TQS: Quality Assurance manual

Alexandre Lopes [88969], André Amarante [89198], João Nogueira [89262], Tiago Melo [89005]
v2020-06-04

1 Project management

1.1 Team and roles

In order to have a more organized team and maintain the focus of each member in their tasks, we decided to assign each one of us a specific role. This way, each team member can focus on their subset of tasks, which are related to his role, instead of getting through a frequent context switch, which is very inefficient. So, we assigned four different roles: team manager, product owner, DevOps master, and developer.

André was assigned to the team manager and, as so, he is responsible to manage the team workload, ensure the priorities are well defined and ensure their concretization in time, as well as promote good collaboration between team members in order to have the expected project outcomes.

Alexandre is the product owner, as he is who came up with the idea and concept for the application. As the product owner, Alexandre will be involved in accepting the solution increments and make clear to the group what are the expected product features and how they shall behave.

João and Tiago were both assigned to the DevOps master role as this one is more complex and critical to the whole functioning of the application. Tiago and João will be in charge of setting up the infrastructure and its configurations. This includes configuring the CI/CD pipeline, placing the git repository, preparing the deployment environment (containers, VMs, etc) and more.

Finally, all four members were also developers, contributing to each development task in order to meet the user stories and the project's requirements. Alexandre and André focused more on the frontend of the application (developing UI, assuring a good user experience, displaying the correct data to the user that came from the backend, etc), while João and Tiago contributed more to the backend (development of the API, database configuration, etc). Also, Alexandre and André were who developed the mobile app for the external client module, which makes use of the developed API.

1.2 Agile backlog management and work assignment

As the tasks required to accomplish the goals and requirements for this project are so many and can be diverse between team members, in order to keep track of what is done, what is being done and what needs to be addressed next, we decided to use Trello with Agile Tools. With that, we were able to raise new tasks, assign to each team member, and establish deadlines in some tasks, according to the iteration plan defined for the project. Besides the internal tasks, we also used Agile Tools add-on on Trello to help us to define, prioritize and estimate user stories based on the project schedule and on the effort needed to have the functionalities that meet each user story. In each iteration, we looked at the backlog, analyzed which ones were fully completed, and addressed the user stories with most priority. About user stories in particular, which were put in the backlog and in each iteration, we also outlined scenarios to define the steps that need to be met to accomplish that same story. Our Trello board can be seen here, but one has to be a member of it in order to see it.

Besides Trello, we also used Slack to make the team synchronized and aware of what is being done at each time, as well as schedule regular meetings in order to each team member present to the rest of the team what has been working on and, as a team, define new goals and which steps that should be taken next.

The project outcomes were also shared between team members through Google Drive (such as meeting outcomes, project reports, etc) and through the git repository.

2 Code quality management

2.1 Guidelines for contributors (coding style)

During the development of this project, which was developed in Java, we followed the main naming conventions of this language, such as class names with first letters in uppercase, public fields named with lowercase, constants with all caps, and underscores, etc. We also followed some generic programming standards and good practices, such as handling possible errors through catching exceptions and dealing with null values, providing useful error messages, avoiding redundant or very extensive code that can be inefficient, writing comments, adding TODOs to the code, etc. These practices are in line with the Java Coding Style Rules of the AOS project and the main language standards, and the team was guided and inspired by them and didn't define any major new internal guidelines.

Besides those practices, the team was also encouraged to adopt the same software design patterns as needed. The rationale of this approach is to have a more consistent and maintainable code that can be easily read, easily corrected, and also less error-prone.

2.2 Code quality metrics

In order to have a reliable measurement of the quality of the developed code, we took advantage of static code analysis tools, mainly SonarQube and JaCoCo plugin. These tools allowed us to see how robust our code was in terms of security, code smells/good practices, testing coverage and so on. To do that, we used the default SonarCloud quality gate, since we considered this sort of configuration consisted in a robust way to enforce quality code, encouraging us to refactor the code as needed. The defined conditions under which our code would pass during the analysis are:

- Code coverage on new code should be greater than 80%
- Duplicated lines should be less than 5%
- Maintainability rating should be A
- Reliability rating should be A
- Security rating should be A

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

The chosen workflow to be applied during the project's development was the GitHub Flow.
It focuses on feature-branching, pull-requests, code-review, and continuous deployment and merging. Essentially, a new branch is created for each new feature to be developed. These branches help maintain the master version stable and clean, allowing experimentation and providing modularity. Each branch will have a series of commits with concise, meaningful commit messages. This allows us to keep track of the changes being made and roll back to a previous point if necessary. After the branch has been worked on, a merge/pull request (GitLab vs GitHub terminology) is opened and the code is reviewed among peers. Here we can keep track of the current changes, commenting on the code, or asking for feedback. In this project the code review is done in pairs, meaning that each member of the backend and frontend teams reviews each other's code. For a merge request to be accepted the following criteria are evaluated:

- Coding standards
- Clear commit messages
- Code organization and repetition
- Unit tests and BDD (developing and committing those tests, before the actual implementation).
- SonarCloud issues.

All merge requests are also notified in the team's Slack Workspace, with the proper gitlab integration configurations. Once a merge request is accepted, we are positive the new code is tested and working, and it is then merged with the project's master branch.

Each feature-branch consists of processed user stories. It works as follows:

1. A user story is discussed with the Product Owner and within the development team.
2. The user story is created, stored in the project's backlog (Trello), and given a priority and points based on the difficulty of the given task.
3. The story is decomposed into features that are placed in the 'To Do' card and assigned to the developers.
4. Features are developed in their own branches, following the aforementioned GitHub Flow.
5. A merge request for the feature-branch results in code reviewing according to the above standards and requirements.

Once the corresponding feature-branch has been merged with the master version the user story is considered to be finished, given it successfully passed the previously referred peer-review, acceptance criteria, and unit testing.

3.2 CI/CD pipeline and tools

In order to promote and have easier code reviews, developers should stick to making short, yet frequent commits, rather than long sporadic ones. This eases the load on the code reviewer side and allows developers to ensure the codebase is kept up with the established standards. For the code to be accepted into the codebase, it needs to be tested which, ideally, is done automatically. One way to achieve this is by implementing a Continuous Integration pipeline, which asserts whether or not a given build passes or fails whenever a commit is pushed.

The tool used for the CI pipeline was the GitLab CI system. By providing Runners and containers where the code will run and be tested, we can configure a CI pipeline file to set up the stages and phases of our pipeline. In case one of them fails, the rest of the pipeline isn't executed and developers can check the logs and troubleshoot.

In an earlier stage, our pipeline consisted of three very basic stages, concerning only the maven project itself:

- **Build:** the code was compiled using $mvn clean compile
- **Test:** which maps to the maven goal of $mvn test
- **Deploy:** which consists of packaging the application

Currently, the CI pipeline has these four stages:

- **Connect:** An image of the chosen project's database, mysql, is initialized with the proper configuration settings, creating the database, and mysql-users. We can now execute the remaining steps while maintaining a database connection.
- **Build:** executes the maven goal $mvn clean compile, compiling all the project's source code.
- **Test:** which runs $mvn test, therefore running all our designed Unit and Integration tests.

- **Package:** running $mvn package and generating the project's .jar file.

SonarCloud integration is also configured in GitLab's CI file so that whenever the master branch is updated, or a merge request occurs, the new code is checked for issues.

Furthermore, any broken (failed) pipelines are notified on the team's Slack Workspace.

Regarding deployment, the application as a whole is permanently available on a Virtual Machine provided by the course instructor, accessible by connecting to the University's VPN, on this address.

In order to achieve continuous deployment, when new code is pushed to the master branch, the previously mentioned pipeline is executed on a local runner installed on the Virtual Machine, instead of using GitLab's shared runners. This pipeline includes a fifth stage, deploy, in order to build the application and make it available. The approach uses standard Docker containers, together with a docker-compose.yml file. Furthermore, the database is also initialized with some sample data after it boots up.

4 Software testing

4.1 Overall strategy for testing

For the tests, the general strategy used was Test Driven Development, with a general set of each type of test developed before the actual functionalities were put in place. This allows the team to first specify in test form how a functionality should behave and then implement it, knowing that it will be finished once the test passes. The team also opted to use Behavior Driven Development with Cucumber to validate the core user stories, which was another way to create higher-level functionality testing before the actual development. All four team members were involved in the continuous testing phase of the project.

4.2 Functional testing/acceptance

Functional tests were developed using the Selenium tool and the Page Object pattern. Some were written after the functionality itself was implemented, while the last iteration followed TDD. Associated with this testing module, BDD was also used so that each functionality had both a Selenium-based test and a Cucumber one. These last ones were based on features and scenarios previously written in the beginning stages of the project by all four members of the group.

The executed plan was to have the team focused on covering as much of the user interaction points as possible, focusing first on the main user interactions flows, followed after by secondary ones. All user functionalities ended up being tested.

This section of testing was also developed by two developers, using pair programming and having two perspectives to discuss and implement not only the tests but also the functionalities.

As a part of the policy for test writing, while the functionalities were shown and approved by the whole team in the weekly team meeting, functional tests had to be approved by half the members (the frontend side of the team) as they had full knowledge of the functionalities flows, actions and interactions.

4.3 Unit tests

Unit testing was developed using the JUnit framework which all team members were already familiar with. These tests were created from a developer perspective. Therefore, it was expected and achieved coverage of all developer-created methods leaving things like getters, setters, and constructors untested as they already obey a tested schema. Furthermore, since the part of our login and registration services make use of the Spring Security framework, said parts aren't tested either.

The unit tests developed for this project focused on the three main modules of the app: Controller, Repository, and Service. Since we had separate controllers, services, and repositories per entity, most of our unit tests were split across Items and Users, the two main entities of the platform.

For the Repository tests, we tested the methods that will be called using our developed API with dummy data and the @DataJpaTest annotation. Once again, we developed tests for the User and Item repositories.

As to the service layer, tests were developed mocking repository behavior, as well as other modules out of the scope of these tests, like password encryption classes. In this context, we aimed to develop tests that allowed us to understand whether the interaction between the service and database was occurring as expected, and if the business logic implementation had been properly achieved. Additionally, sometimes multiple services needed to interact in order to achieve certain operations. For such cases, we developed a UserAndItemService test, where we mock other dependencies and simulate and test the coupled behavior between the modules. This proved to be extremely useful without compromising the test structure. Furthermore, since this is still a same-module sort of test, we considered it to be Unitary.

The controller tests weren't developed in an extensive manner since most of the controller interaction was tested and validated in Integration Testing, as we will discuss shortly. Nevertheless, using mocks for the Repository and Service layers, we tested the User Controller using the MockMvc module.

4.4 System and integration testing

Integration tests were developed using Spring MockMvc and REST Client (TestRestTemplate). This sort of testing was mostly used to test the API since the requests tested most of the system by interacting with the Controller, Service and Persistence modules. For this part of QA, we tried to develop extensive tests and cover most of the scenarios and combinations with the parameters.

For our integration tests, we focused primarily, as mentioned above, on the API interaction, since this included every other module on which the system relies on. This way, we ensure everything is working as expected when tests pass. The strategy for the tests was to generate most of the combinations possible using the parameters exposed by each endpoint, in order to maximize the coverage on the business logic and have a better grasp of the results each endpoint is producing. Once again, these tests were achieved by loading the database with dummy data and performing operations on said data, in order to have a controlled testing environment and know what to expect from each operation.

4.5 Performance testing

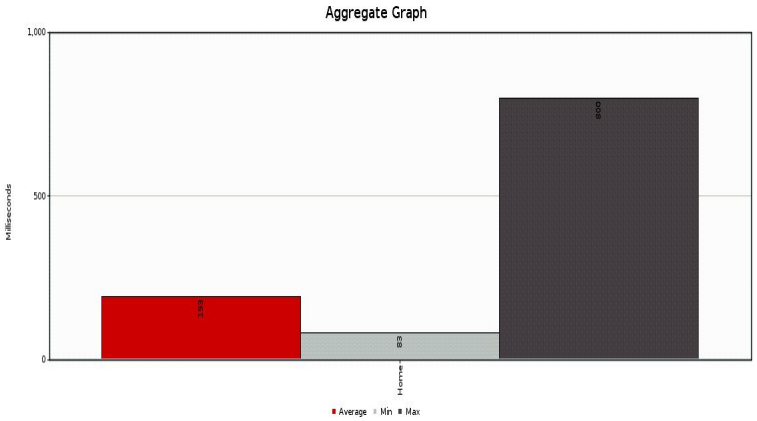
In order to test the performance of our platform under reasonable user loads, we made use of the JMeter tool and exported the response times for the core interactions with our app: accessing the Home page, fetching all items using the respective API endpoint and fetching only a category of items likewise. Furthermore, in order to have a better understanding of the results we obtained with this sort of testing, we performed similar tests on the major Portuguese sales platform OLX, since it is somewhat similar to ours, and used these results as a baseline comparison for our own.

The tests we wrote consisted of 10 loops of a Thread Group of 5 total threads (simulating 5 users) with a ramp-up period of 5 seconds. This Thread Group was used for testing the 3 scenarios, hence, the first few spikes in response times are likely due to the thread overlap in between tests. Below are the Response Time Graphs for each of the interactions with ReCollect as well as the max, minimum and average times. One note about the obtained results we couldn't grasp was the differing max

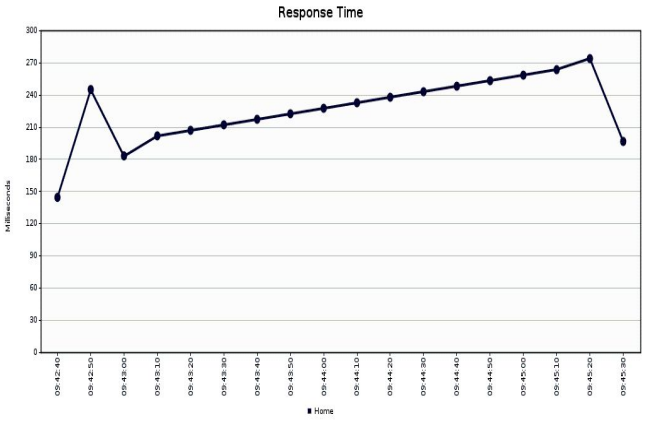
response times in different types of plots JMeter produced, but, for this analysis, we considered the maximum of the two. All of the results as well as the scripts can be found under /reporting/JMeter in the repository.

4.5.1 Accessing Home

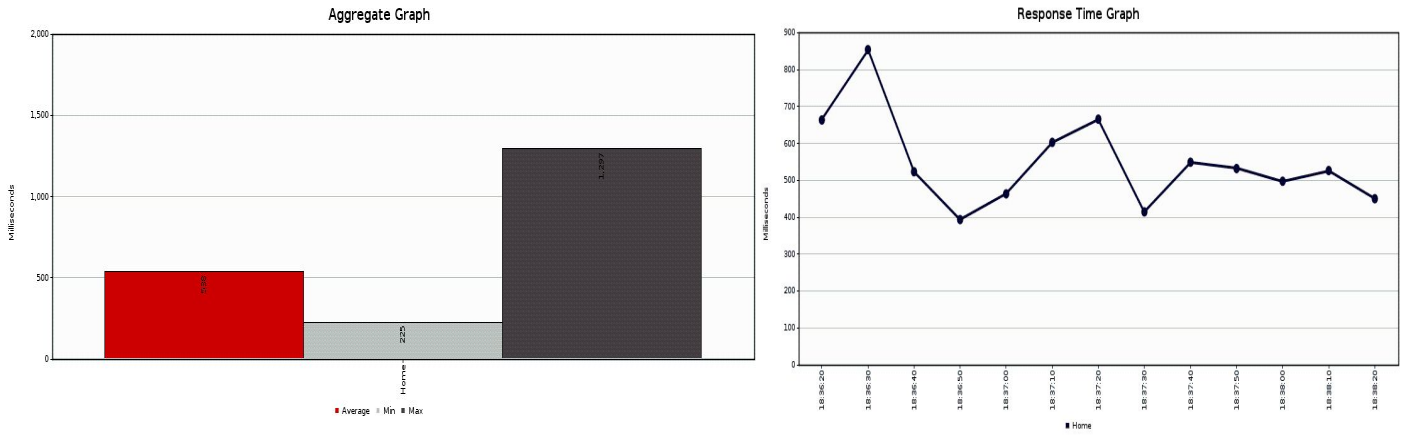
From the presented results we can infer that despite the sudden spike in Response Time, we obtain fairly solid results. The load increases as more threads and requests are made, as expected, but in a linear fashion, without too many hints of a potential exponential time complexity. Furthermore, we notice in the first graph a maximum response time of 800 ms, which we consider that poorly represents the accuracy of the results obtained, considering how far it is from the average and minimum results. If we consider the results obtained with OLX's performance when fetching the home page with the same conditions, we notice overall smaller values for the aggregate graph. As to the



response time, we notice some stabilization over time, but higher values than ours regardless.



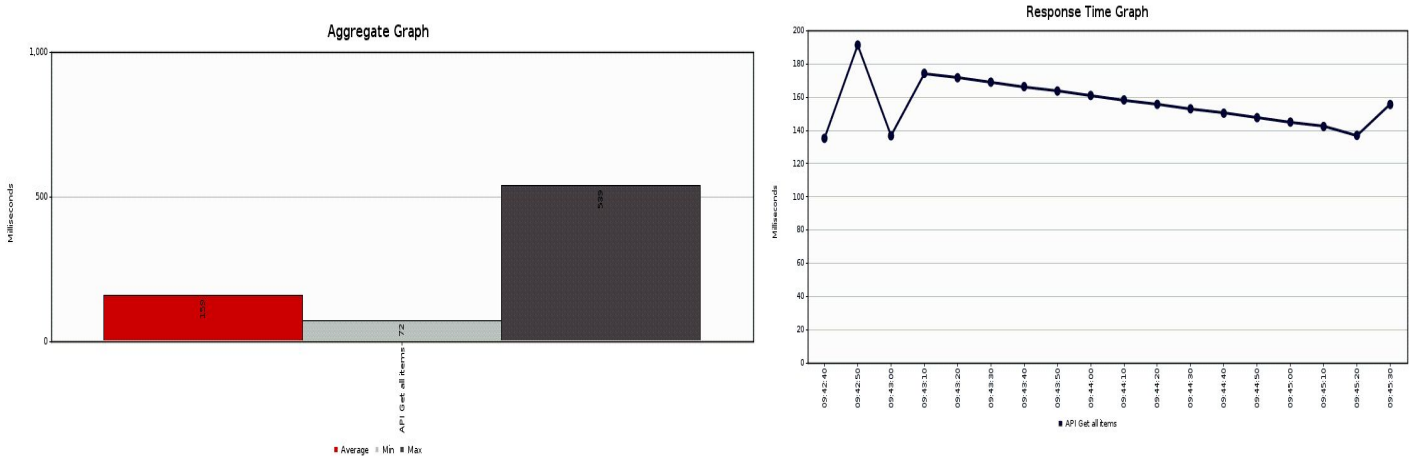
ReCollect Performance Times when accessing Home page



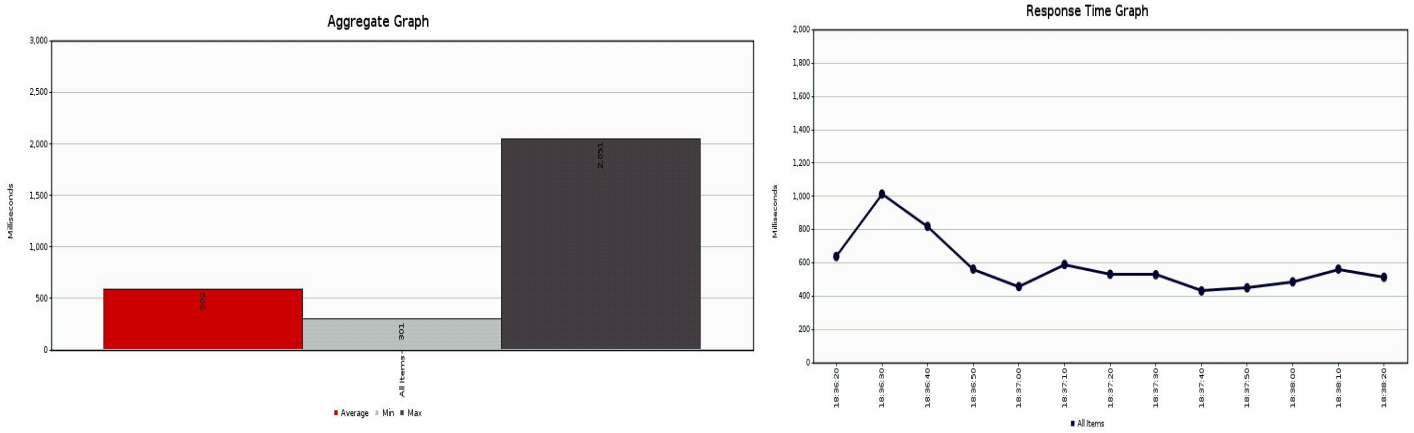
OLX Performance Times when accessing Home page

4.5.2 All items

Since OLX does not provide an API, we extracted the response times when accessing a web page equivalent, like getting all the ads. As we can see in the above results, our API behaves fairly well, never exceeding the 200ms when fetching every item in the database. On the other hand, albeit similar behavior, OLX displays overall much superior values that don't display convergence to a lower one.



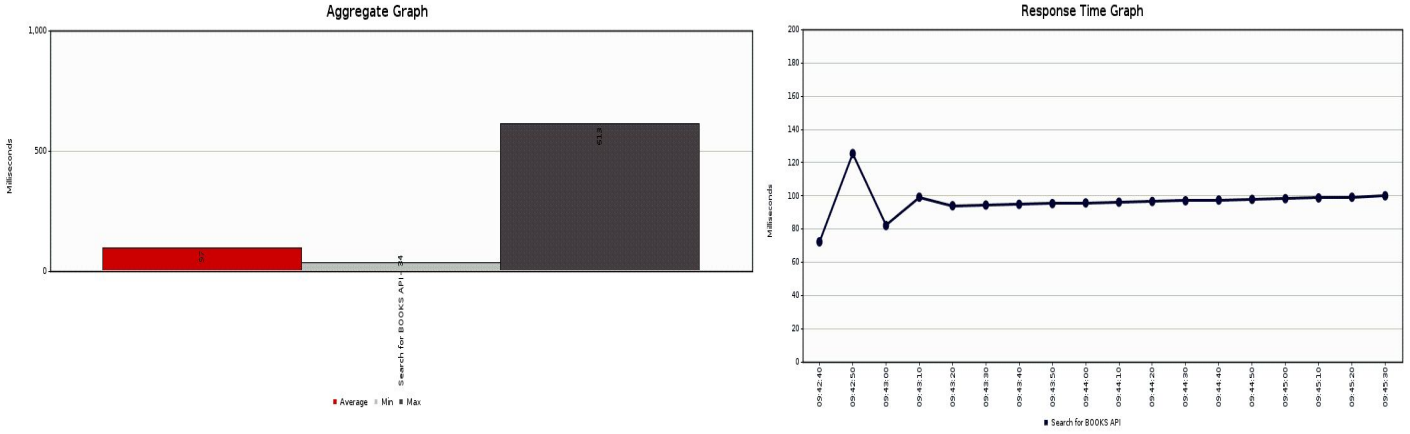
ReCollect Performance Times when fetching every item



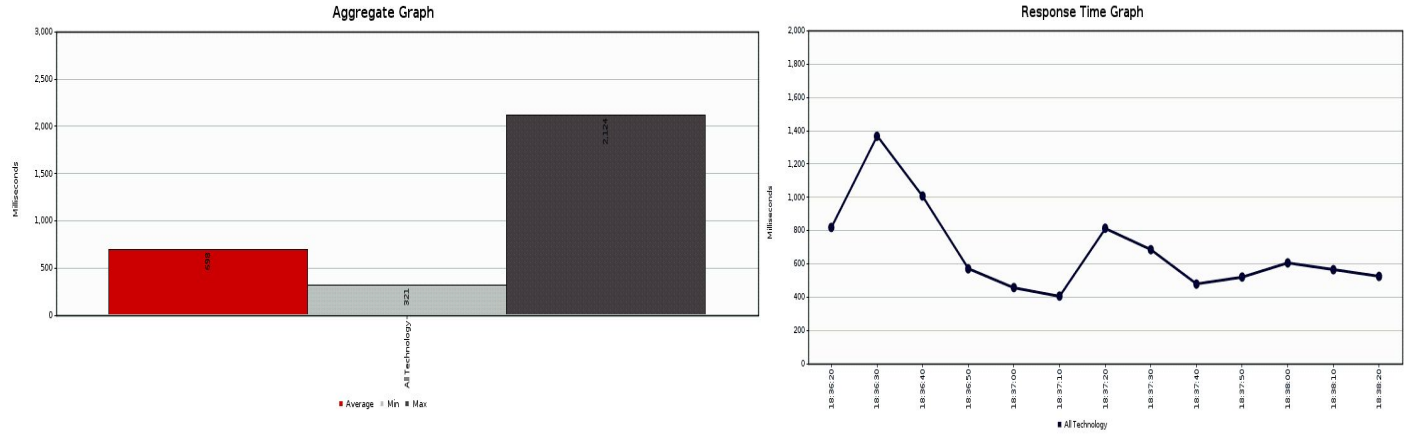
OLX Performance Times when fetching every item

4.5.3 All items of a given category

Lastly, we can notice once again significantly smaller average response times in our platform when compared to OLX. However, we notice a slightly increasing trend over time whereas in OLX we can infer that, despite the occasional spikes, response times tend to lower over time.



ReCollect response times when fetching every book using the API



OLX response times when fetching every Tech item via webpage

From the presented results, we can conclude our platform provides reliable and efficient access for common users given our current data volume. Naturally, these values most likely will not remain once ReCollect achieves the dimension OLX has, but, for development and testing purposes, we consider these results positive. Furthermore, we can also notice in some cases, small growing trends, which can indicate that this data increase can be supported and tolerated, in other words, we conclude our platform is scalable performance-wise.