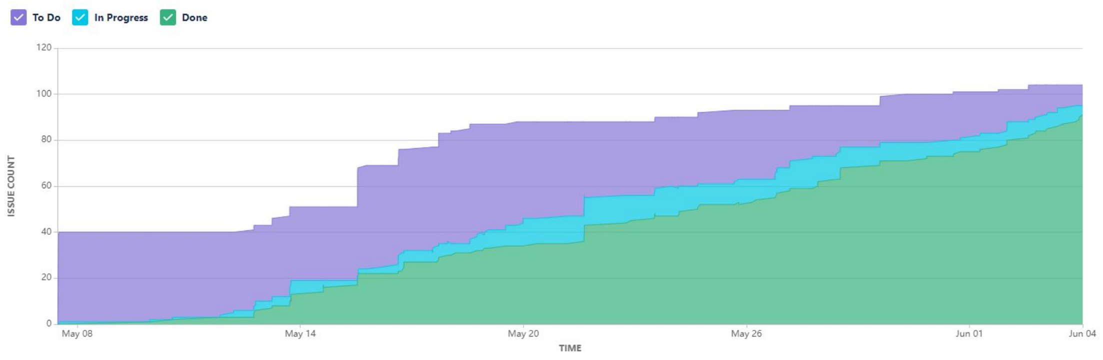
IN PROGRESS 5

- Raspberry Pipeline Integration — BACKEND - REST API, IT_Support — GM-43 — 3 — PE
- Production Environment Monitoring Integration — BACKEND - REST API, IT_Support — GM-44 — 3 — PE
- Admin Controls — BACKEND - REST API, User_Story — GM-114 — 4 — RS
- Reports — REPORTS & REFINEMENT, Documentation — GM-38 — 4 — D
- Cucumber Feature Test on Frontend — FRONTEND - WEB APP, Quality_Assurance — GM-110 — 3 — RS

DONE 20 ✓

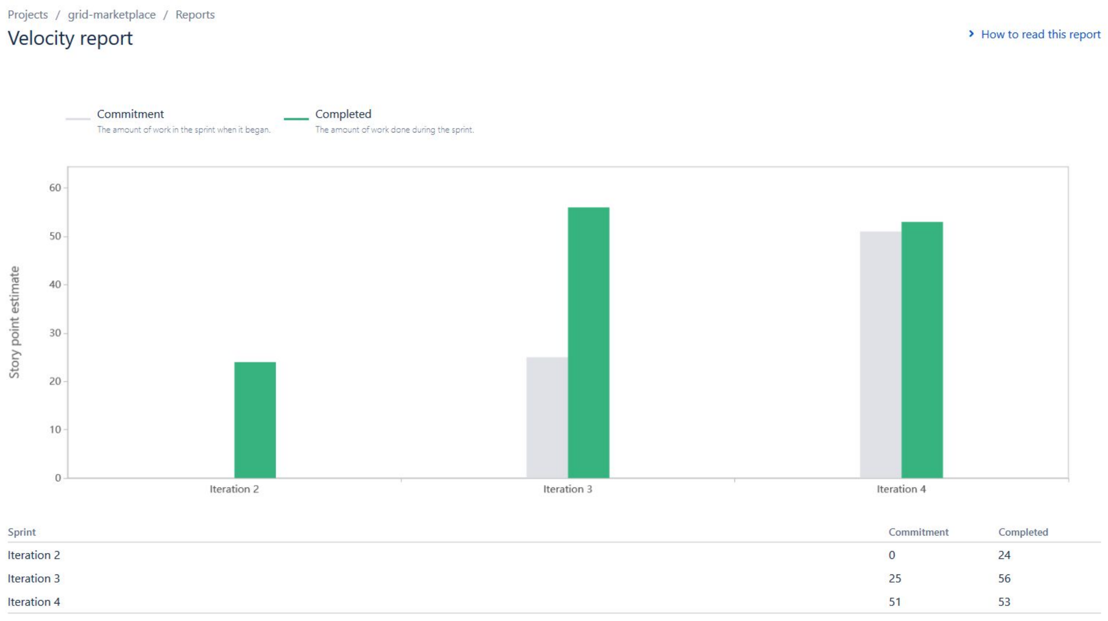
- Auction Listings — BACKEND - REST API, User_Story — GM-129 — ✓ 4 — PE
- Create CI pipeline to selenium tests — BACKEND - REST API, IT_Support — GM-128 — ✓ 5 — PE
- Add Funds Endpoint — BACKEND - REST API, User_Story — GM-63 — ✓ 2 — P
- Remove Sale Endpoint — BACKEND - REST API — GM-70 — ✓ 3 — P
- Change Endpoint Names — BACKEND - REST API, IT_Support — GM-131 — ✓ 3 — P
- Change User Information Endpoint — BACKEND - REST API, User_Story — GM-72 — ✓ 2 — P
- Admin Ban a User Endpoint

Img 1.2 - An image representative of one of our Sprints defined in Jira



Img 1.3 - A cumulative graph representative of our backlog's progress the day before delivery

We also associated each task with a **score** detailing its estimated difficulty. This score helps us balance the workload between Sprints and between developers, since it would not be particularly fair if one person had all the most important and most challenging tasks, while others had simpler ones. We also added a **label** detailing if the task is referring to a Use Case, meaning it's part of a feature the client will interact with, or general IT support, implying the task is a more technical one, it may be related to database maintenance, or CI/CD pipelines, but it's not directly related to our end-user.


Projects / grid-marketplace / Reports
Velocity report
› How to read this report
Commitment
The amount of work in the sprint when it began.
Completed
The amount of work done during the sprint.
Story point estimate
60
50
40
30
20
10
0
Iteration 2
Iteration 3
Iteration 4

Sprint	Commitment	Completed
Iteration 2	0	24
Iteration 3	25	56
Iteration 4	51	53

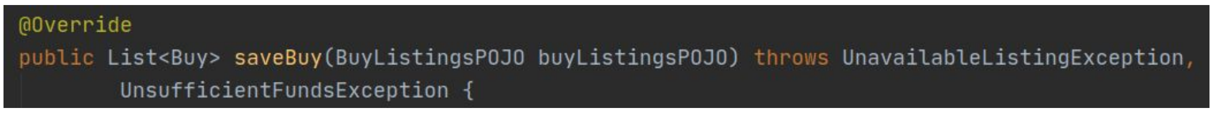
Img 1.4 - A graph showcasing the speed of each of the sprints. It should be noted that we only started assigning points to each task at the start of the second sprint, due to the teacher's suggestion. It should also be noted that on Iteration 3 we managed to complete all of our assigned tasks and even go beyond. Iteration 4 shows the end of the project, the reason the two bars aren't leveled is due this screenshot having been taken the day before delivery, with some work still needed to be done

2 Code quality management

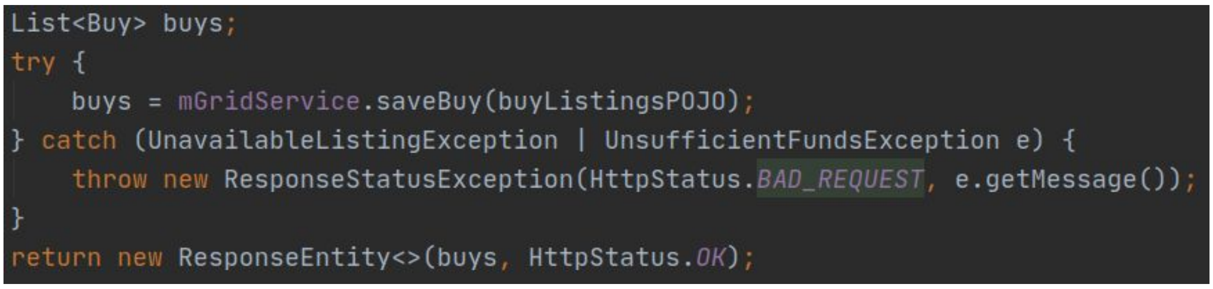
2.1 Guidelines for contributors (coding style)

As our main programming language for the backend was Java-based, we used the more famous Google code, following the guide under the AOSP Java Code Style.

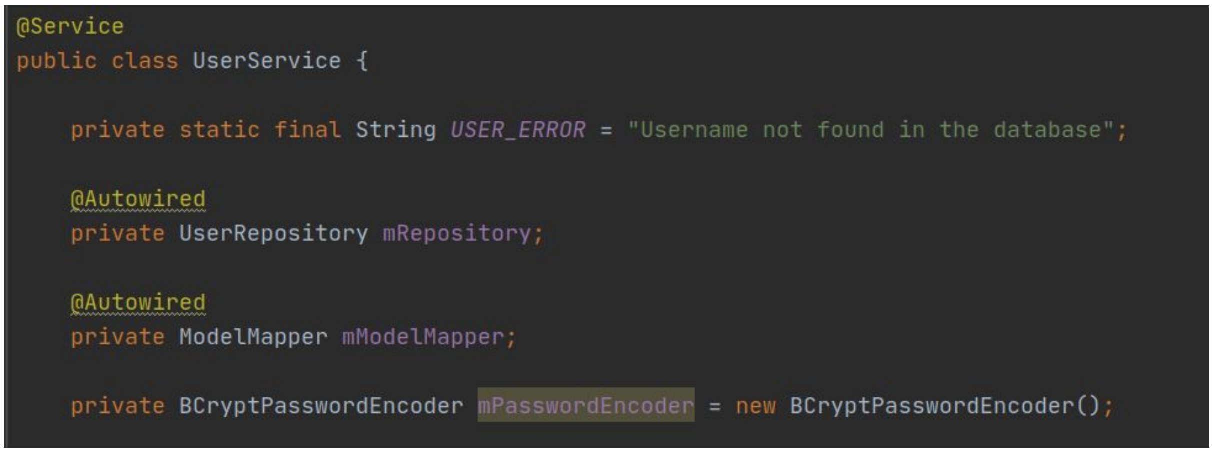
You should already expect some of these rulings from other languages, such as properly treating Exceptions (catching specific exceptions and properly treating them, by returning an error message or throwing another exception), properly naming Test functions (by making the conditions and result explicit), avoiding generic imports and making sure that lines remain under 100 characters.



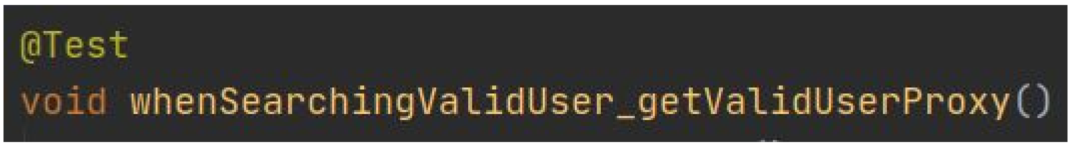
```java
@Override
public List<Buy> saveBuy(BuyListingsPOJO buyListingsPOJO) throws UnavailableListingException,
        UnsufficientFundsException {
```



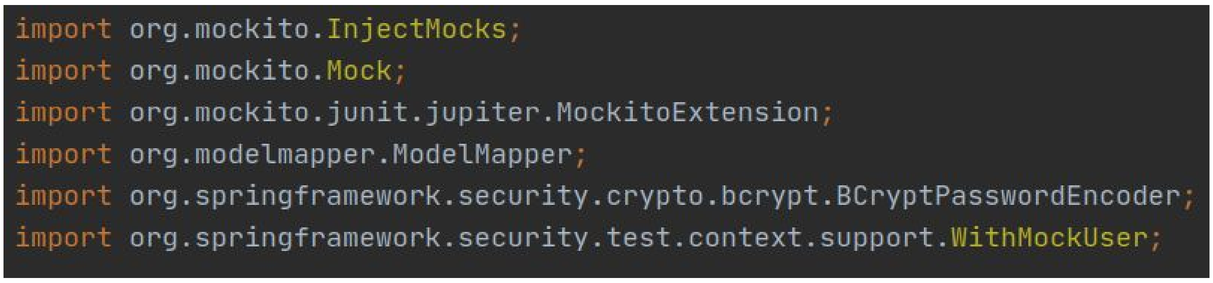
```java
List<Buy> buys;
try {
    buys = mGridService.saveBuy(buyListingsPOJO);
} catch (UnavailableListingException | UnsufficientFundsException e) {
    throw new ResponseStatusException(HttpStatus.BAD_REQUEST, e.getMessage());
}
return new ResponseEntity<>(buys, HttpStatus.OK);
```



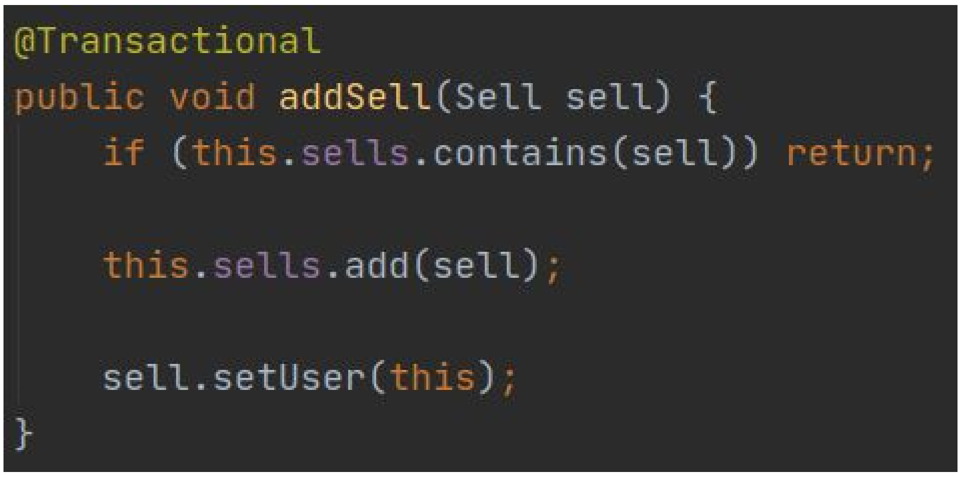
```java
@Service
public class UserService {

    private static final String USER_ERROR = "Username not found in the database";

    @Autowired
    private UserRepository mRepository;

    @Autowired
    private ModelMapper mModelMapper;

    private BCryptPasswordEncoder mPasswordEncoder = new BCryptPasswordEncoder();
```



```java
@Test
void whenSearchingValidUser_getValidUserProxy()
```



```java
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import org.modelmapper.ModelMapper;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.test.context.support.WithMockUser;
```



```java
@Transactional
public void addSell(Sell sell) {
    if (this.sells.contains(sell)) return;

    this.sells.add(sell);

    sell.setUser(this);
}
```

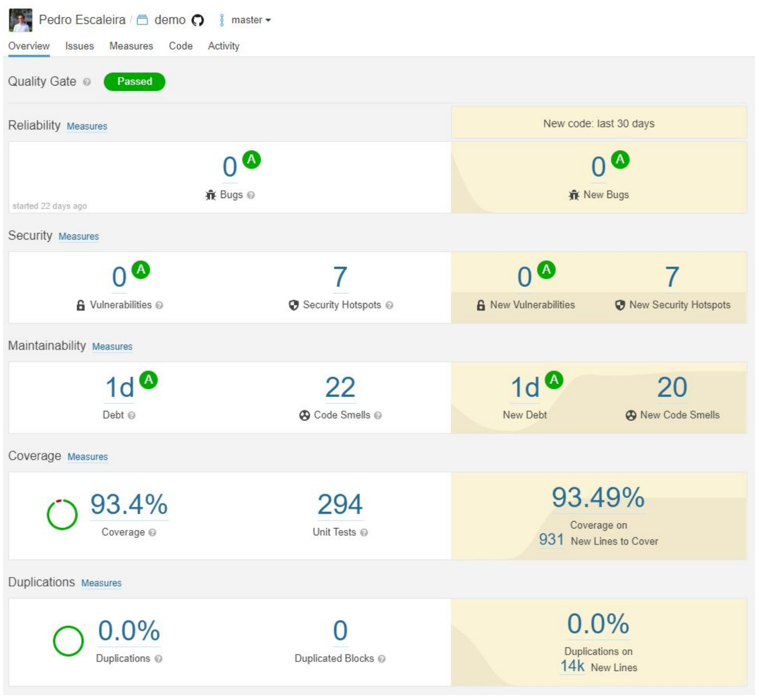
Img 2.1 - Some examples of our code style

Others that are more specific to the project include the naming conventions we adopted for our variables: any non-public, non-static field in a class must start with *m*, any static field must begin with an *s,* and any constant must be written fully capitalized and with underscores.

Furthermore, **if** blocks with one instruction only must be written either inline with the condition, or with brackets. A developer can’t have the block and the condition in different lines without brackets surrounding the code.

2.2 Code quality metrics

Following efforts made in other prior projects, in which we used **SonarCloud** or **SonarQube**, we decided to use the first one for this project. In the image below, we can behold the dashboard of it for our project on the early stages, where we can consult the number of **bugs**, **code smells**, **security issues** and others:



Pedro Escaleira / demo | master

Overview | Issues | Measures | Code | Activity

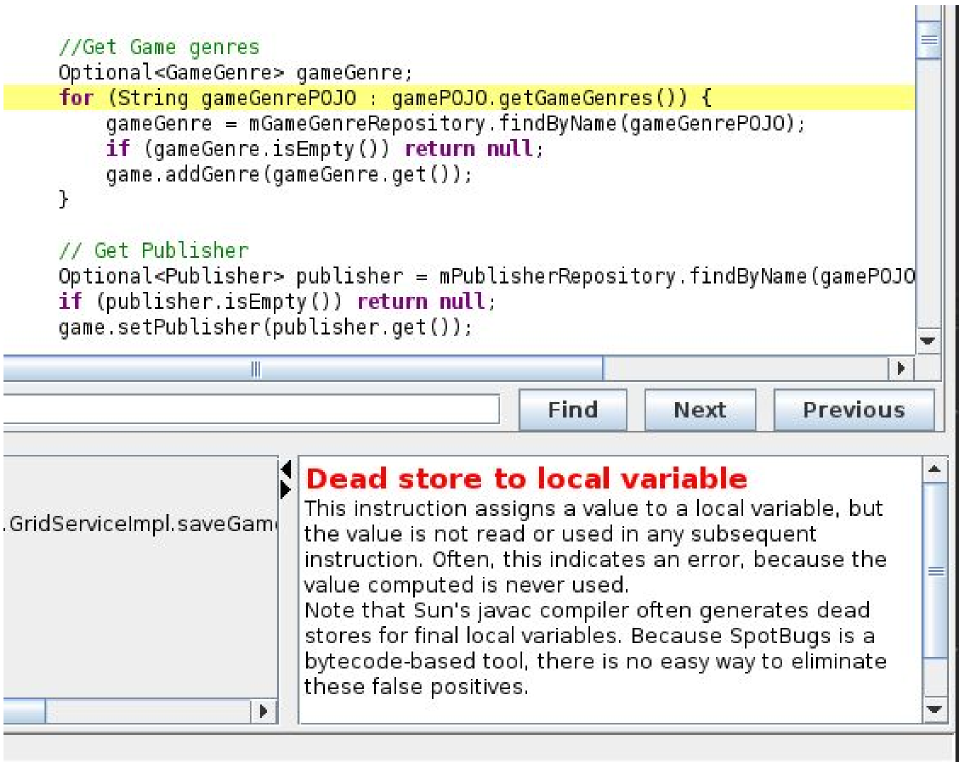
Quality Gate: Passed

		New code: last 30 days	
Reliability			
0 (A) Bugs		0 (A) New Bugs	
started 22 days ago			
Security			
0 (A) Vulnerabilities	7 Security Hotspots	0 (A) New Vulnerabilities	7 New Security Hotspots
Maintainability			
1d (A) Debt	22 Code Smells	1d (A) New Debt	20 New Code Smells
Coverage			
93.4% Coverage	294 Unit Tests	93.49% Coverage on 931 New Lines to Cover	
Duplications			
0.0% Duplications	0 Duplicated Blocks	0.0% Duplications on 14k New Lines	

Img 2.2 - Statistics that came from SonarQube

This tells us that our source code currently has near 300 unit tests, covering a total of 93.5% of the Backend. It’s important to note, the pipeline isn’t accounting for the frontend code or the functional tests that have been implemented.

We also have a pipeline in our repository that automatically searches for possible bugs in each code push, using the **Spotbugs** tool. This would tell us if our code is completely bug-free, that may lead to a future error that we were not expecting or accounting for in our tests.



```java
//Get Game genres
Optional<GameGenre> gameGenre;
for (String gameGenrePOJO : gamePOJO.getGameGenres()) {
    gameGenre = mGameGenreRepository.findByName(gameGenrePOJO);
    if (gameGenre.isEmpty()) return null;
    game.addGenre(gameGenre.get());
}

// Get Publisher
Optional<Publisher> publisher = mPublisherRepository.findByName(gamePOJO
if (publisher.isEmpty()) return null;
game.setPublisher(publisher.get());
```

Dead store to local variable

This instruction assigns a value to a local variable, but the value is not read or used in any subsequent instruction. Often, this indicates an error, because the value computed is never used.
Note that Sun's javac compiler often generates dead stores for final local variables. Because SpotBugs is a bytecode-based tool, there is no easy way to eliminate these false positives.

Img 2.3 - Bug found when running the pipeline

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

In this project, we decided to use the **GitHub workflow**, as all of the members are used to working with it from other projects. In summary, our process is as specified on the page GitHub Guidelines . When each of us wants to work on a new feature, these steps are followed:

1. Create a new branch from **master**
    a. For features the branch should follow the nomenclature: **feature/<feature_name>**
    b. For hotfixes and bug corrections the branch should have the naming template: **hotfix/<feature_name>**
    c. Note that there may be variations of these templates such as **feature/frontend/<frontend_feature>** for all frontend features, and backend features which may be **feature/api/<backend_feature>**.
    d. When we wanted to make a new deploy, the nomenclature of the branch was: **deploy/<branch_name>**
2. Work on that branch

a. We start working on the features, introducing new functions, or correcting past bugs; occasionally, we may also have to update our branches with the code in the main branch.

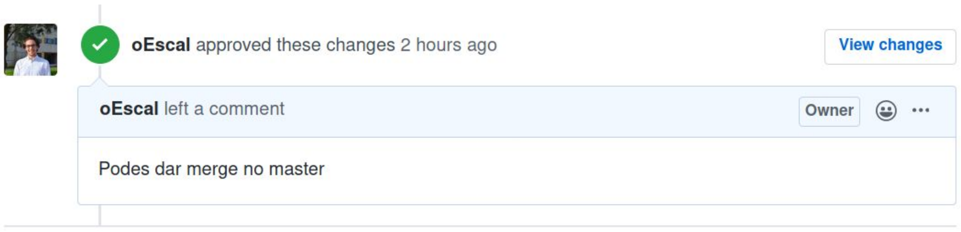
b. A feature is considered done not just if the code for the feature is finished, but also if it has the tests for what might go wrong or right when executing the feature.

3. Pull request and Code review

a. When we finish the given work on a branch, we finally request a review from one of our colleagues, with the purpose to have a second person to verify if the new code is well written and in accordance to the code style, doesn't have any bugs, has the proper tests for the feature, and it has passed all of them.

b. The strategy we use to select the person who will review our code is to pick the person working on the corresponding frontend requests or the person that's most connected to the feature.

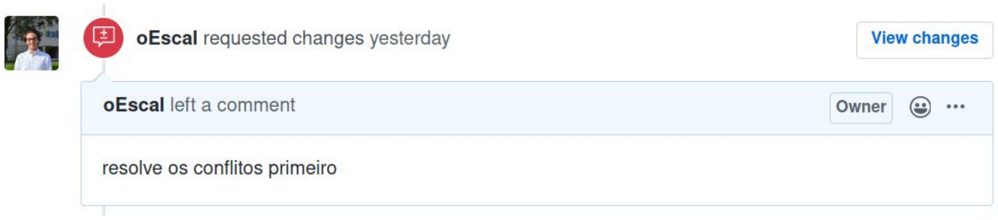
c. Lastly, if the reviewer thinks that the code can be merged to the main branch, he may **approve**:


oEscal approved these changes 2 hours ago

oEscal left a comment

Podes dar merge no master

On the other hand, if the reviewer thinks the code shouldn't be merged to the main branch, for some possible problem, e.g. the code has conflicts with the master branch, he requests the main author to fix the problems found.


oEscal requested changes yesterday

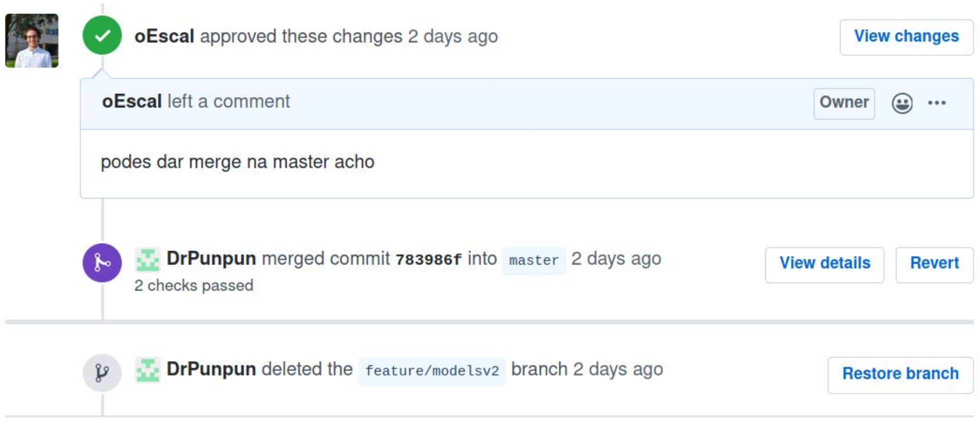
oEscal left a comment

resolve os conflitos primeiro

When this happens, the author fixes the code as requested, after talking with the reviewer for clarification, and, once again, the author asks a new review for the new changes:

4. Merge the branch code to master

    a. When at last the author of the branch gets an approval on its pull request from every reviewer, he will merge the new code to the master's branch, deleting the pull request branch in the process. This is done to avoid clutter and keep our git clean.



oEscal approved these changes 2 days ago

oEscal left a comment

podes dar merge na master acho

DrPunpun merged commit **783986f** into `master` 2 days ago
2 checks passed

DrPunpun deleted the `feature/modelsv2` branch 2 days ago

In terms of our team's "definition of done" it basically boiled down to, when a person is done with both implementing and properly testing a feature, creating the aforementioned pull request. This request would then be reviewed by the appropriate members, who could request changes, if anything wasn't up to standards or a bug was found, or accept the branch if they thought everything was ok. That feature would only be called as "done" after every single one of the reviewers accepted it. With this we assured that only fully done features would be merged over to our master branch, assuring that it contained the most stable and complete code possible. This definition also allowed us to have a concrete set of subtasks that would have to be ticked before a pull request could be opened giving us a method to better organize both ourselves and our workflow.

3.2 CI pipeline and tools

As we used **GitHub** to host our repository's code, we decided to use the new tool for Continuous Integration: **GitHub Actions**. Although we could have used other, more powerful, tools for this task, e.g. **Jenkins**, this is a relatively small project, and we just needed to have tests and static code verification, so we decided it would be better to stick to a simpler, but powerful tool.

In our project, we created 4 CI pipelines:

- **SonarCloud Workflow:** used to send new code to the **SonarCloud** project associated with this repository.
- **SpotBugs Workflow:** as the name implies, this workflow was used to verify possible bugs that our new code could have. Obviously, this task is already done by

SonarCloud, but with this tool, we got right away the possible bugs existence on the **GitHub** page, without the need to check the **SonarCloud Dashboard.**

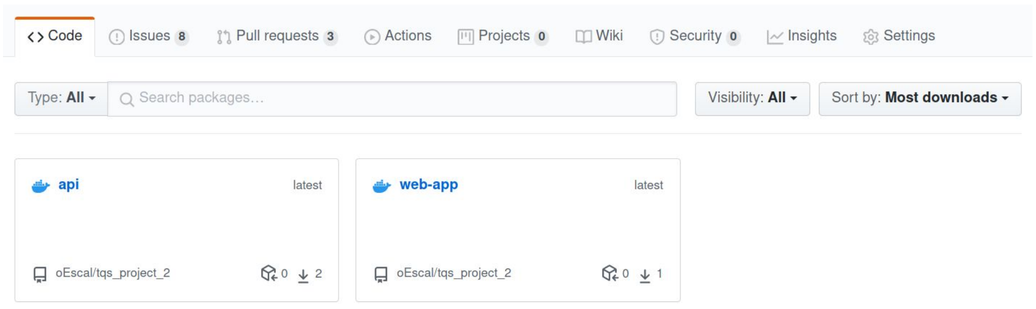
- **Tests Backend Workflow:** created with the purpose to run all tests we created during the project's elaboration. This way, we had the confirmation that new code doesn't break what's already been done.
- **Tests frontend Workflow:** created to test the features to the frontend as this develops, the same way we do to backend. This pipeline runs a development react server and maven selenium tests associated with the web application.
- **Tests mobile app Workflow:** this simple workflow was used to run the **npm tests** for the mobile application.

3.3 CD pipeline and tools

As for the CI pipeline, for the continuous delivery and continuous deployment we also used **GitHub Actions** to create the web application and rest api **docker images**.

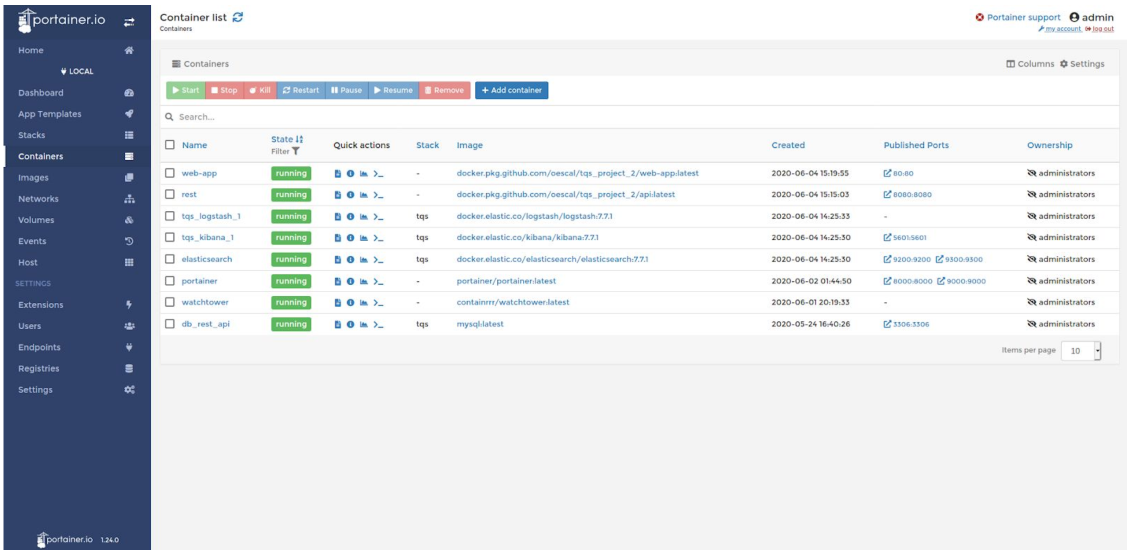
The pipeline created for this tasks was:

- **Deliver Workflow:** triggered when we made a **pull request** with the tag **deploy**. When a **new push** to a pull request with these properties was made, this pipeline created two **docker images**: **api** and **web-app**. This packages then where placed on the **GitHub Packages** repository page:



Then, on the server side, we had a service deployed on **docker** called **watchtower**, which is responsible for, every 5 minutes, to verify if there are new **docker images** on the repository and, in that case, to remove the old containers and build new ones.

It is noteworthy that we used **Portainer** to manage more easily the deployed containers:



Name	State Filter	Quick actions	Stack	Image	Created	Published Ports	Ownership
web-app	running		-	docker.pkg.github.com/oescal/tqs_project_2/web-app:latest	2020-06-04 15:19:55	80:80	administrators
rest	running		-	docker.pkg.github.com/oescal/tqs_project_2/api:latest	2020-06-04 15:15:03	8080:8080	administrators
tqs_logstash_1	running		tqs	docker.elastic.co/logstash/logstash:7.7.1	2020-06-04 14:25:33	-	administrators
tqs_kibana_1	running		tqs	docker.elastic.co/kibana/kibana:7.7.1	2020-06-04 14:25:30	5601:5601	administrators
elasticsearch	running		tqs	docker.elastic.co/elasticsearch/elasticsearch:7.7.1	2020-06-04 14:25:30	9200:9200 9300:9300	administrators
portainer	running		-	portainer/portainer:latest	2020-06-02 01:44:50	8000:8000 9000:9000	administrators
watchtower	running		-	containrrr/watchtower:latest	2020-06-01 20:19:33	-	administrators
db_rest_api	running		tqs	mysql:latest	2020-05-24 16:40:26	3306:3306	administrators

3.4 Pipeline Monitoring

Besides the actual pipeline we also implemented a pipeline monitoring tool. We coded a program that tracks the stage of the CI/CD pipeline running on our master branch utilizing **Python3** and the **Pygame** library. We then left this program running on a **Raspberry 3** computer continuously, so that when a new commit pushed to our main branch the graphical display showcases informations such as who made the push, the merge name, aswell as the state of all pipeline stages.

4 Software testing

4.1 Overall strategy for testing

When developing the source code for the Backend, we adopted a TDD approach, meaning we would talk, in group, about what the function should do, what exceptions it should raise and on what occasions, and what the return and final result should be.

After this, we would make the tests in accordance with the requirements established, unit tests and Integration tests, before actually coding the feature in the backend. Only after these tests are programmed, would we start the backend development, and implement the feature.

This was something we were already familiar with, as we had done the same type of approach in our last project. Of course, the scale was much smaller and had less features, since it didn't require authentication or database interactions, but we still had some experience in using this strategy.

As for our strategy for the frontend testing, due to the limitations of the technologies, we wrote the tests after the development of the frontend. However, it must be noted that we also worked with BDD in some of the features in the frontend, to experiment with this strategy as well.

The tests were developed using **JUnit 5** as the base framework for out testing; **Mockito** to simulate the interactions between classes that are dependent on one another; **SpringBoot MockMVC** to simulate the user and his interactions with the REST API, for the integration testing; **Selenium** to test the interaction with the Web app interface, **Cucumber** for the some feature tests (and to experiment with BDD) and **JEST** and **Enzyme** to test the mobile app's code.

4.2 Functional testing/acceptance

When it came to testing our service from a functional standpoint we opted to utilize **Selenium**. Each time a new page was added to the web app a corresponding test would be written for it, with the feature/page only being considered done after these tests passed. It should be noted that this is in fact not a TDD approach. We opted to write functional tests a posteriori due to the way Selenium lends itself to utilization, being way more intuitive to utilize after the pages are done, as well as due to the fact that due to the low time we had to complete the project we couldn't produce frontend mocks for every page, hence the design was constantly evolving which would make writing the functional tests first complicated.

Before writing the tests themselves, we started off by creating the **WebAppPageObject.java** class. This class follows the PageObject Pattern, encompassing all support methods for driver control and page interaction. With this we were able to produce readable and clean tests, alongside being able to reutilize a lot of interaction code easing the production of tests. We included drivers for Firefox and Chrome due to our team members preferring either or browser, but these were posteriorly commented out in lue of a Headless Chrome Driver that allowed these tests to run on our pipeline.

We also wanted to try out using a **BDD** methodology in this project, mostly for pedagogical intents, and as such we decided to implement a couple of features following this practice. For this we combined **Cucumber** with **Selenium** in order to produce feature tests that were actually written before the actual page was produced.

Lastly, it should be stated that at first we wanted to create tests for our mobile app using **Appium**, which functions similarly to Selenium, hence allowing us to produce functional tests also. However, the setup process was rather infernal, and especially when considering the time we had and online suggestions, we opted to not include these tests (swapping them for Unit tests as we'll explain in the next section).

4.3 Unit Tests

Our Unit tests mainly revolve around the three different components in our system: our Repository classes, our Service classes and our Controller Classes.

The unit tests revolving around the repository tests were to make sure that everything was being saved properly in the database in-memory, and that the more complex queries were working as we expected them to.

The Service unit tests are around the business logic between the interaction of the Controller classes and the Repositories, receiving the expected objects from the controllers, and using mocks around the repository. This would allow us to make sure that the appropriate methods are being called, and the different results from the repositories will result in the right response or throw the correct exception.

Finally, the Control unit testing is around making sure that the JSON objects are being mapped correctly when a valid object in the request, and if they're not valid, we are sending the correct Error HTTP status, along with a descriptive reason for the failure. In this unit tests, we used the MockMVC from the **Spring Boot** framework to simulate the request, and we mocked the behaviour from the Service.

On the frontend side we implemented Unit tests on the mobile app utilizing a mixture of **Jest** and **Enzyme**. With these tools we were able to mock the behaviour of our REST API and test the code we utilize to fetch information and state updates. Each screen on the mobile app has a corresponding test suite associated with it (sometimes even more than one) which tests all of their methods extensively, either methods like page changing or interactions with our REST such as searching or loading information. By proxy, we would also like to point out that, due to code used to interact with the API being mostly the same between the Web and Mobile app, by doing unit tests on the latter we are also inadvertently unit testing some of the web apps fetch methods.

Img 4.1 - Jest - A JS Unit Testing Tool

4.4 System and Integration Testing

API testing was similar to unit testing in the Controller. The tests themselves are practically the same, differing only in the setup that's connected to the test. Instead of simply using Mocks to simulate the entities, we are using the already implemented entities, and save perfectly valid objects as we would expect from a realistic scenario to set up the context for the return value of the request.

To apply these tests, we have use some of the Springboot annotations:

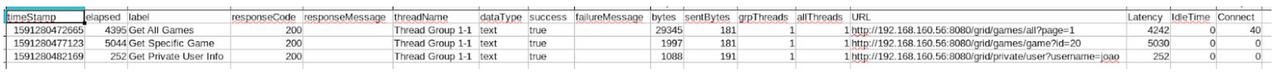
- @AutoConfigureMockMvc to configure the MockMvc and simulate the client requests to the API.
- @AutoConfigureTestDatabase to configure the in-memory H2 database, so that we don't have to use our actual database for the testing, as it would cause a lot of unnecessary errors.
- @SpringBootTest to start the web context as if it was a regular execution of the REST API
- @DirtiesContext to reset the databases, and not have errors regarding the dependencies between the foreign keys when clearing the databases.

4.5 Performance Testing

In order to test the performance of our REST API, we used the **JMeter** framework, and tested the most important features: check all the games, check a game in particular, and check my information as a user of the platform.

Endpoint	Latency	Total Elapsed Time
/grid/games/all?page=1	4242	4395
/grid/games/game?id=20	5030	5044
/grid/private/user?username=joao	252	252

As we can see, we have quite the large latency, which can be explained due to be using a VPN and some problems on the server side, but as we can see in the actual difference between requests, they do not take long to be processed internally, proving to be very good results for our developers.



timeStamp	elapsed	label	responseCode	responseMessage	threadName	dataType	success	failureMessage	bytes	sentBytes	grpThreads	allThreads	URL	Latency	IdleTime	Connect
1591280472665	4395	Get All Games	200		Thread Group 1-1	text	true		29345	181	1	1	http://192.168.160.56:8080/grid/games/all?page=1	4242	0	40
1591280477123	5044	Get Specific Game	200		Thread Group 1-1	text	true		1997	181	1	1	http://192.168.160.56:8080/grid/games/game?id=20	5030	0	0
1591280482169	252	Get Private User Info	200		Thread Group 1-1	text	true		1088	191	1	1	http://192.168.160.56:8080/grid/private/user?username=joao	252	0	0

Img 4.2 - Some of our JMeter Results

4.6 Production Analysis