UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Kert Prink

A dashboard to visualize the product quality level

Master's Thesis (30 ECTS)

Supervisor: Ezequiel Scott, PhD

Tartu 2021

A dashboard to visualize the product quality level

Abstract:
Software releases nowadays happen faster than ever. People responsible for the software products are called Product Owners (POs) in a modern popular framework called Scrum. The role of the PO is difficult because it requires making decisions based on the business and technical data. Successful POs need an overview of all the variables to make decisions with the best value. Software quality is one such variable that the POs must consider while making decisions. There are tools that provide data about the software quality, but there is a lack of tools that give a straightforward overview of the quality in a way that is easy to understand. In this thesis, an open-source software application was delivered that provides a straightforward overview of the quality level on each release. The application was designed so that additional future developments could be easily made. Firstly, the requirements for the application were elicited. Secondly, the application was implemented. Next, the application was tested to verify functional correctness. And finally, the application was evaluated with a survey study along the POs in the industry to understand the perceived usefulness and ease of use of the application.

Keywords:
Product Quality Dashboard, Product Owner, software quality level, Hexagonal architecture

CERCS:
P170 Computer science, numerical analysis, systems, control

Toote kvaliteeditaseme visualiseerimise tööriist

Lühikokkuvõte:

Tarkvara väljalasked toimuvad tänapäeval kiiremini kui eales varem. Inimesi, kes vastutavad tarkvaratoote eest modernses raamistikus Scrum, kutsutakse Tooteomanikeks. Tooteomaniku roll on keeruline, sest see nõuab ärilistele ja tehnilistele andmetele tuginedes otsuste tegemist. Edukad Tooteomanikud vajavad ülevaadet kõikidest muutujatest parimate valikute tegemiseks. Tarkvaratoote kvaliteet on üks muutujatest, mida Tooteomanik peab arvestama otsuste tegemistel. On olemas tööriistad, mis annavad andmeid toote kvaliteedi kohta, aga puuduvad tööriistad, mis annaks kiire ülevaate toote kvaliteedist nii, et sellest oleks lihtne aru saada. Selle magistritöö raames loodi avatud lähtekoodiga tarkvararakendus, mis annab kiire ja lihtsa ülevaate tarkvaratoote kvaliteedist iga väljalaske kohta. Rakendus sai disainitud nii, et tulevikus oleks täiendavaid edasiarendusi lihne teha. Kõigepealt loodi rakenduse jaoks nõuded. Teise sammuna arendati rakendus. Järgmisena kontrolliti rakenduse funktsionaalsust. Ning lõpuks hinnati rakendust uuringu abil, millega kaasati Tooteomanikud IT sektorist. Uuring aitas mõista rakenduse tajutavat kasulikkust ja kasutusmugavust.

Võtmesõnad:

Tootekvaliteedi töölaud, Tooteomanik, tarkvara kvaliteeditase, Heksagonaalne arhitektuur

CERCS:

P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1 Introduction **6**
1.1 Problem statement 7
1.2 Thesis outline 7

2 Background **8**
2.1 Software product quality 8
2.2 Product Owner 9

3 Related work **11**
3.1 Tools in the industry 11
3.2 Academic work 12

4 Method **13**
4.1 Overview of the system 13
4.2 Requirements elicitation 16
4.3 Architecture 16
4.4 Implementation 16
4.5 Testing 17
4.6 Evaluation 18

5 Results **21**
5.1 Overview of the system 21
5.2 Elicited requirements 23
5.2.1 User stories 23
5.2.2 Use cases 24
5.2.3 Non-functional requirements 29
5.2.4 Traceability 31
5.2.5 Prototype 32
5.3 Architecture 33
5.4 Implemented system 36
5.4.1 PQD-API 36
5.4.2 PQD-DB 44
5.4.3 PQD-Front 46
5.5 Test results 48
5.5.1 Unit and integration test 48
5.5.2 System test 49
5.5.3 Fixes after tests 50
5.6 Evaluation results 51
5.6.1 Background of the POs 51

5.6.2 POs feedback on the PQD 53

6 Discussion **56**
6.1 Limitation 56
6.2 Lessons learned 56
6.3 Future work 57

7 Conclusion **59**

References **60**

Appendix **64**
I. Repositories 64
II. System test report 65
III. Licence 81

1 Introduction

Software development nowadays is fast-paced with weekly or even hourly deployments [1]. Short cycle times allow the companies to deliver software as the smallest useful and functioning product with an aim to monitor the customer reactions and tweak the product in the next releases [2]. Releasing software faster than competitors provides an advantage on the market [3]. Organizations that stay ahead of their competitors make deployments up to 30 times faster [4]. The quality of the software could suffer from such fast deliveries; therefore, it is important to keep track of the quality throughout the releases [3, 5]. Automation can help teams to monitor quality in a systematic and financially feasible way [5].

Product Owner (PO) is a role in a Scrum team, whose responsibility is to define how the product looks like and what features it holds; he/she is responsible for the product throughout its life-cycle [6]. PO role is considered difficult because it requires skills to be the middle-man and understand both sides of the project - development side and management or business side [7, 6]. The stakeholders expect the POs to deliver the highest value product possible [6]. To do so, they need data to support their decisions. There are tools like SonarQube [1] that provide raw data of quality, but there is a lack of tools that provide a higher-level overview of the quality, which can be synthetically and intuitively understood by POs [1, 7]. Furthermore, the existing tools do not provide flexible architecture that allows connecting different sources of information together without relying on one specific as a base.

In this thesis, a software solution is presented that provides a straightforward overview of the software quality level of each release. ISO/IEC 25010 divides the quality into eight characteristics: functionality, performance, compatibility, usability, reliability, security, maintainability, and portability [8]. The solution presented in this thesis provides an extra layer for the existing solutions to collect measurements of different quality characteristics, which are displayed on a dashboard in a synthetic and intuitive way. One value for quality is visualized for quick and easy understanding of the quality. The solution is implemented using an architecture that supports connecting different sources of information and is easy to extend in the future as needed. The solution was evaluated with test cases and real users regarding not only its functional correctness but also the usefulness in the real-world context. Testing ensured that the solution worked correctly. The survey study on the POs' validated the usefulness of the solution in the industry context. Possible future improvements were suggested after the evaluation.

[1] SonarQube `https://www.sonarqube.org/`

1.1 Problem statement

Product Owners (POs) are responsible for the quality of their product throughout its' life-cycle [6]. There are tools that provide information about the software product's quality, but these tools do not provide an overview of the quality that is quick and easy to understand by Product Owners [7]. This thesis focuses on POs, but the problem and the solution can be extended to the similar roles as well, such as Project Managers or Scrum Masters.

From the industry perspective, an open-source software solution is presented that provides a straightforward overview of software quality of each release. The solution is built with an architecture that is easy to extend and does not rely on one tool as a base. The main focus is that POs or other roles interested in quality could have higher-level overview of the software quality of each release. To make this happen, a software solution acting as a layer on top of existing tools, such as SonarQube and Jira [2], was created. That solution collects measurements from different existing tools and displays them on a dashboard. The solution is built so that connections to additional tools could be added easily.

POs and other roles interested in product quality can have an overview of the overall quality level of the product that the development team delivers. POs will be able to set priorities more easily as the tradeoff between the quality and the features are visualized. The development team can have an overview of which quality levels they deliver through different iterations that can be used as input during the team activities, such as refinements, plannings, or reviews.

From an academic perspective, the software solution will provide an implementation that can be aligned to ongoing research by M. Falco, E. Scott, and G. Robiolo which proposes an automated version of the Product Quality Evaluation Method (PQEM) [9].

1.2 Thesis outline

This thesis consists of five main sections. Section 2 provides an introduction to software quality, Product Owner, and how the two are related. Section 3 describes the tools used in the industry, academic papers, and how they are linked to the work in this thesis. Section 4 describes the methodology used in this thesis. Section 5 describes the results. Sections 4 and 5 have consistent sub-sections - the methodology describes what and how will be done, and the results describe what was actually delivered. Section 6 describes the limitations, lessons learned in terms of the process and the future work in terms of the developed application.

[2]Jira `https://www.atlassian.com/software/jira`

2 Background

This chapter gives an overview of the software product's quality and the role of a Product Owner (PO). A brief overview of the quality in software engineering is made. After that, the role of a PO is described in a Scrum team. It is described how the PO fits into the team, what are his/her responsibilities, and how the quality is associated with the PO.

2.1 Software product quality

Quality is relative and has no explicit definition [5]. It has remained a subjective term with several definitions, although there are numerous writings on the subject by experts like W. E. Deming or P. B. Crosby [10]. The word *quality* in technical context has a meaning of a characteristic that can satisfy a product's stated or implemented needs [10]. The quality is often though as follows:

1. quality is a measurement of satisfied and unsatisfied specifications of the product [10]; and

2. quality product is a product that satisfies customers expectations during the use or consumption of the product [10].

In software engineering, quality can be defined as a *"degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value"* [8]. ISO/IEC 25010 divides the software quality into eight categories: functionality, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability [8]. The ISO/IEC 25010 standard supersedes the ISO/IEC 9126 standard [11].

Functionality is described as a degree to which software performs the stated and implied functions under specified conditions [8]. Performance efficiency describes how much resources the software uses under specified conditions [8]. Compatibility describes two aspects: how much information can the software exchange with other softwares and how well it can work with other softwares in the same environment or on the same hardware [8]. Usability describes how well the software can be used to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified use case [8]. Reliability describes how the software can perform specified functions under specified conditions in a specified time [8]. Security describes two aspects: how well the software protects information from unauthorized access and how appropriate the access levels are for each user [8]. Maintainability describes how well the software can be modified for improvement, correction, adaption to the environment, and adaptation to the requirements [8]. Portability describes how well the software can be transferred from one environment or hardware to another [8].

Measuring software quality provides visibility to the structural quality of the software, which then provides an insight into the business risks [11]. The Consortium for IT Software Quality (CISQ) specifies five quality characteristics that must be measured to keep track of the business risks [11]. These characteristics are functional size, maintainability, reliability, performance efficiency, and security [11]. Maintainability, reliability, performance efficiency, and security can be measured automatically with a set of architectural and coding based rules that are based on the common weakness enumeration (CWE) [11, 12]. Automation helps to measure the quality in a systematic and financially feasible way [5]. There are static analysis tools, like SonarQube, that measure the quality characteristics of a product automatically.

2.2 Product Owner

14th Annual State of Agile Report says that 95% of the respondents to their survey use agile methodologies [13]. Organizations, who use agile methods for development, most often use Scrum [6]. 14th Annual State of Agile Report says that 58% of agile practitioners use Scrum [13]. Scrum is a framework that helps to solve complex problems by iterative and incremental approaches [14].

The Scrum Teams consist of Developers, one Scrum Master, and one Product Owner (PO) [14]. Developers are responsible for creating incremental value on each iteration [14]. Scrum Master is responsible for the establishment of Scrum in the team and the effectiveness of the team [14]. PO is responsible for getting the best value for the product from the Scrum Team [14].

The PO role is crucial in the Scrum Team [6]. PO leads, manages, and makes decisions on the product throughout its life-cycle, while all of the decisions directly impact the products' value and profitability [6]. PO role is often compared to the classical Project Manager or Product Manager role, although the PO is a new role according to Scrum [15, 6]. PO is the person who must work with the Scrum Team on one side and the business people on the other side to maximize the value of the product [7]. PO represents all the stakeholders and their interests in his/her product [6]. PO maximizes the value based on the return on investment (ROI) formula, which is calculated by dividing the business value with development costs [6]. The ROI depends on the products' quality, features, content, and many other variables [6]. The ROI cannot always be measured, and the PO needs to make cognitive decisions based on feelings and past experiences [7]. To succeed as a PO, the PO needs data to make decisions [7], and the decisions must be respected by the entire organization [14],

One of the essential duties of the PO is to create a shared understanding between the business and technical sides of the organization [6]. The PO must understand the business needs, but he/she does not have to know technical details, although it would be beneficial [6]. It is said that the PO needs to look at all the trees in the forest, not just focus on one specific tree [6]. The PO must always be available for the development

team to ensure that the team focuses on the right things and does not waste time doing things that are not necessary [6]. If the PO does not have the technical knowledge as the Developers do, he/she needs to have support to understand the technical details to make good decisions [6].

The PO role is a difficult role, that requires working with many variables simultaneously [6]. It is necessary to provide the POs a higher-level overview of the data they need to make decisions [7]. The quality of a product is one of the technical data that the PO must consider while making decisions. There are many tools, like SonarQube, that provide data of quality. Still, there is a lack of tools that provide a higher overview of the quality, which can be synthetically and intuitively understood by POs [7].

3 Related work

This chapter gives a short overview of the tools in the industry and the related academic work. A brief overview of the concept of the tools is made, and the popular ones are shortly described. After that, a quick overview of academic work is given along side their relation to the work done in this thesis.

3.1 Tools in the industry

Automatic static analysis tools (ASATs) analyze and measure software's quality without executing it [16]. The ASATs can detect *bugs*, *defects*, *vulnerabilities*, *coding style*, *test coverage*, etc. [16]. ASATs can detect defects in the software cheaper and faster than humans would with code reviews or manual software testings [16]. There are widely used ASATs in the industry like SonarQube that can perform a wide range of analysis tasks [17]. Still, there is a lack of tools that can measure the whole set of quality characteristics [1].

ASATs are regularly used in three software development contexts: local environment, code review, and continuous integration (CI) [16]. Studies on ASATs usage show that the ASATs are most often used in the CI out of the three contexts [16]. The same research shows that the most popular ASATs used in the industry are FindBugs, Checkstyle, PMD, SonarQube, and ESLint [16].

FindBugs is an open-source ASAT that can look for code defects within Java code [18]. PMD is an open-source ASAT that can look for coding defects in Java code and few others [19]. FindBugs and PMD have limited usages compared to the SonarQube, but they can be integrated into SonarQube [20]. Checkstyle is a coding style checker for Java code [21]. ESLint is a popular JavaScript linter that can be easily customized and plugged into other tools, including SonarQube [22].

SonarQube is a widely used tool in the industry [17]. It is said that SonarQube is the most used ASAT in the CI context [20]. SonarQube includes popular rules from other tools, such as FindBugs, PMD, and ESLint [20]. SonarQube can analyze 27 different coding languages according to their documentation at the time of the writing [23]. SonarQube is not limited to measuring the quality characteristics; it can measure additional things, like test coverage and time to solve the technical debt [23]. SonarQube uses rules to enforce the coding standards; when a piece of code violates a rule, an issue is raised [20]. SonarQube can raise three types of issues associated with quality measures: bugs, vulnerabilities, and code smell [20]. These issues are meant to be read by developers or other people with technical backgrounds. Although the information from SonarQube can provide POs help to make decisions, it is often not used because the PO does not understand the technical details or does not have the time to dig into them.

3.2 Academic work

This thesis is partly a successor to the work of M. Falco, E. Scott, and G. Robiolo, called "Overview of an Automated Framework to Measure and Track the Quality Level of a Product" (AFMTQ). AFMTQ addresses the issue that there are no tools that provide an overview of the complete set of quality characteristics [1]. The AFMTQ is based on the Product Quality Evaluation Method (PQEM) [9], which provides a systematic overview of the products' quality on different iterations, and provides a single value of quality [1]. PQEM is built by following the ISO/IEC 25010 standard [9]. AFMTQ proposes to semi-automate the PQEM by automatically connect different ASATs to receive measurements of different quality characteristics, therefore minimizing the manual work [1].

The tool implemented in this thesis, called Product Quality Dashboard (PQD), uses the ideas from the AFMTQ and PQEM. The main idea is to collect data from multiple ASATs and provide one value of quality to give a straightforward overview of the quality level. PQD is open-source software whose maintainable architecture supports connecting multiple tools, ASATs or different kinds, to scale the system even more in the future. PQD runs automatically once the user has configured it. PQD provides an overview of different products' qualities, including one value of quality, of different releases in one place. Besides, PQD provides an overview of project data and the quality level trends useful for the POs.

Similar work is performed by M. Choras, R. Kozik, D. Puchalski, and R. Renk, called "Increasing product owners' cognition and decision-making capabilities by data analysis approach" (POCDC) [7]. The solution in the POCDC collects data from various tools, processes them to generate a set of more advanced metrics so that POs can look for correlations between them [7]. The idea that information must be collected from different sources is the same in the POCDC and the PQD. However, the solution in the POCDC focuses on the deeper analysis that can be done if the data from various sources are furthermore processed [7]. The POCDC seems to be a good solution for POs or analysts who understand technical details and want to dig deeper into the details of the data. To compare with, the PQD focuses on giving a straightforward overview of the quality to ease and speed up the decision-making process for the POs. Furthermore, the PQD is open-source software available in GitHub that is built so that additional developments could be easily made in the future.

4 Method

This chapter covers the method used to fulfill the task that is to deliver a software solution that provides an overview of the software quality of each release, and that is easy to extend in the future. This chapter is divided into six sections: 4.1 overview of the system, 4.2 requirements elicitation, 4.3 architecture, 4.4 implementation, 4.5 testing, and 4.6 evaluation.

Methodology of design science in information systems [24] was taken as a guideline to the software solution. Figure 1 describes the methodology of design science, outputs of each step in the context of the thesis, and output references to the sections in this chapter.

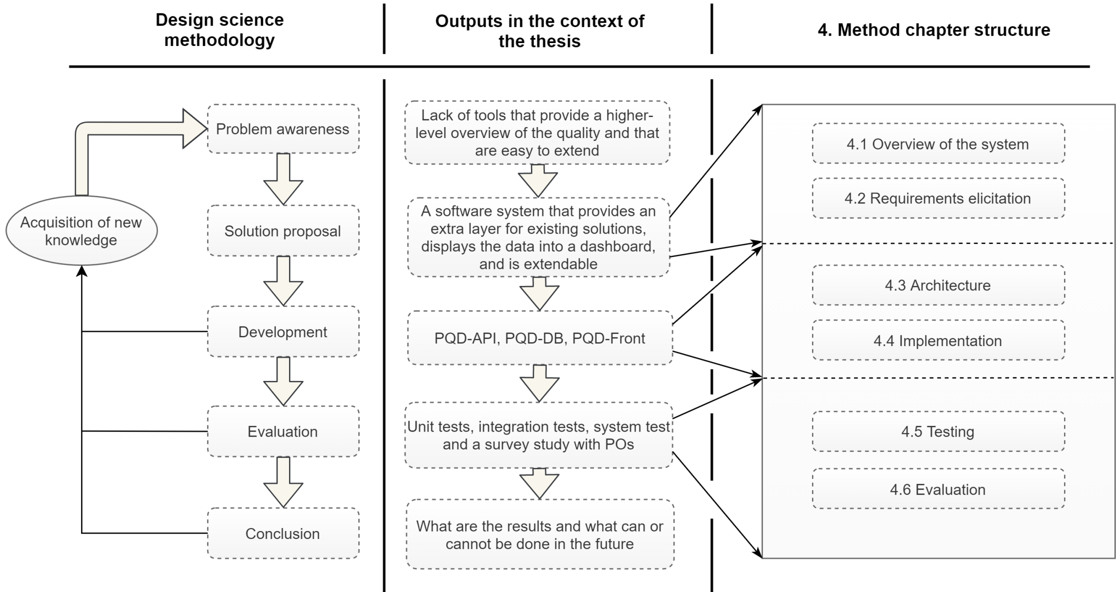

Figure 1. Design science [24] steps, outputs of the thesis, and chapter 4 structure

4.1 Overview of the system

The software solution, developed during this thesis, provides an extra layer for the existing solutions to collect measures of different quality characteristics. The solution collects the quality characteristics, calculates one value for quality and displays these values in a dashboard. The solution is implemented so that additional developments can be made with reasonable effort and connections to other tools could be added. The solution is called Product Quality Dashboard (PQD).

Figure 2 shows overview of the PQD system and how it fits into the development and release infrastructure. The PQD is meant to be used as an automated tool next to a release pipeline that triggers the PQD at some point of the process. Technically there are no restrictions who or what can trigger the PQD release info collection (RIC) as long as the request is authorized. The PQD collects and saves information from other tools, such as SonarQube, that are executed during the release process. The results are displayed in a dashboard.

Figure 3 shows an example of how the PQD fits into the release process of an agile development environment. After the development is done, the release pipeline starts. Pipeline performs various tasks and assesses code quality by running tests and static analyzers. If the assessment fails then, the pipeline exits without deploying code or sending a message to the PQD. Otherwise, the pipeline sends a message to the PQD and releases the code. The PQD collects data from various tools that are configured within the project, calculates quality, and saves the release info into the database. The release info from the database is visualized in the user interface.

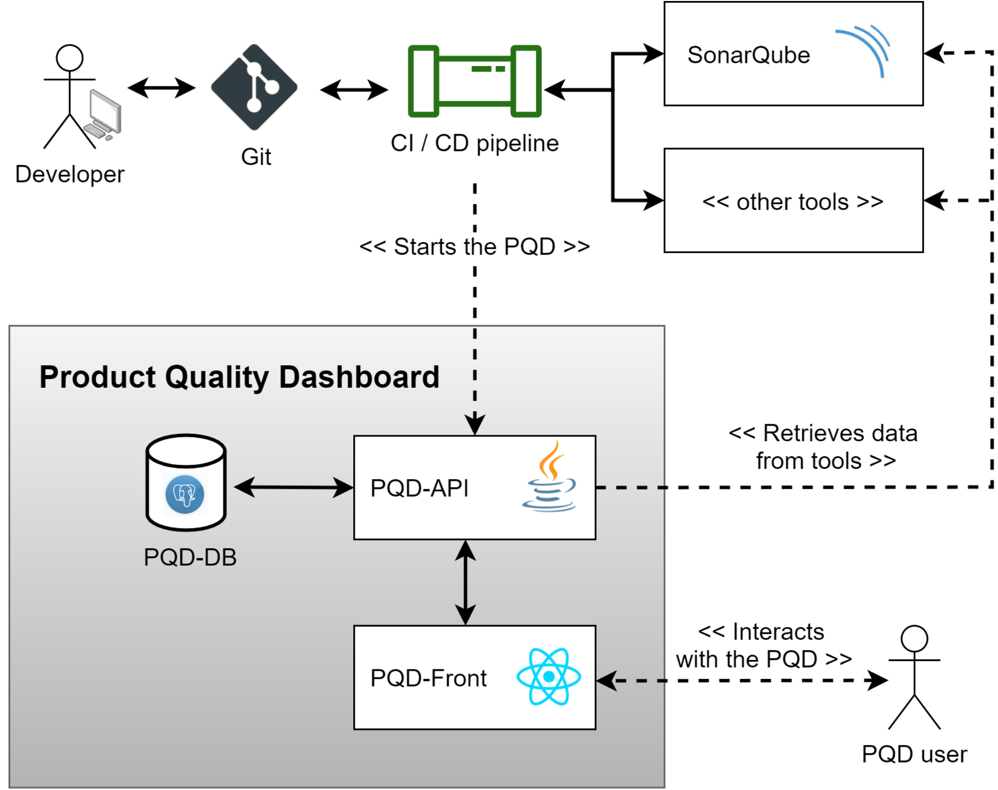

Figure 2. Overview of the PQD

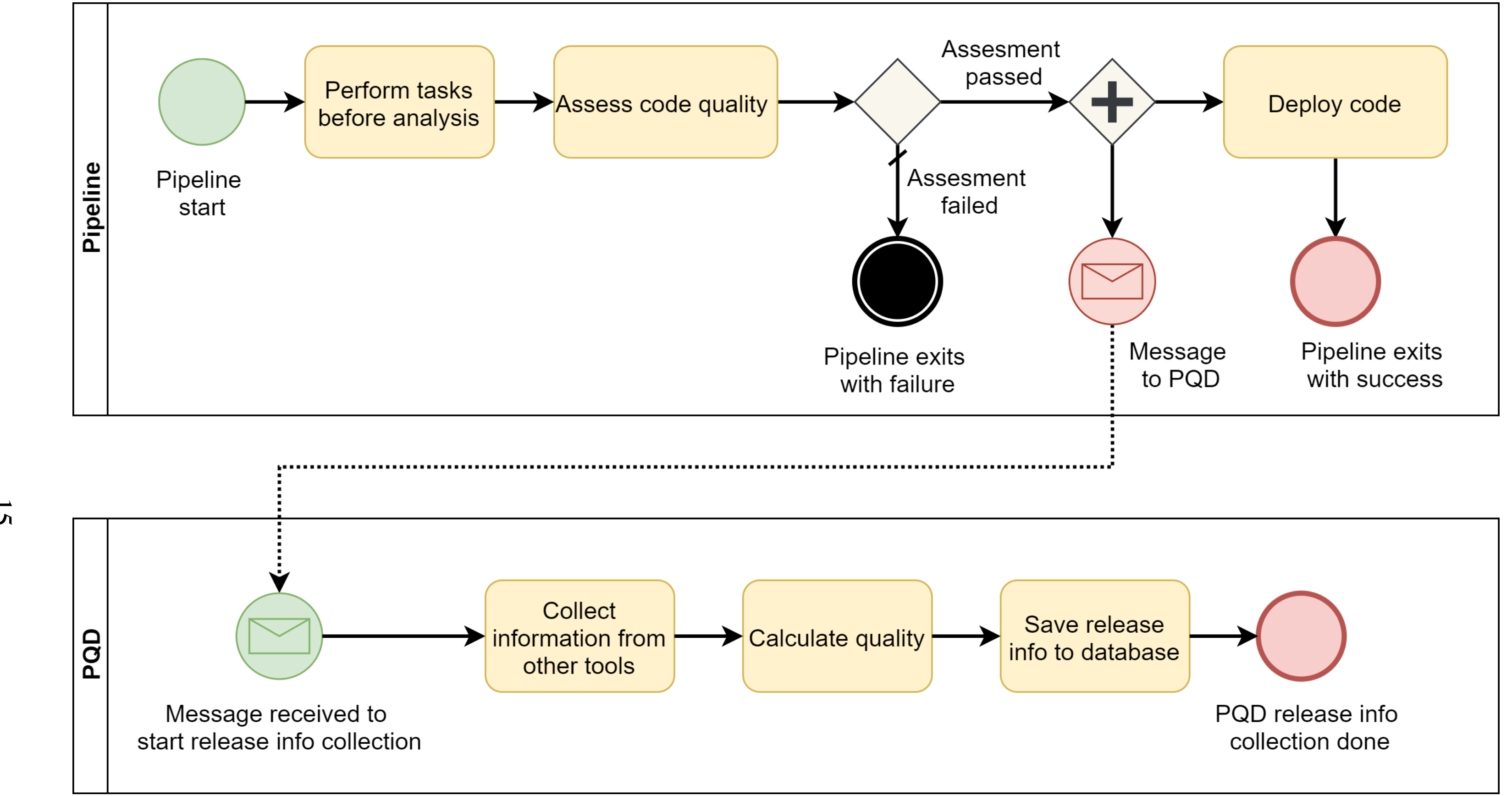

Figure 3. Release pipeline process with PQD

4.2 Requirements elicitation

User stories, use case modelling and prototyping techniques were used to create initial requirements for the product. Requirements traceability matrix was used to ensure that the requirements were linked to one another and that the functionalities of the implemented system were tested thoroughly to the specified requirements. Non-functional requirements were added next to the functional requirements to specify the system attributes such as security, maintainability, and scalability.

SonarQube was chosen to be the first tool that the PQD provided an extra layer for. SonarQube is a well known and widely used static analyzer in the industry [17]. SonarQube provides an API that can be used to retrieve data from the SonarQube server. Selected pieces of information were collected from SonarQube by the PQD. The PQD was designed so that additional tools could be added next to the SonarQube. Once the requirements regarding SonarQube were satisfied, then support for Jira Scrum board was added. The initial requirements considered SonarQube and omitted Jira.

4.3 Architecture

The PQD consisted of two main components: PQD-API [3] and PQD-Front [4]. PQD-API holds the business logic and provides a REST API to drive the application. PQD-Front is the user interface (UI) of the PQD system. The PQD system saves the data so that the data must be collected only once from the external systems.

The architecture for the PQD-API was chosen so that additional developments could be made with reasonable effort. These developments could add additional connections to external tools next to the implemented ones. Testability, maintainability, and scalability in terms of additional developments were the characteristics that the architecture shall support. Overview of different well-known architectural patterns were made to choose the pattern that fits the best for the PQD-API. Comparison of advantages and disadvantages were made after the overview.

4.4 Implementation

The implementation was split into two major parts: PQD-API and PQD-Front. PQD-API is the central part of the PQD system. PQD-API is written in Java [5] using Spring Boot [6] framework. PQD-API is the backend part of the application that contains the business logic. The PQD-API implementation involved the implementation of a database called

[3] PQD-API `https://github.com/Kert944/PQD-BE`
[4] PQD-Front `https://github.com/Kert944/PQD-Front-React`
[5] Java `https://www.oracle.com/java/`
[6] Spring Boot `https://spring.io/projects/spring-boot`

PQD-DB. Implementation of the PQD-API together with the PQD-DB was the main effort of the implementation part of the thesis.

PQD-DB was implemented together with the PQD-API. PostgreSQL [7] was used to implement the PQD-DB. PostgreSQL is a popular, mature, and open-source relational database [25]. Flyway [8] Gradle [9] plugin is used within the PQD-API for the migration of the database. PQD-DB migration SQL scripts are located within the PQD-API repository. The database contains information about users, user products, and release info of the products.

PQD-Front is the frontend part of the application. PQD-Front is a user interface that does not contain business logic. PQD-Front is written using React [10]. React is a Javascript library used for writing interactive and stateful frontend applications [26]. React can be used with third party libraries and frameworks [26]. CoreUI React components library [11] is used for the implementation of the user interface components within the PQD-Front. CoreUI provides free and ready to use UI components that are extended from Bootstrap to build web applications [27].

All the components of the PQD were put into Docker [12] containers to ease the deployment of the PQD to a cloud instance. To ease the development, PQD-API and PQD-Front could be run directly on the local machine, while the PQD-DB could be running in a container. Running the database in a container eases the development environment setup because the developers do not have to install the database on their local machine. However, when it comes to the PQD-API and PQD-Front, it is more efficient to develop while these components run directly on the local machine because the changes to the code take less time to take effect that way.

4.5 Testing

To evaluate the functional correctness of the PQD, different types of testing were used. Unit tests and integration tests were used to ensure that the system works technically. In addition, if some future developments should break some of the logic, then the tests will fail and indicate the developers that something went wrong. JUnit [13], a Java testing framework, is used for unit tests within the PQD-API. MockMVC [14], a Spring framework testing class, and TestContainers [15] Java library are used for the integration

[7] PostgreSQL https://www.postgresql.org/

[8] Flyway https://flywaydb.org/

[9] Gradle https://gradle.org/

[10] React https://reactjs.org/

[11] CoreUI https://github.com/coreui/coreui-react

[12] Docker https://www.docker.com/

[13] JUnit https://junit.org/junit5/

[14] MockMVC https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/servlet/MockMvc.html

[15] TestContainers https://www.testcontainers.org/

tests. Testcontainers is a Java library that provides lightweight, throwaway instances of databases, or anything that can run in Docker containers [28]. PQD-DB runs with the Testcontainers, to provide an isolated instance of the database. MockMvc is used to run the tests on the REST API endpoints provided by the PQD-API.

Manual system test was done to ensure that the elicited requirements were satisfied by the implemented system and that the system was ready to use. Testing environment for the manual system test was set up on a local machine. The elicited requirements were taken as a base for the system test cases.

4.6 Evaluation

A survey with Product Owners (POs) was made using technology acceptance model (TAM) to evaluate the potential usage of the PQD in the industry. The TAM introduces two variables that affect the users attitude towards the using the system, and therefore an actual system usage [29]. These two variables are called *perceived ease of use* (PEU), and *perceived usefulness* (PU) [29]. The *perceived ease of use* is the degree to which the user thinks that using the system is easy and effortless [29]. The *perceived usefulness* is the degree to which the user thinks that using the system improves his/her performance [29].

The survey is made using a questionnaire, where POs had to try out the PQD with some predefined data. The PQD was deployed to a cloud instance to make it available for the participants. The survey consisted of two parts: background of the PO, and evaluation of the PQD.

The aim of studying the background of the PO was to understand how they currently operate, and to evaluate if the respondent was an actual PO that fitted in the scope of the survey. The questions in the background part were the following:

1. What is your job title?
2. what is the domain of your product(s)?
3. What are your responsibilities?
4. How long have you been in the role at your current company?
5. Where are you located now?
6. How large is your team?
7. Do you keep track of the quality of your software products (quality definition is given before the question)?
    (a) If yes, then:

i. Do you use automatic tools? For example SonarQube, Coverity, etc.

A. If yes, then:

- What tool(s) do you use to keep track of the quality?
- What do you like about the tools you are using?
- What don't you like about those tools?
- How else do you keep track of the quality?

B. If no, then:

- How do you keep track of the quality?

(b) If no, then:

i. Why don't you keep track of the quality of your software products?

The aim of the PQD evaluation part was to understand if the PQD has potential usage in the industry, and what could be done to improve the PQD. The evaluation part started with a description of the PQD, which was followed by asking the POs to try out the tool with some predefined data. After the POs have tried out the PQD, a set of *likert* scale questions followed to understand the PU and PEU. Finally, a free form textfields were added in case the POs would like to add anything else. The questions in the PQD evaluation part were the following:

1. *Likert* scale from 1 to 5, strongly disagree to strongly agree

   (a) I understand how to use PQD. (PEU)

   (b) Using the PQD gives me a better overview of my products. (PU)

   (c) Using the PQD improves my performance. (PU)

   (d) PQD addresses my job-related needs. (PU)

   (e) It is easy to compare the quality levels of different releases of a product. (PEU)

   (f) It is easy to compare the quality levels of different products. (PEU)

   (g) I can understand the quality level of a product on each release. (PEU)

   (h) I understand the composition of the quality level of a release. (PEU)

   (i) All the quality characteristics that I am interested in are considered. (PU)

   (j) All the project data that I am interested in are considered. (PU)

   (k) I would use this product. (neither PU nor PEU)

   (l) I would recommend this product. (neither PU nor PEU)

2. What would you change about this product? Give priorities if possible.

3. Feel free to add any additional comments or explain some of your answers if you like.

The survey was announced using different strategies. First, a list of emails of POs was built using contacts of the author and supervisor. In addition, to direct emails, email newsletters by the University of Tartu were used to distribute the survey.

5 Results

This chapter describes the results of the method described in chapter 4. The chapter is divided into six subsections:

- section 5.1 - overview of the system;
- section 5.2 - elicited requirements;
- section 5.3 - architecture;
- section 5.4 - implemented system;
- section 5.5 - test results;
- section 5.6 - evaluation results.

5.1 Overview of the system

The PQD system is made of three components: PQD-API, PQD-Front, and PQD-DB. PQD-API is the main component of the system. The PQD-API is implemented using Hexagonal architecture. PQD-Front is the user interface of the system. PQD-DB is the database of the system, that is used by the PQD-API. The system is open-source and available in two different repositories - PQD-API with PQD-DB repository, and PQD-Front repository. The repositories can be found in Appendix I.

Figure 4 shows high level architecture of the implemented PQD system. The system is put into three Docker containers: PQD-DB container, PQD-API container, and PQD-Front container. Running the system inside Docker containers makes the application easily portable. User interacts with the PQD through the user interface called PQD-Front. PQD-Front interacts with user and PQD-API. PQD-API holds the business logic and is responsible for security. PQD-API interacts with database and external systems. PQD-API can retrieve information from Jira and SonarQube. Connecting two different tools shows that the PQD can be extended. External systems can make HTTP request to PQD-API to trigger PQD-API release info collection (RIC). The RIC retrieves the data from external tools, makes calculations on the data and saves the data into the database.

The screenshots of the user interface can be seen with the system test report in Appendix II.

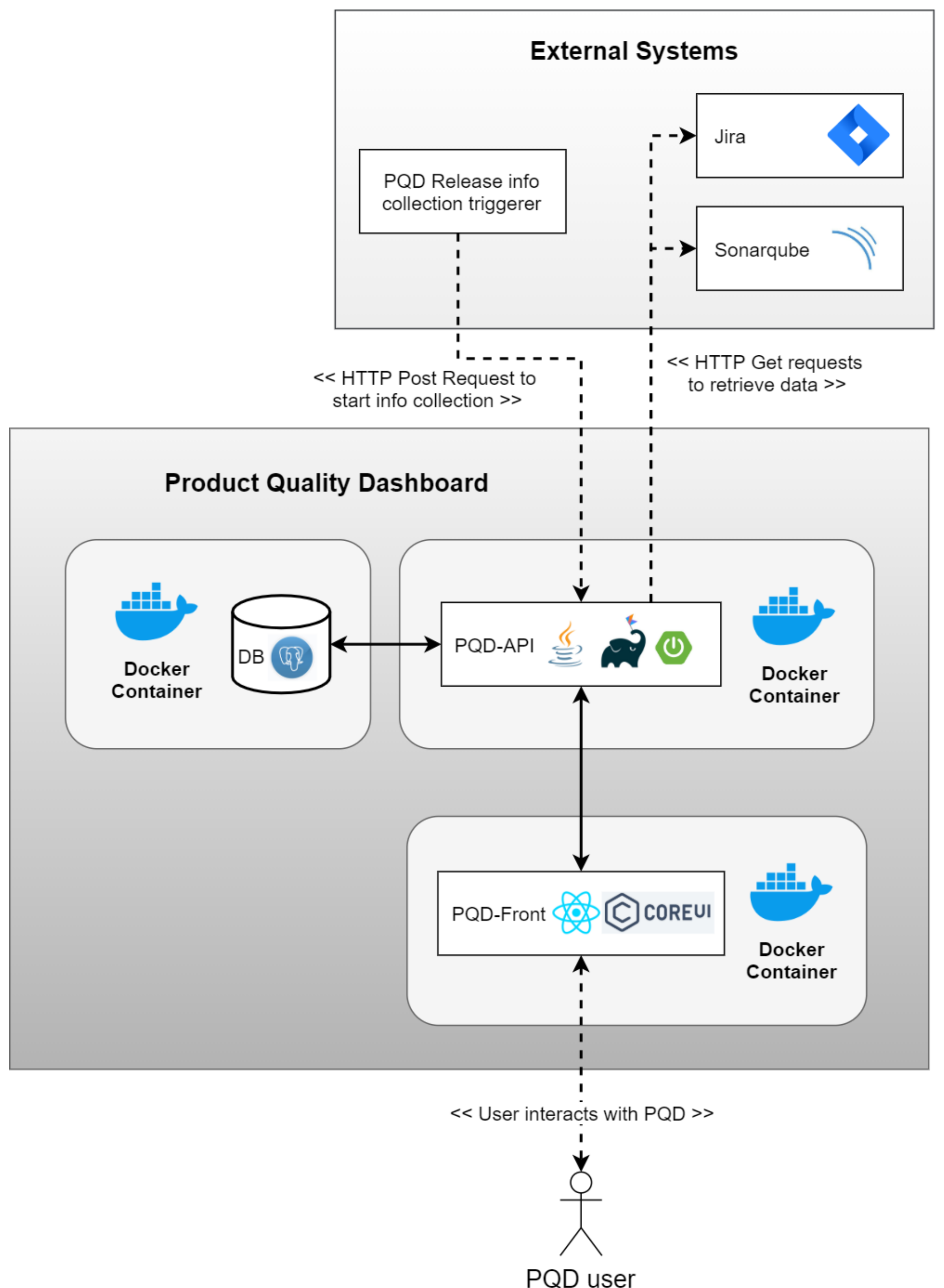

Figure 4. High level architecture of the PQD

5.2 Elicited requirements

User stories, use case modelling and prototyping techniques were used to elicit the initial requirements for the product. These techniques developed a specification that contains user stories, use case model, use cases, non-functional requirements, traceability matrix and a prototype of a minimum viable product (MVP). The elicited requirements contain requirements regarding SonarQube as it is the tool that is necessary for the MVP. Requirements regarding Jira are not considered a part of the MVP. Table 1 describes the terms used in the specification.

Table 1. Explanation of terms

Term	Explanation
PQD	Product Quality Dashboard
Home-view	View that contains a list of products that the user has right to
Product-view	View where the user is directed after clicking on a product at the home-view. This view contains information about the quality for the releases
General user	Any kind of user that uses the system. In this context, mainly a PO

5.2.1 User stories

Five user stories were elicited for the MVP that are described in table 2.

Table 2. User stories

ID	Content of the user story
US-1	As a PO, I want to have a list of my products, so that I can manage all my products from one place.
US-2	As a PO, I want to see the quality level by percentage of a given release, so that I don't have to look every characteristic one by one to have an overview
US-3	As a PO, I want to see information of quality characteristics of a given release, so that I can see what and how impacts the overall quality level
US-4	As a PO, I want to be able to choose a release from a history of releases, so that I can see the information of quality at a certain point of time
US-5	As a PO, I want to see a graph of quality levels of recent releases, so that I can easily see how the quality has changed over the time

5.2.2 Use cases

Eight use cases were elicited for the MVP. Seven use cases must be included with the product and one use case, UC-3, is an optional extra, which is not required for the MVP. Figure 5, use case model, shows the actor and which functionalities the system provides for the actor. There are three main use cases: adding a product, viewing a list of products, and viewing product quality in a detailed product view. Tables 3, 4, 5, 6, 7, 8, 9, and 10 describe the elicited use cases UC-1, UC-2, UC-3, UC-4, UC-5, UC-6, UC-7, and UC-8.

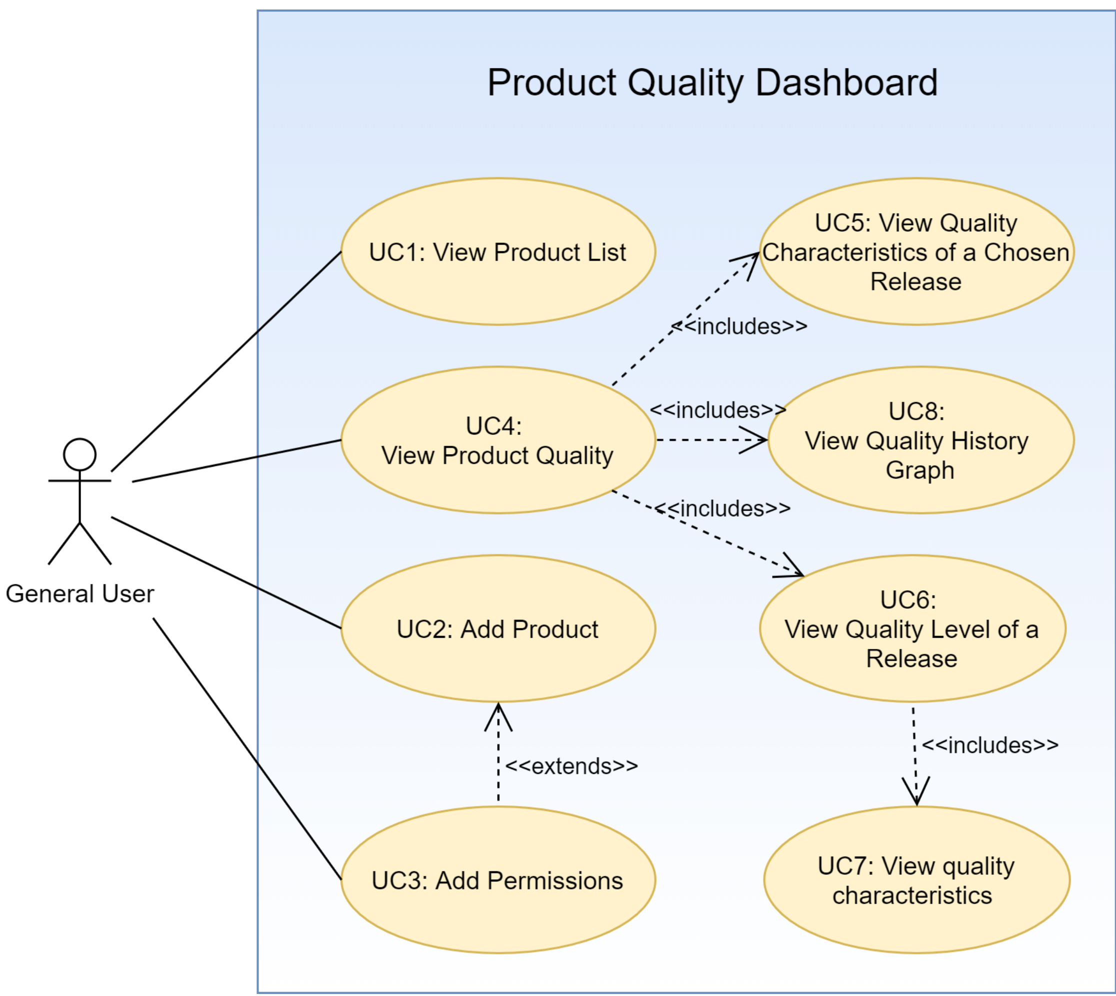

Figure 5. Use cases of the PQD system

Table 3. Use case 1

ID	**UC-1**
Name	Product list at home view
Description	User can see the list of his/her products at the home view. The list of products returned to the frontend dashboard are the products that the user has rights to.
Actors	General user
Preconditions	- User is authenticated
Primary scenario	1. User is directed to the home-view 2. The system retrieves user products from the database 3. The system displays the products
Postcondition	User sees a list of products that he/she is authorized for.
Priority	Must be included
Traceability	Satisfies US-1

Table 4. Use case 2

ID	**UC-2**
Name	Adding product to the system
Description	User can add a product to the system by configuring the necessary parameters.
Actors	General user
Preconditions	- User is authenticated - User is at the home-view where product list is displayed
Primary scenario	1. User is directed to the home-view 2. Modal is opened where instructions are displayed how to add a new product to the dashboard 3. User follows the instructions: Adds the name of the product, Adds base urls for connectors such as SonarQube API, Adds credentials 4. User saves the product 5. Loader is shown during saving process 6. The system saves the product into the database
Postcondition	Modal closes after successfully saving the product into the system and newly added product is displayed in the product list
Priority	Must be included
Traceability	Depends on UC-1

Table 5. Use case 3

ID	**UC-3**
Name	Permissions to the product for other users
Description	User can add permissions for other users for given product
Actors	General user
Preconditions	- Configured product exists in the system - Multiple users exist in the system
Primary scenario	1. User adds a permission for a user for a product 2. The system updates the product permissions in the database
Postcondition	The user, who got permission, can see the product and quality information about the product
Priority	Nice to have
Traceability	Depends on UC-1, UC-2

Table 6. Use case 4

ID	**UC-4**
Name	Quality overview at the product-view
Description	User can see overview of product quality at the product-view
Actors	General user
Preconditions	- Configured product exists in the system - The system has data for at least one release
Primary scenario	1. User navigates to the product view 2. The system retrieves product details from the database
Postcondition	User sees the overall quality level and info for each individual quality characteristics for the last release
Priority	Must be included
Traceability	Satisfies US-4, Depends on UC-1, UC-5, UC-6, UC-8

Table 7. Use case 5

ID	**UC-5**
Name	Quality overview of chosen release
Description	User can choose a release from a history of releases to see information about quality regarding the given release
Actors	General user
Preconditions	- Configured product exists in the system - The system has data for at least one release: data is retrieved when a release is made (release branch builds), data is saved into the PQD database
Primary scenario	1. User navigates to the product view 2. User chooses a release from a history of releases 3. The system retrieves the data of the chosen release
Postcondition	User sees information of quality for the chosen release
Priority	Must be included
Traceability	Satisfies US-3, Depends on UC-6

Table 8. Use case 6

ID	**UC-6**
Name	Quality level of a release
Description	The system must calculate a quality level of a given release. The calculation takes individual quality characteristics as an input and calculates them into one value. The quality level is displayed at the product view for the currently selected release and at the home-view for each product's last release.
Actors	PQD System
Preconditions	- Release has happened - PQD system receives an input to start fetching data for the release
Primary scenario	1. PQD system retrieves data for reliability, security and maintainability characteristics from SonarQube API 2. One value for quality is calculated 3. The retrieved values and calculated quality level is saved into the database
Postcondition	The quality level and individual characteristics are saved for the given release and can be accessed via PQD API
Priority	Must be included
Traceability	Satisfies US-2, Depends on UC-7

Table 9. Use case 7

ID	**UC-7**
Name	Individual quality characteristic - reliability, security, maintainability (SonarQube)
Description	Reliability, security and maintainability metrics are fetched for the given release from SonarQube
Actors	PQD System
Preconditions	- Configured product exists in the system
Primary scenario	1. The system receives a signal to fetch the data for the given release 2. The system fetches data about reliability, security and maintainability
Postcondition	Reliability, security and maintainability metrics are saved into the system for the given release
Priority	Must be included
Traceability	-

Table 10. Use case 8

ID	**UC-8**
Name	Quality history graph
Description	A graph of quality levels through the releases at the product view
Actors	General user
Preconditions	- Data for more than one release exists
Primary scenario	1. User navigates to the product-view
Postcondition	A graph is displayed for the last releases that shows the quality levels of each release
Priority	Must be included
Traceability	Satisfies US-5, Depends on UC-4, UC-6

5.2.3 Non-functional requirements

Four non-functional requirements (NFR) were elicited for the MVP. Two NFR-s must be satisfied with the system and two were optional extras. The system must have user authentication and authorization; and the system must be built with a clean architecture that allows future improvements. Table 11 describes the non-functional requirements.

Table 11. Non-functional requirements

ID	PQD-NFR1	PQD-NFR2	PQD-NFR3	PQD-NFR4
Name	Portability - Desktop and mobile support	Security - User authentication and authorization	Performance - System responsiveness	Maintainability / Scalability - Good architectural design and documentation
Description	The user should be able to use the system with desktop and with mobile view	Only authenticated and authorized users can access information about certain product	The system should respond to user actions in less than 1 sec. During actions that take more time, a loader is shown	The system should be built so that future improvements can be easily made. This means that the architecture should be so that additional interfaces can be coded easily next to the existing ones.
Success criteria	The same information is readable in both views - mobile and desktop. The FE is designed to scale differently with desktop and mobile view.	Requests without valid token receive Unauthorized response and requests with no rights receive unauthorized response	The system should respond to user actions under 1 second at least 90% of the time	The system is built using architecture that supports making additional developments. The system is documented
Priority	Nice to have	Must be included	Nice to have	Must be included

5.2.4 Traceability

Table 12, traceability matrix, shows traces of user stories and use cases. Reading the matrix goes from row to column. For example: "UC-1 satisfies US-1" or "UC-2 depends on UC-1".

Table 12. Traceability matrix

	US-1	US-2	US-3	US-4	US-5	UC-1	UC-2	UC-5	UC-6	UC-7	UC-8
UC-1	Satisfies										
UC-2						Depends on					
UC-3						Depends on	Depends on				
UC-4				Satisfies		Depends on		Depends on	Depends on		Depends on
UC-5			Satisfies						Depends on		
UC-6		Satisfies								Depends on	
UC-8					Satisfies				Depends on		

5.2.5 Prototype

The wireframes of the prototype were designed after the requirements specification. Figure 6 shows the home view, where the user has an overview of his/her products. Each product at the home view has a name, quality level of the last release and a miniature graph that shows quality levels of the last releases. Clicking on a product redirects to the product view.

Figure 7 shows the detailed product view. Quality level and individual characteristics are displayed for a selected release. Release can be selected from a dropdown menu. A graph shows quality levels of the last releases.

Figure 6. Prototype of home view

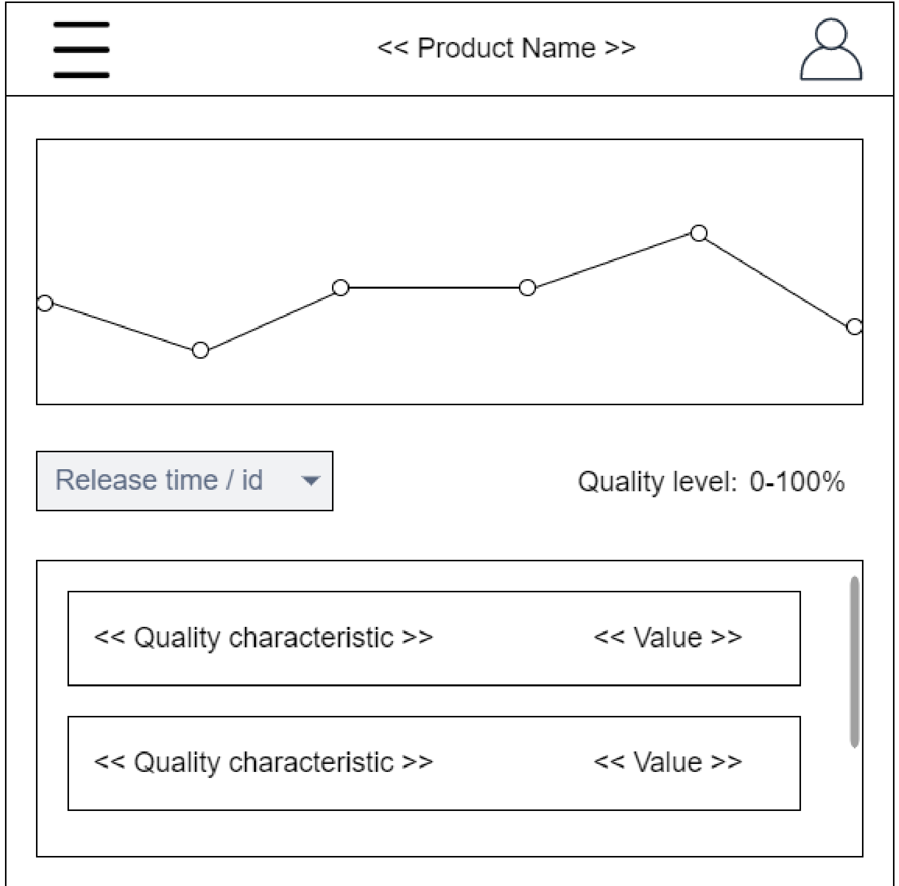

Figure 7. Prototype of product view

5.3 Architecture

Four different architectures are reviewed based on their advantages and disadvantages in this section to choose the architecture for the PQD-API. The following architectures are reviewed: n-tier architecture, Model-View-Controller (MVC) pattern, Onion architecture, and Hexagonal architecture a.k.a. ports and adapters pattern.

Historically, the n-tier architectures were thought of as physically separated components [30]. On the other hand, layers are logically, not necessarily physically, separated components [30]. The n-tier architectures were used before the modern architectures such as Model-View-Controller (MVC), Onion, and Hexagonal became widespread [31].

N-tiered architectures have had two fundamental issues through time: separation of concerns, and tight coupling [32]. *Separation of concerns* means solving the increasing complexity of growing software programs in small independent patches that are combined

afterward [33]. *Tight coupling* means that one object depends on the other objects, and changing one object requires changes in the other objects [32]. *Loose coupling* means that objects are independent and changes to one object do not require changes to other objects [32].

The n-tier applications have issues of tight coupling and separation of concerns [32]. N-tiered applications tend to have mixed business logic between the components. [32, 34]. It is difficult to keep clear boundaries between the layers in n-tier applications [34].

Model-View-Controller addresses *separation of concerns* by separating the presentation layer, business logic, and data access [32]. However, the *tight coupling* issue remains [32].

Onion and Hexagonal architectures address the *separation of concerns* by dividing the software into independent pieces [35], and both keep the components *loosely coupled* by introducing abstraction with interfaces [36, 37]. Onion and Hexagonal architectures make the systems:

- Framework independent that allows using frameworks as tools next to the system rather than using the framework as the system [35].
- Testable by separating the concerns of business logic, presentation layer, database, or any other element [35].
- Maintainable by pointing the dependencies towards the center of the application to allow changing outer components without changing the business logic [36, 35, 37].
- Scalable by introducing interfaces that are injected during runtime [36].

Although the Onion and Hexagonal architectures are very similar and both are called "clean architectures," there are some differences [35]. Onion architecture divides the application into layers, hence the name Onion [36]. Dependencies are pointed towards the center - the code in each layer can access the code in the same layer and the layers more inward, but it cannot access layers outward of its position [35]. Hexagonal architecture has no layers like Onion has. The Hexagonal architecture consists of business logic, input and output ports a.k.a. gateways, and adapters [35, 37]. Adapters are separated from each other, and they cannot access each other directly. Clear adapter separation makes the Hexagonal architecture more modular and maintainable than the Onion architecture [38]. More modules require more resources to build the application, giving the advantage to Onion in this case [38]. Although Onion and Hexagonal have more advantages over the N-tier and Model-View-Controller architectures, they require more effort to add new functionality because the developers must deal with abstractions and data mappings. Table 13 concludes the pros and cons of the reviewed architectures.

Table 13. Comparison of architectural patterns

Architectural pattern	**Advantages**	**Disadvantages**
N-tier	- Quick to develop	- Tight coupling - No separation of concerns
Model-View-Controller	- Separation of concerns - Quick to develop	- Tight coupling
Onion	- Separation of concerns - Loose coupling - Testable - Maintainable - Application build requires less resources than Hexagonal does	- Slow to develop - No explicit separation between components in the same layer
Hexagonal	- Separation of concerns - Loose coupling - Testable - Maintainable - More modular than Onion	- Slow to develop - Application build requires more resources than Onion does

The main component of the PQD system is the PQD-API. The PQD-API must connect with multiple external tools, provide REST API and a database connector. This indicates that the PQD-API will become a complex core of the PQD system. Onion and Hexagonal architectures are good choices to keep complex systems maintainable and testable. Hexagonal architecture introduces explicit boundaries in the form of adapters for the external interfaces that improve modularity. Onion architecture would keep the external interfaces in one layer. Absence of explicit boundaries introduces a threat of mixing up the code, and that reduces maintainability and testability.

Hexagonal architecture is chosen for the PQD-API. Although Hexagonal and Onion are very similar, the Hexagonal architecture's better modularity is the key in this decision. Hexagonal architecture supports making the PQD-API maintainable, scalable and testable. Java with Spring Boot is used to implement the Hexagonal architecture API. Spring Boot is a popular Java based framework that can be effectively used to create hexagonal architecture API's. Popular and mature technology for the solution, makes the solution more future proof, as there are community for support, good documentation and developers can be found more easily for future improvements. The decision to use Hexagonal architecture was partially influenced by the author's work experience since being familiar with the architecture ease the development efforts.

5.4 Implemented system

The implementation was divided into two major parts: PQD-API, along with the database called PQD-DB; and PQD-Front. The PQD-API and PQD-Front contained together roughly 14 000 lines of code. The links for the repositories can be found in Appendix I.

5.4.1 PQD-API

PQD-API was written in Java using Spring Boot framework and Gradle for build automation. The solution contains 161 Java files with over 9000 lines in total, including unit and integration tests. The API was implemented using Hexagonal architecture. The layers were separated by Gradle modules, which were composed together by another Gradle module to run the application. Table 14 describes the directories with modules in the PQD-API.

Figure 8 provides an overview of the high-level architecture of PQD-API. PQD-API application core is isolated from outer code by interfaces called gateways. Adapters implement these gateways. Web and Messaging adapters do not implement gateways, because it is not necessary as the directions of dependencies are correct - outside to inside. Use cases perform actions with domain objects, which hold the business knowledge. Use cases call gateways, which are implemented by adapters. Use cases do not know how the gateways are implemented.

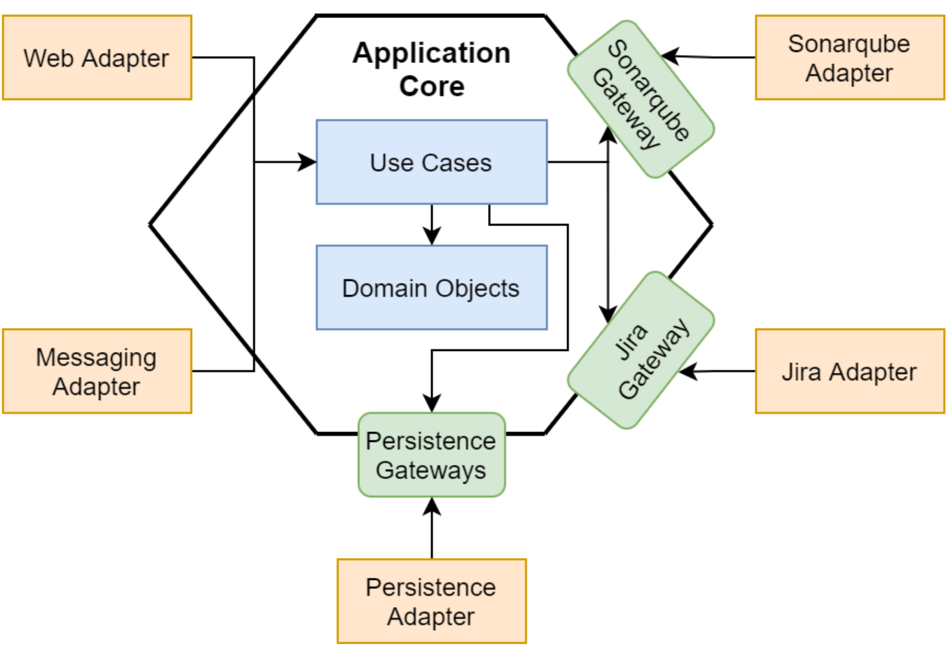

Figure 8. PQD-API high level architecture

Table 14. PQD-API directories containing Gradle modules

Directory		Description	Tests
adapters/		Parent directory for adapters	-
	messaging/	Rest API for triggering RIC. Requires basic authentication with a token	Unit tests
	persistence/	Adapter for database connection	Unit tests
	sonarqube/	Adapter for SonarQube connection. Retrieves code analysis results	Unit tests
	jira/	Adapter for Jira connection. Retrieves project info from Jira Scrum board	Unit tests
	web/	Rest API for web communication. Responsible for user authentication and authorization	Unit tests
application/		Core business logic	Unit tests
configuration/		Spring Boot module. Responsible for running the application.	Integration tests

Figure 9 describes the location of classes regarding the SonarQube information retrieval use case. Figure 9 is an example of how the use cases are located in modules and packages. Other use cases in the PQD-API follow a similar pattern of modularity as the *RetrieveSonarqubeData* described in figure 9. Use cases and business logic domain objects are located in the *PQD.application* module. Domain objects are located in *com.pqd.application.domain.** packages. Use cases and corresponding gateways are located in *com.pqd.application.usecase.** packages. The gateway implementers are located in *PQD.adapters.** modules. All the *PQD.adapters.** modules are dependent on the *PQD.application* module, but not on each other. The *PQD.application* module is not dependent on any of the *PQD.adapters.** module - the mapping into business logic domain objects are done in the *PQD.adapters.** modules, and the *PQD.application* module receives business domain objects from the interfaces that are implemented by the adapters.

The PQD-API calculates a quality level based on the data received from SonarQube. Figure 10 shows the code of the quality level calculation use case. Each use case in the PQD-API follow the same pattern as the use case *CalculateQualityLevel* shown in Figure 10 and use case *CollectAndSaveAllReleaseData* shown in Figure 13. *Request* and *Response* are static nested classes [16] within use cases. Use case specific *Request* class object must be given to the use case's *execute* method. Use case's *execute* method responds with a use case specific *Response* class object. The PQD-API contained 19 use cases in total at the time of the writing, 7th April 2021.

CalculateQualityLevel use case's *execute()* method, described in figure 10, uses

[16] Nested classes `https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html`

quality characteristic ratings retrieved from SonarQube that are in the original format. The original SonarQube ratings are numbers from one to five, with one being the highest and five the lowest score. The ratings from SonarQube are transformed to the scale of the PQD, with one being the best and zero the worst. At the moment, three quality characteristics are taken into account and assumed equally important; thus, the average is returned as a result.

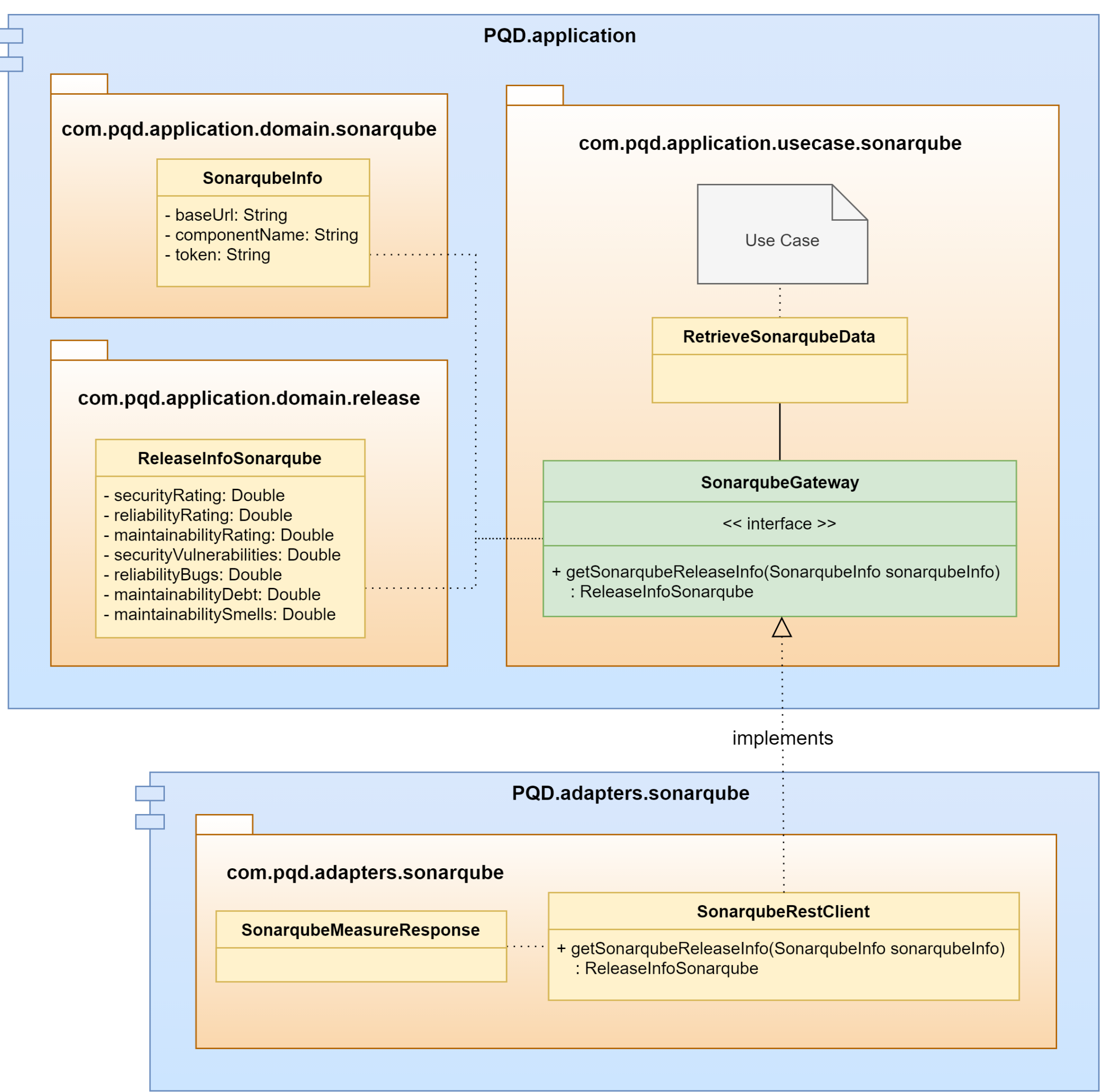

Figure 9. PQD-API SonarQube release info retrieval use case

```java
@RequiredArgsConstructor
@UseCase
public class CalculateQualityLevel {

    public Response execute(Request request) {
        int totalCharacteristics = 3;
        ReleaseInfoSonarqube releaseInfoSq = request.
            getReleaseInfoSonarqube();
        Double sqSecurityRating =
                transformSonarqubeRating(releaseInfoSq.
                    getSecurityRating());
        Double sqReliabilityRating =
                transformSonarqubeRating(releaseInfoSq.
                    getReliabilityRating());
        Double sqMaintainabilityRating =
                transformSonarqubeRating(releaseInfoSq.
                    getMaintainabilityRating());

        Double qualityLevel =
                (sqSecurityRating + sqReliabilityRating +
                    sqMaintainabilityRating) / totalCharacteristics;
        return Response.of(qualityLevel);
    }

    @{...}
    public static class Request {...}

    @{...}
    public static class Response extends AbstractResponse {...}

    private Double transformSonarqubeRating(Double rating) {
        if (rating == 1.0) {
            return 1.0;
        } else if (rating == 2.0) {
            return 0.75;
        } else if (rating == 3.0) {
            return 0.5;
        } else if (rating == 4.0) {
            return 0.25;
        } else {
            return 0.0;
        }
    }
}
```

Figure 10. Quality level calculation use case

The PQD-API application domain consists of two major objects: *Product* and *ReleaseInfo*. Figure 11 describes *Product* and *ReleaseInfo* classes. *Product* is a PQD object that contains information about the product itself, *SonarqubeInfo*, and *JiraInfo*. Sonarqube info and Jira info contain information needed to retrieve data via APIs.

Each *Product* is associated with zero to many *ReleaseInfos*. *ReleaseInfo* is a PQD object that contains information from Sonarqube and Jira at a given time. *ReleaseInfo* contains calculated quality level at the given time. The quality is calculated based on the data retrieved from Sonarqube. The quality level is calculated with a use case called *CalculateQualityLevel* described in Figure 10. Release info from SonarQube is contained in *ReleaseInfoSonarqube*. *ReleaseInfoJira* contains zero or more *JiraSprints*. *JiraSprint* contains information from the Jira Scrum board that includes zero or more *JiraIssues*. *JiraIssue* contains information about Jira issues. Each Jira issue has fields that describe the issue. *JiraIssueType* describes the type of the issue.

ReleaseInfoSonarqube and *ReleaseInfoJira* contain selected information collected from SonarQube and Jira. These objects can be extended if needed. Additional associations can be added to *ReleaseInfo* if support for additional tools is added next to SonarQube and Jira.

Figure 12 describes the program flow when a request for RIC is received on the messaging adapter. Figure 13 describes the code of *CollectAndSaveAllReleaseData* that is a part of the flow shown in Figure 12. A pipeline is shown as an example request maker in Figure 12. Technically, every valid POST request would do the job, and it does not need to be originated from the pipeline. Authorization is the first step after the request is received in the *MessagingController*. HTTP unauthorized message is returned if the token check does not give authorization for the product that the request was made for. The program flow stops after returning an unauthorized HTTP response.

On the other hand, if the request is authorized for the given product, then a second Java thread is started that executes *CollectAndSaveAllReleaseData* use case asynchronously. The first thread that received the request returns HTTP ok message saying that the RIC was started. The first thread does not know if the collection is successful; it just returns a message saying that the program started the RIC. Asynchronous execution, in this case, allows giving a faster meaningful response than synchronous execution would. *CollectAndSaveAllReleaseData* use case (see Figures 12, 13) retrieves the specified product to read the information that is needed for RIC. Checks are performed to see if SonarQube and Jira are configured with the product. *RetrieveSonarqubeData* use case is executed if SonarQube is configured with the product. *RetrieveReleaseInfoJira* use case is executed if Jira is configured with the product. The naming of *RetrieveSonarqubeData* and *RetrieveReleaseInfoJira* is different, but the idea behind the names is the same. *SaveReleaseInfo* use case is executed with the collected data to save the release info at the given time after the retrieve release info use cases are executed. The program flow concludes after the saving.

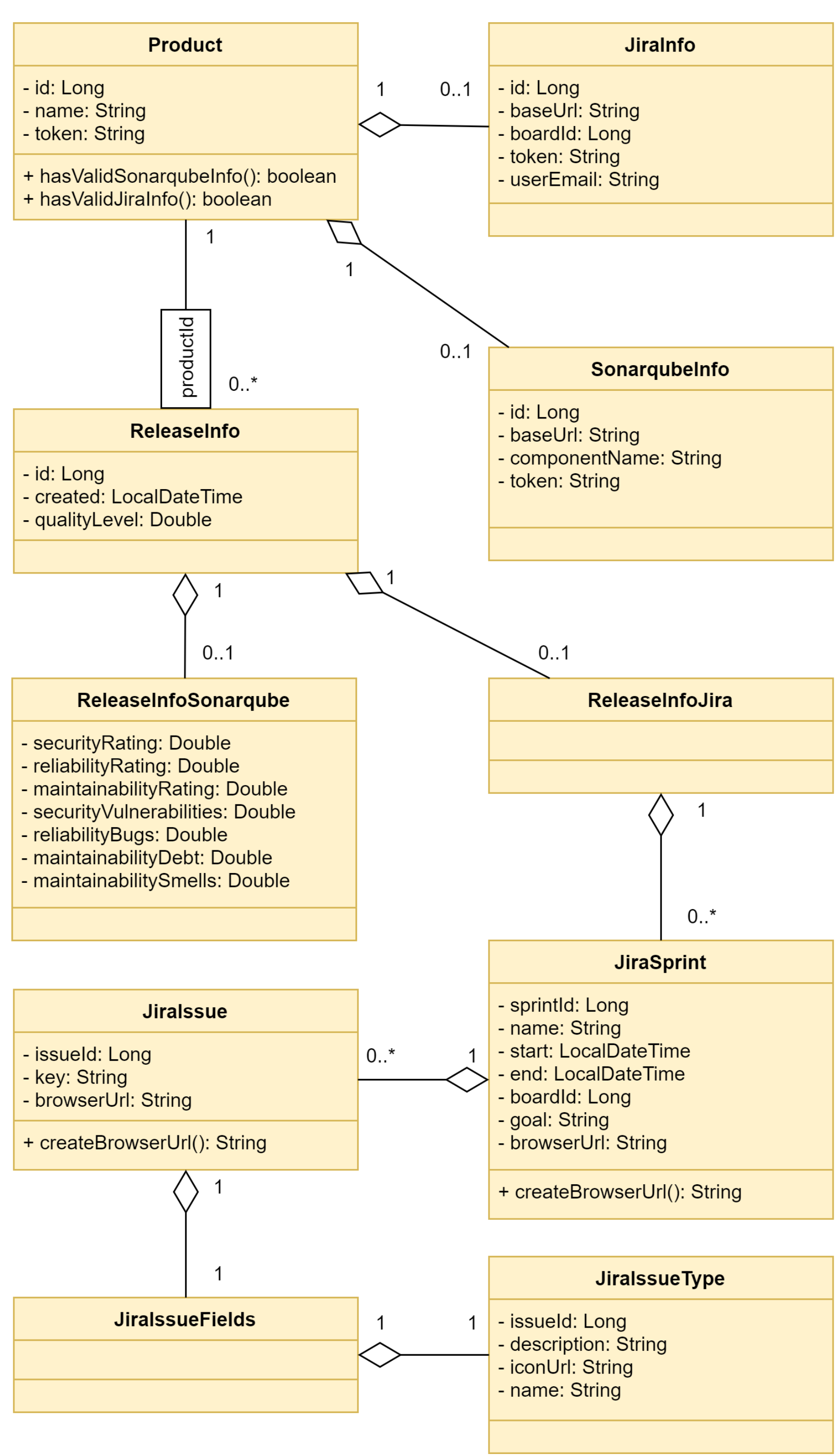

Figure 11. PQD-API diagram of product and release info classes

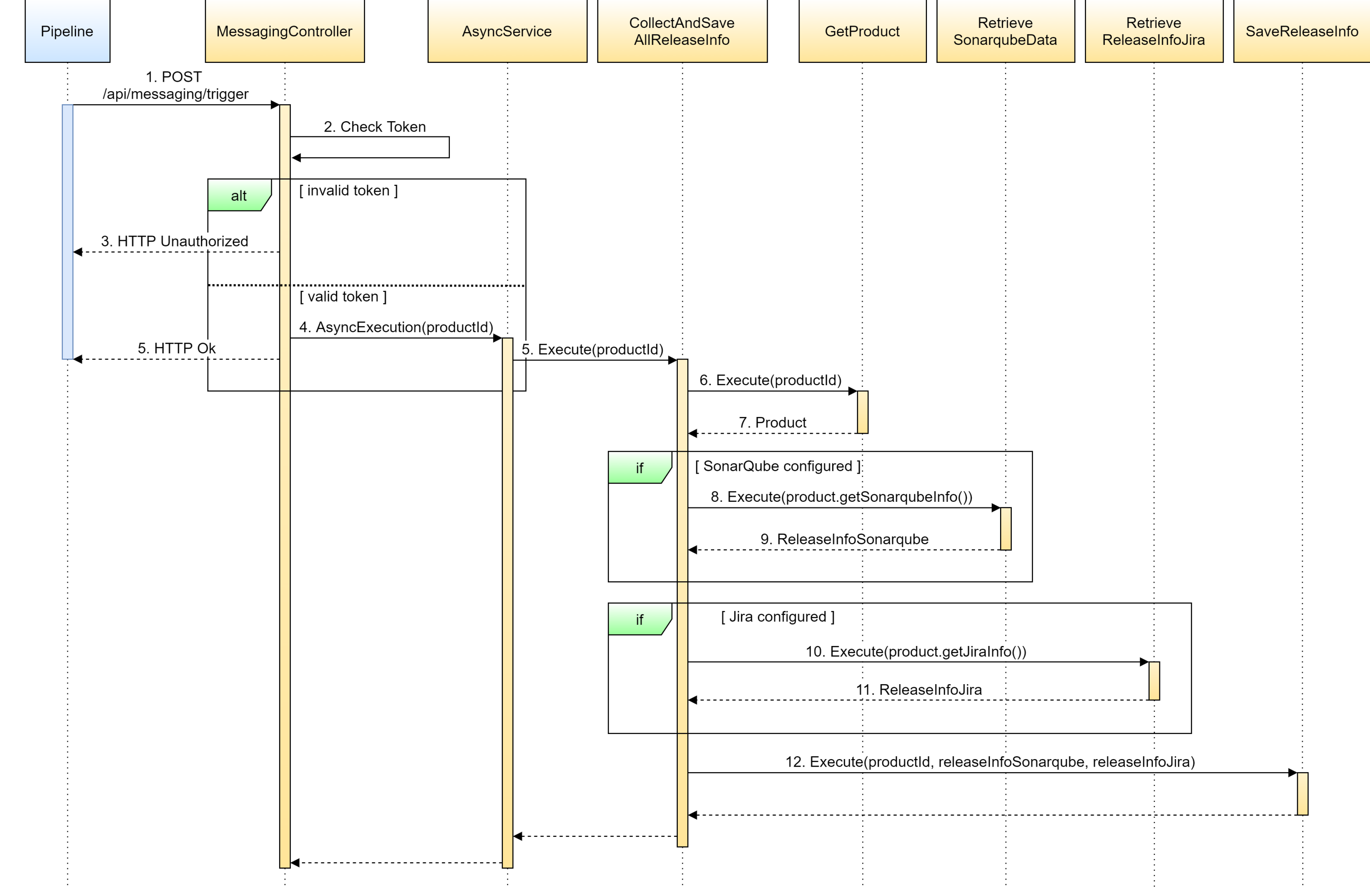

Figure 12. PQD-API sequence diagram of triggering the release info collection (RIC)

```java
@{...}
public class CollectAndSaveAllReleaseData {
    private final RetrieveSonarqubeData retrieveSonarqubeData;
    private final SaveReleaseInfo saveReleaseInfo;
    private final GetProduct getProduct;
    private final RetrieveReleaseInfoJira retrieveReleaseInfoJira;

    public void execute(Request request) {
        Product product = getProduct.execute(...);
        ReleaseInfoSonarqube releaseInfoSonarqube =
            ReleaseInfoSonarqube.builder().build();

        if (product.hasValidSonarqubeInfo() && ...) {
            releaseInfoSonarqube =
                    retrieveSonarqubeData.execute(...)
                                        .getReleaseInfo();
        }

        List<JiraSprint> activeSprints = List.of();
        if (product.hasValidJiraInfo() && ...) {
            activeSprints =
                    retrieveReleaseInfoJira.execute(...)
                                          .getActiveSprints();
        }

        saveReleaseInfo.execute(
            SaveReleaseInfo.Request.of(
            releaseInfoSonarqube,
            ReleaseInfoJira.builder()
                           .jiraSprints(activeSprints)
                           .build(),
            request.getProductId()));
    }

    @{...}
    public static class Request {
        Long productId;
    }

    @{...}
    public static class Response extends AbstractResponse {
        ReleaseInfo releaseInfo;
    }
}
```

Figure 13. Code of collecting and saving release info use case

5.4.2 PQD-DB

PQD-DB is PostgreSQL database that is connected to the PQD-API via the persistence adapter. Flyway Gradle plugin was used within the PQD-API to migrate the database. The migration contained definition of the schema and sample data insertion. The migration files are located inside the *configuration* directory of the PQD-API.

Figure 14 describes the PQD-DB schema that was extracted with IntelliJ [17]. The multiplicities were added with taking into account the business logic of the application. The application logic works around products that are defined with the *product* table. The *product* is associated with zero or one *sq_info*, meaning that one product can contain information about zero or one SonarQube component. The *product* is associated with zero or one *jira_info*, meaning that one product can contain information about zero or one Jira Scrum board. One *product* is associated with multiple *user_product_claim*'s that means claims for one product can be given to multiple users. One claim is associated with one *user* and one *product*. *User* is associated with multiple *user_product_claim*'s that means one user can have claims for multiple products.

One *product* is associated with multiple *release_info*'s, meaning that one product can contain information about multiple releases. *Release_info* is associated with zero or one *release_info_sq*, meaning that one release info can contain information about zero or one batch of information retrieved from SonarQube. *Release_info* is associated with multiple *release_info_jira_sprint*'s, meaning that one release info can contain information about multiple Jira sprints. *release_info_jira_sprint* is assiciated with multiple *jira_issue*'s, meaning that one sprint can hold information about multiple issues.

The schema can be extended to support additional tools next to SonarQube and Jira. To do that, a new association to the *product* table with the information about the new tool must be added. Then, association with the new tool's release info must be added to the *release_info* table.

[17]IntelliJ `https://www.jetbrains.com/idea/`

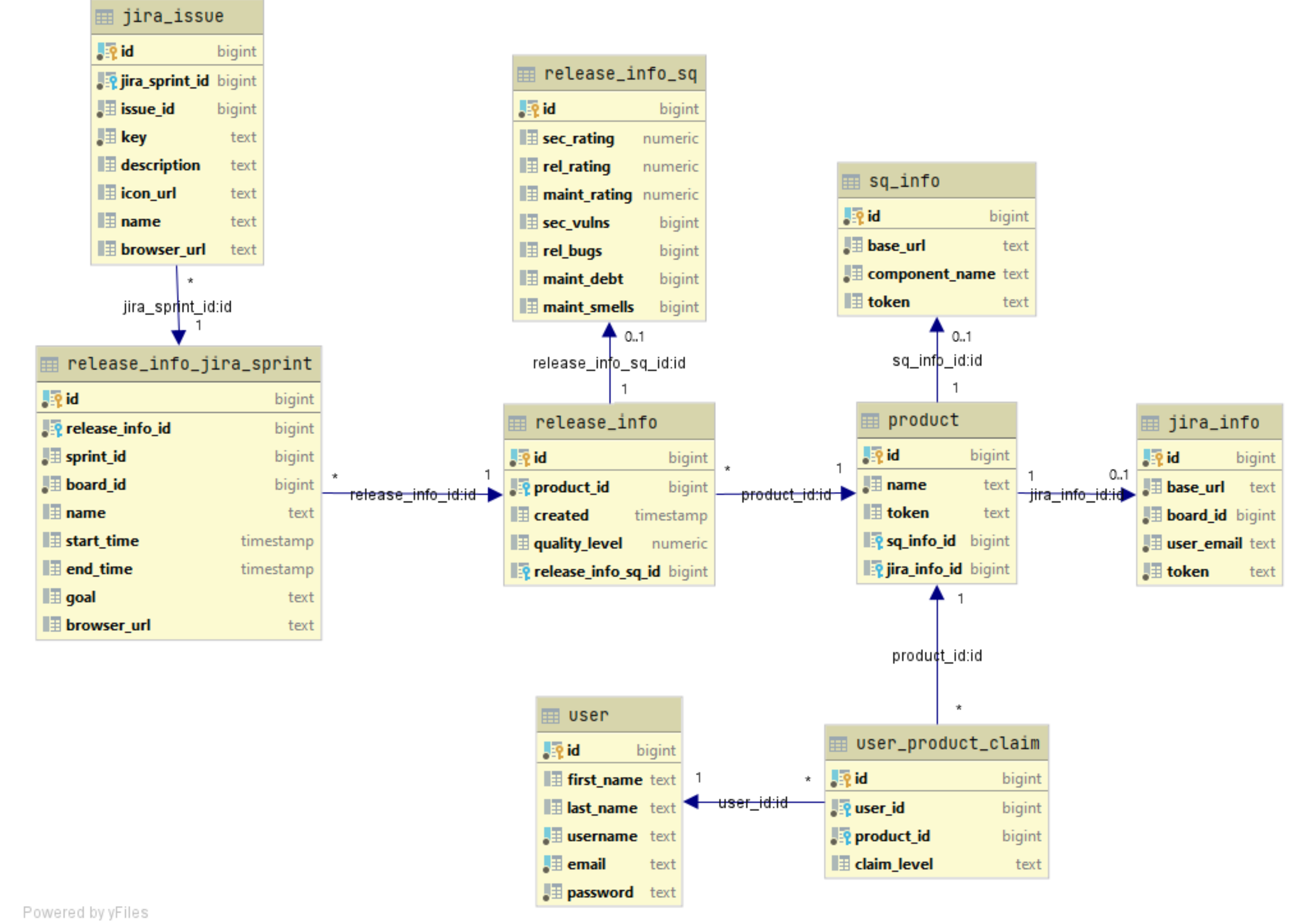

Figure 14. PQD-DB diagram extracted with IntelliJ and multiplicities added

5.4.3 PQD-Front

PQD-Front was written using CoreUI React components library. The solution contains 39 Javascript files with roughly 4500 lines. The front-end was implemented using React Hooks [18] to manage state, and React Context [19] to share global data, such as user and product information. The CoreUI components were taken as a base and modified to fit the needs of the PQD-Front use cases.

Figure 15 shows an example how UI components interact with the React Context in the PQD-Front. PQD-Front is a React application that contains PQD Context. PQD Context contains: User Context, which holds information about the authenticated user; and Product Context, which contains information about the products, including release info. All the UI components are children of the PQD Context. The UI components use React Hooks to interact with the User and Product Contexts.

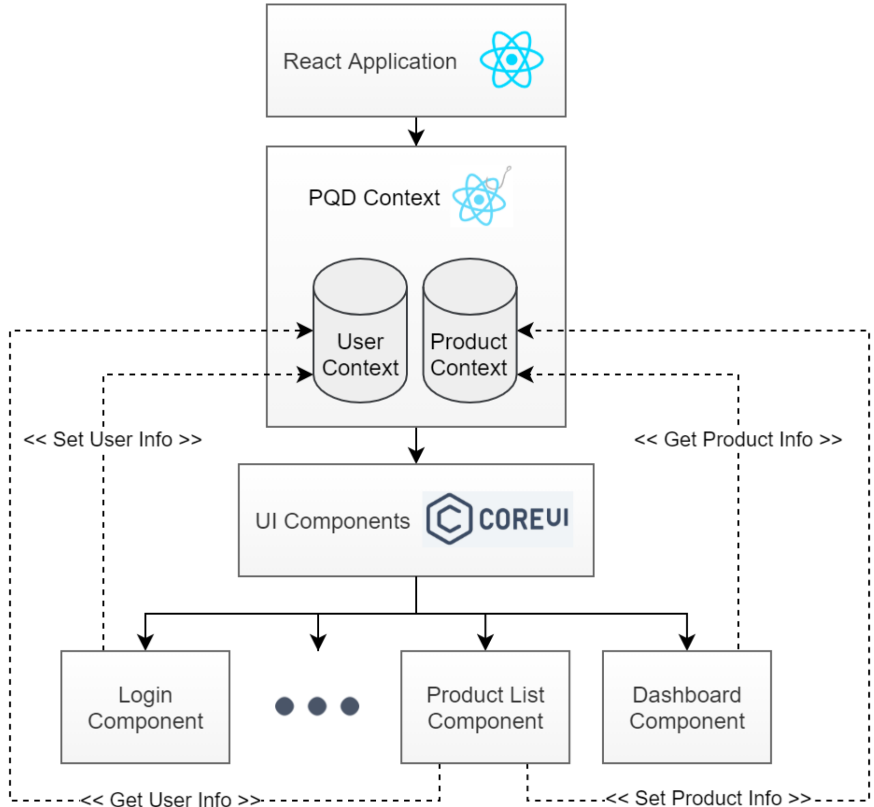

Figure 15. Example of PQD UI components and Context interaction

[18]React Hooks https://reactjs.org/docs/hooks-intro.html
[19]React Context https://reactjs.org/docs/context.html

Figure 16 gives a simplified overview how the PQD-Front retrieves and uses the data from the PQD-API. *GetProducts(jwt)* method from the *product-service* is called when the *ProductList* component is rendered the first time. JWT is included that contains information of the user's products. HTTP Get request is made to the PQD-API from the *product-service*. The PQD-API responds with the list of products authorized for the current user. For each of the products, another request is made to retireve the release info, which is associated with each product. The *product-service* returns the products to the *ProductList* component. The *ProductList* component sets the products into *ProductContext*, switches its own state, and then retrieves the products from the context to display them in HTML.

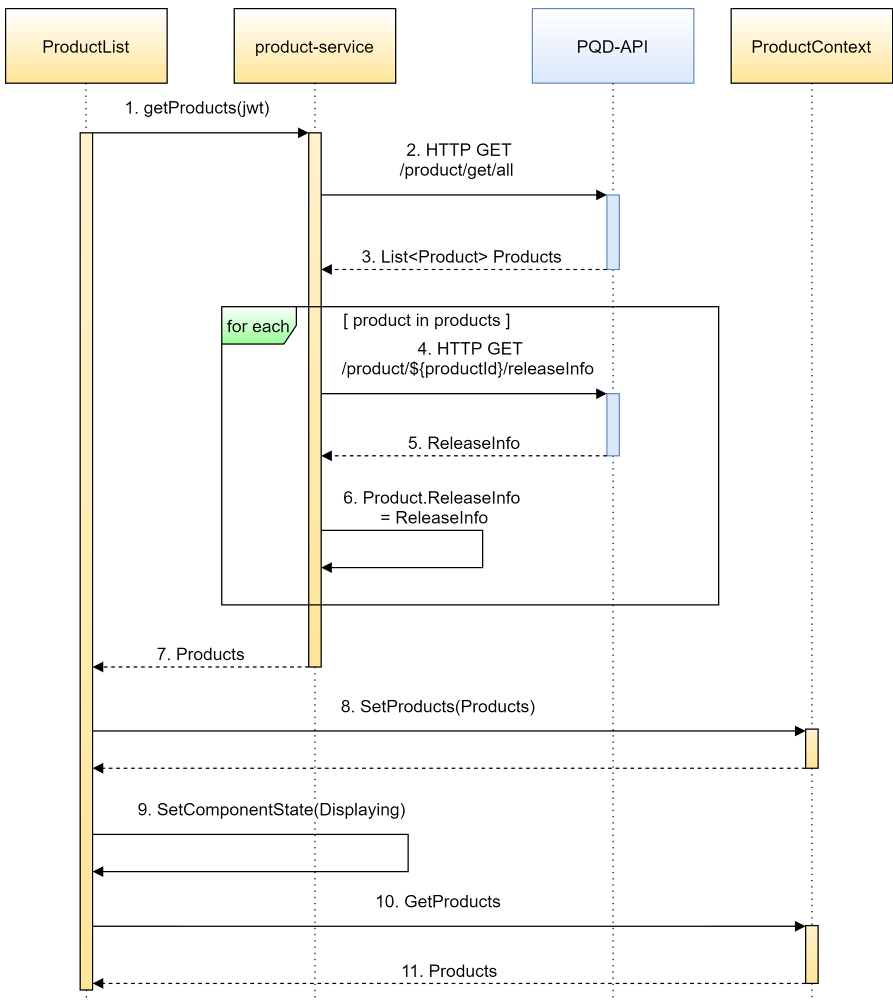

Figure 16. Example interaction of the PQD-Front and the PQD-API

5.5 Test results

Testing was divided into two: unit and integration tests together; and a system test. Unit and integration tests were automated within the PQD-API. System test was done manually on a local machine running the PQD system and SonarQube server. Jira was used from the Atlassian web during the manual system test.

5.5.1 Unit and integration test

The statistics were measured on the 28. January 2021 after the PQD-API developments were finished in the context of this thesis. In total 104 unit tests were written. These tests were divided between the layers as follows:

1. application layer - 27 unit tests
2. adapters layer - 77 unit tests, which are divided by adapters as follows:
   (a) jira - 8 unit tests
   (b) messaging - 6 unit tests
   (c) persistence - 20 unit tests
   (d) sonarqube - 7 unit tests
   (e) web - 36 unit tests

Unit test coverage is measured with JaCoCo within Intellij with default configuration. The report includes application logic files and excludes all the files related to testing. Lombok annotations were also excluded. Class coverage in total is 84.85%. Method coverage in total is 72.82%. Line coverage in total is 79.50%. Note that the coverage results may be different if running on different versions of PQD-API, JaCoCo or IntelliJ. Table 15 shows class, method and line coverage by each module.

Table 15. Unit test code coverage statistics. Created on 28. January 2021.

Module	**Class %**	**Method %**	**Line %**
application	89% (79/88)	76% (206/270)	81% (324/400)
adapters.web	84% (49/58)	67% (186/274)	71% (403/560)
adapters.sonarqube	87% (7/8)	40% (12/30)	71% (53/74)
adapters.persistence	71% (23/32)	85% (117/137)	91% (308/337)
adapters.messaging	33% (1/3)	50% (4/8)	63% (19/30)
adapters.jira	100% (9/9)	70% (35/50)	85% (103/121)
Average	**84.85% (168/198)**	**72.82% (560/769)**	**79.50% (1210/1522)**

In total 35 integration tests were written using MockMvc and Testcontainers. Testcontainers with MockMvc allows to create an isolated environment for the application with database. The integration tests are performed by making http requests on the PQD API endpoints. The tests cover paths that are only dependent on the PQD system with no external dependencies, like SonarQube or Jira server.

5.5.2 System test

System testing was performed manually with the PQD system running on a local machine. The system test was done on 28. February 2021. PQD-Front and database were running in Docker containers. PQD-API was running directly on the local machine. SonarQube server was running directly on the local machine as well. Jira was the only system that was running on the cloud.

Use cases, introduced in section 5.2.2, were taken as an input to validate the system functionality to the user stories, introduced in section 5.2.1. The implemented PQD system satisfies all the elicited user stories. Table 16 shows that all the use cases with priority "must be included" are satisfied (see tables 3, 4, 6, 7, 8, 9, 10). One functionality, regarding use case with priority "Nice to have", is missing from the system and therefore it is not satisfied (see table 5). The system meets with the non-functional requirements described in table 11. Appendix II describes the system test with more details and provides figures showing the key views on the user interface of the implemented system.

Simulated development infrastructure and development process were used during the tests. Application in development, SonarQube, Jira and pipeline were simulated. Java Spring Boot application from Enterprise System Integration (MTAT.03.229) [31] course from spring 2020 was used as an application to simulate development. Sonar Scanner analysis was running on the local code repository and the results were pushed to the local SonarQube server simulating SonarQube analysis on the pipeline. Jira Scrum board was used to simulate actual project data such as sprints, stories, tasks and bugs.

To start with one release cycle, changes to the code were made to simulate development. These changes included injecting bugs and other issues that SonarQube would catch. Sonar Scanner run after the coding was simulated and the results were pushed to the SonarQube server. Some of the results were modified on the SonarQube server to simulate high impact changes on the code. RIC was triggered by http request to the PQD-API messaging endpoint to simulate last step of the running pipeline, which concluded the simulated development to release cycle. New release info was seen at the PQD-Front after these simulated steps.

Table 16. System test results and user story satisfaction

	Priority	Test pass / fail	US1	US2	US3	US4	US5
UC1	Must be included	Pass	Satisfied				
UC2	Must be included	Pass					
UC3	Nice to have	Fail					
UC4	Must be included	Pass				Satisfied	
UC5	Must be included	Pass			Satisfied		
UC6	Must be included	Pass		Satisfied			
UC7	Must be included	Pass					
UC8	Must be included	Pass					Satisfied

5.5.3 Fixes after tests

After the testing was completed, some additional developments were made to the PQD-Front. A bug, that caused missing zero from minute block if the minute block contained minutes under 10, was fixed. For example "25. Jan 2021, 20:6" is displayed as "25. Jan 2021, 20:06" after the fix. Second development was not related to any failure found on testing, but was an improvement. This improvement added tooltips to various places to help the user understand what is meant with different expressions or words in the UI. For example, tooltip, explaining the calculation of quality level, was added next to the place where the quality level was described.

5.6 Evaluation results

The survey was conducted between 17.03.2021 and 16.04.2021. The survey was done using a questionnaire that was distributed by direct emails and newsletters. 88 direct emails were sent along with three newsletters. The direct emails were sent to people who work as POs, perform the activities of POs, or are interested in the product ownership activities. The newsletters were composed and sent out by the University of Tartu Institute of Computer Science.

Seven responses were received in total. Six out of seven responses were in the scope of the evaluation. One response was not considered to be received from the target group of POs. The six responses came from Germany, Argentina, Finland, Estonia, and the USA. Two people responded from Estonia, while the other countries had one respondent.

5.6.1 Background of the POs

The background studies confirmed that six out of seven POs have the responsibilities of a PO and perform the activities of a PO. The job titles of the six POs were different, but the responsibilities were quite similar. The team sizes varied from three to ten persons. The domains included telecommunications, healthcare, finance, facilitation, accountancy, smart city, and sustainability.

The participants were asked whether they keep track of the quality or not, and if they use automated tools for that purpose. Figure 17 shows the answers. Four out of six POs said that they keep track of the quality of their software products. One PO, who had just recently started the role in a new company, said that he/she does not keep track of the quality just yet, but wants to start. Another did not specify the reason. Three out of four POs, who said that they keep track of the quality, said they are using automated tools. One PO, who keeps track of the quality, but does not use tools to do so, told that he/she uses the feedback from the users and stakeholders to keep track of the quality.

Figure 18 shows the tools that the POs use to keep track of the quality. All three POs, who use tools to keep track of the quality, said they use SonarQube. Two POs said that they use IDE-s like IntelliJ or similar. One PO said that he/she uses Codacy. One PO said that he/she uses Plumbr.

Three POs specified what they like about the tools they are using. One PO said that SonarQube, IDE, and Codacy are easy to use. One PO said that he/she likes Plumbrs' automatic Jira ticket generation. One PO said that he/she does not use the tools personally but likes that the team uses them and gives an overview to him/her. No PO specified anything that they do not like about the tools they are using to keep track of the quality.

Two POs specified they keep track of the quality with other measures as well next to the automatic tools. One PO said that he/she keeps track of the quality with measurements related to completed user stories, waste, and technical debt. One PO said that he/she keeps track of the quality by counting the number of bugs per month.

Figure 17. Count of POs who keep track of the quality of their software products

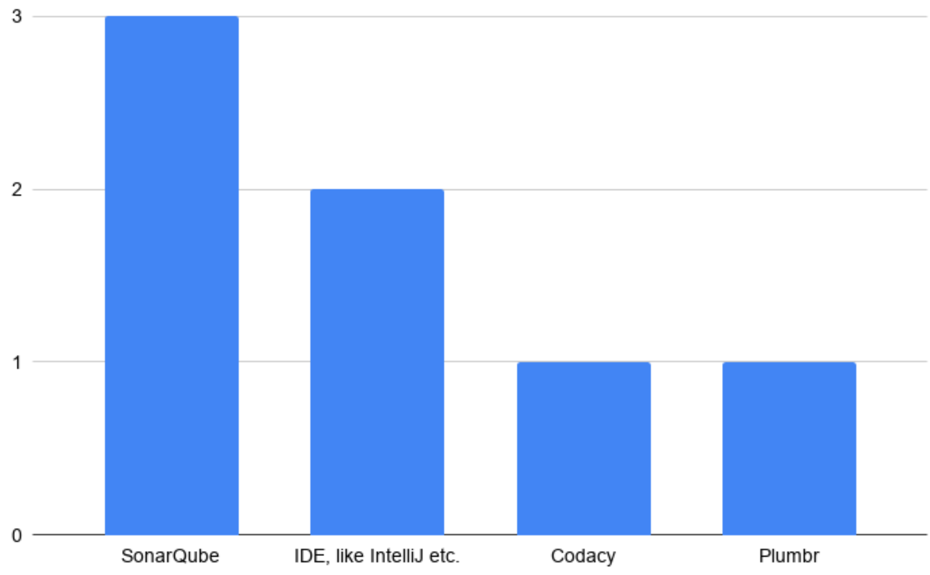

Figure 18. Tools that the POs use to keep track of the quality

5.6.2 POs feedback on the PQD

The technology acceptance model (TAM) was used to determine the potential usage of the PQD in the industry and to see which parts of the PQD need improvement. Ten *likert* scale questions were asked, including five questions regarding *perceived usefulness* (PU) and five questions regarding *perceived ease of use* (PEU). Figure 19 shows the questions and feedback regarding the PU of the PQD. Figure 20 shows the questions and feedback regarding the PEU of the PQD. Based on the answers from POs on the statements regarding PU and PEU, it seems that the PQD is easy to use, but the usefulness is perceived less.

The feedback regarding PU (see figure 19) is rather neutral and leaning on the negative side, indicating that the PQD needs improvements to be a useful tool in the industry. The strong-point of the PU is that the PQD gives a better overview of the POs' products. Weak points of the PU are that too few quality characteristics, and too few project data are considered. The POs opinion remains neutral when it comes to the statements regarding performance improvement and job-related needs addressing.

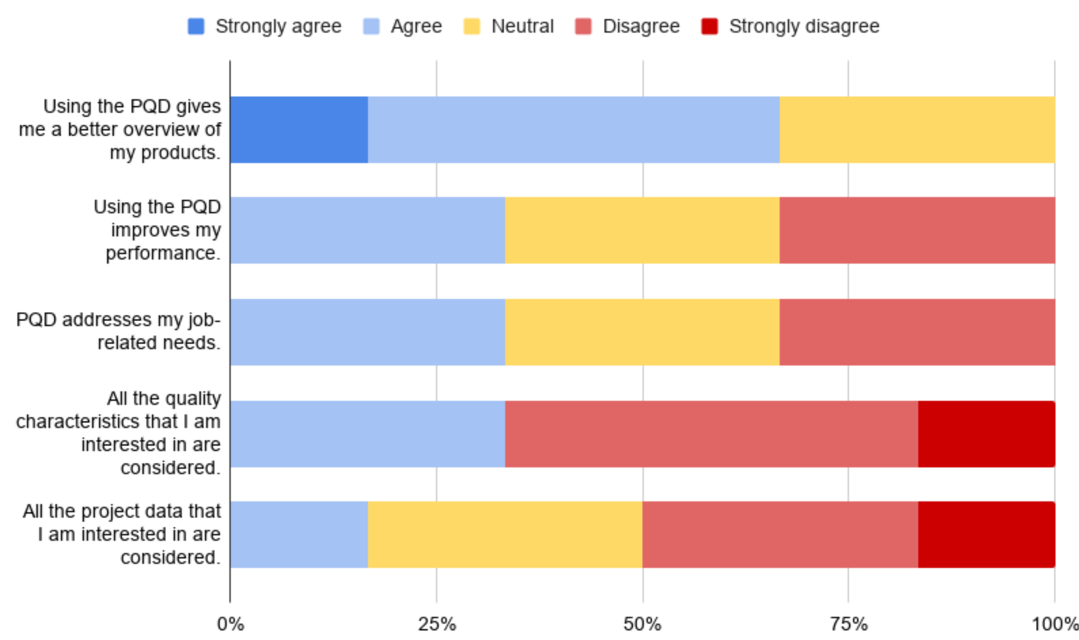

Figure 19. Perceived usefulness of the PQD

Identifying the data that the POs want to see and then implementing it within the PQD can improve the PU. Three of the POs added comments what they would like to see in addition to the current data that is considered by the PQD. One PO said that he/she

wants to see which Jira ticket caused the change in the quality. That PO asked, *"Which jira ticket caused a decline in quality?"* Another PO said that he/she would like to see more details of the quality level. One PO said that he/she would like to track the user feedback with an explanation saying, *"... you can have the best performing product (technical wise), but if the end-user can't understand how to complete their task to move forward because the user experience isn't intuitive then it's doubtful they will use your product ... ".*

The feedback regarding PEU (see figure 20) is mostly positive, which indicates that the PQD is easy to use. Five of the POs agreed that they understand how to use the PQD, while one PO remained neutral on the statement. All of the six POs agreed that it is easy to compare the quality levels of different releases of a product. Five of the POs agreed that it is easy to compare quality levels of different products, while one PO disagreed with the statement. Five of the POs agreed that they understand the quality level of a product on each release, while one PO remained neutral on the statement. The only weak point regarding the PEU is that the composition of the quality level is not understood.

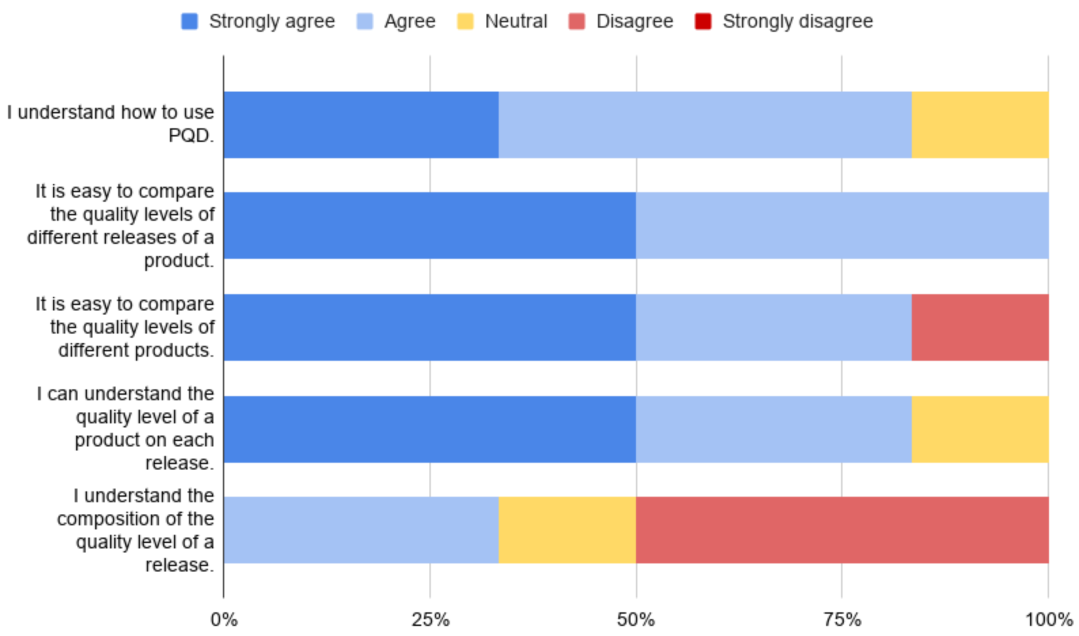

Figure 20. Perceived ease of use of the PQD

The PEU can be improved by increasing the user experience on the PQD-Front. Some of the POs added comments on what they would improve to make the PQD easier to use and understand. One PO said that he/she would like to understand better how the metrics are generated and have a deeper look. Two POs said that they would work on organizing

the content to give a better overview. One PO specified that the quality characteristics were hidden behind two clicks while they could have been shown instantly when the page was opened.

Figure 21 is a composition of the answers regarding if the POs use automatic tools already and if they would use the PQD. The feedback regarding if the POs would use the product or recommend the product remained neutral on average. However, two out of three POs who use the tools, such as SonarQube, to keep track of the quality told that they would use the PQD as well, while one PO remained neutral on the statement. The POs who did not use any tools to keep track of their products' quality told that they would not use the PQD as well. The answers to the statement if the POs would recommend the product were roughly correlated to the answers if the POs would use the PQD by themselves - they would recommend it if they use it by themselves.

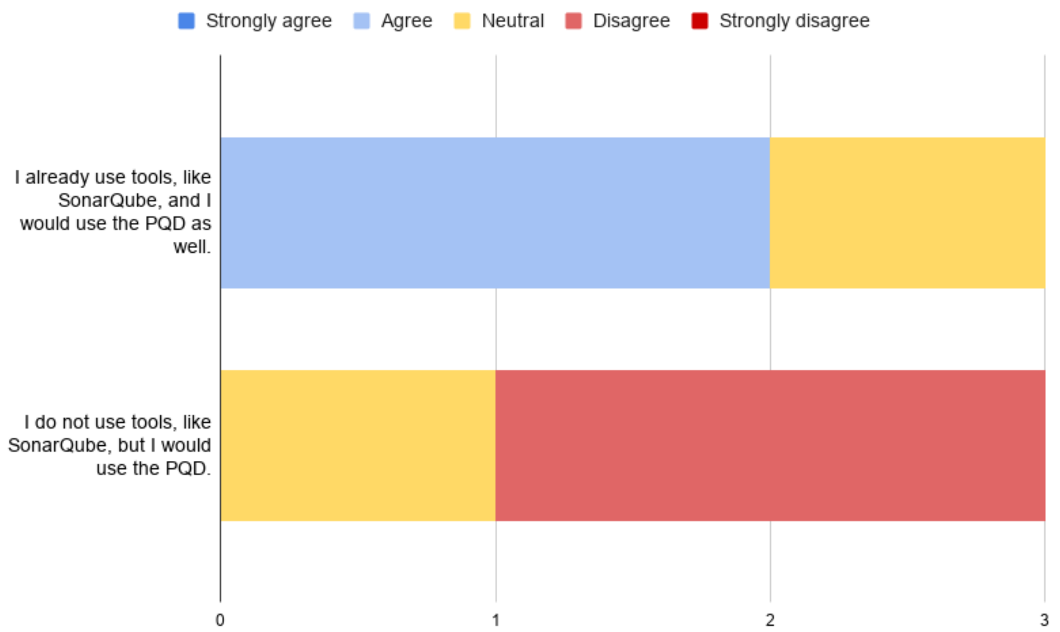

Figure 21. Statements of POs if they would use the PQD

6 Discussion

This chapter describes the limitations, lessons learned from the process, and suggestions for future improvements of the PQD. First, some limitations are discussed. Then, some thoughts are expressed on the process used in the thesis. After that, some suggestions are made for future improvements based on the feedback from the POs and the author's personal opinions.

6.1 Limitation

The validity of the PQD can be questioned because it is not known for sure what the POs' needs are since there is a limited number of studies about it. The validation performed in this thesis was done with few people, using TAM. A larger-scale evaluation should be done to validate the PQD better. This evaluation should be done so that the POs use the PQD with their real projects to see if it addresses their needs. Also, a higher number of participants would give a better model to make assumptions from.

PQD collects parts of data from external tools, such as SonarQube and Jira. The parts of data might not give a complete overview. Users have to look into the external tools to understand the details. This limitation requires that the users can be able to move to specified views of the external tools from the PQD. Also, support to collect information from additional tools should be added to the PQD. Different POs use different tools, so it is necessary that the PQD can connect with more tools.

Another limitation is that the PQD cannot analyze historical data. The PQD must be used over a period of time to visualize the trends in the graphs. It is a difficult task to retrieve historical data from different tools while making associations. Retrieving data in real-time and making associations is more simple and intuitive.

6.2 Lessons learned

One of the biggest problems was to find the target group of people and get them to try out the tool and fill the questionnaire. Larger companies usually have people who can help find the people in the target group. Although the University of Tartu keeps companies' contacts, it is impossible to directly address these companies due to the European General Data Protection Regulation (GDPR). Thus, newsletters were used for distributing the survey. It seems that direct addressing to people works better than the newsletters. The majority of the answers came after sending the direct emails. The response rate was about 8% if to take account that 88 emails were sent and seven responses received. This calculation excludes the newsletters and assumes that all the responses came from the invitations sent by email. It is hard to make sure how many target group people actually received the invitation through the newsletters. When the survey is near the closing

deadline, sending a remainder also seems to work, as seen from performing this survey. It is also important to verify somehow if the response is from the target group.

It is also good to test out the whole survey before sending it out. This includes keeping the system running on the cloud for regular checks. If the system should be unavailable for some reason when somebody wants to try it, then it is a response lost. It was seen while testing the survey in this thesis that the free-tier version of the cloud instance froze at some random point in the 48h since the start. It was probably due to a lack of resources because more capable instance worked fine during the testing and the survey conduction period.

Another quite obvious lesson that was learned after analyzing the results of the evaluation part was that the people who do not use some specific tools now, probably will not use a similar new tool too. The important thing here is to find the people who are using similar things already and then provide them with a better solution. It is harder to convince people that they should use the new solution when they do not even use the existing solutions.

One lesson learned is a piece of common knowledge in the management of software development. It is to keep a buffer time for solving unexpected issues. During the development of the PQD application, few such unexpected issues emerged. For example, one of the testing dependencies used a Docker container that it downloaded itself, and the container disappeared from the URL. The solution was simple, but finding the problem took time.

It is important to point out that small value-adding iterations can be effectively used to write a thesis. This approach was used during the writing of this thesis as well. The high level plan was set at the beginning. After that, small increments were made to the work and these increments were regularly discussed every second week. This helped to keep a common understanding of the work, detect wrong paths early on, and keep the work going continuously.

6.3 Future work

The future improvements to the PQD should be done in an incremental and iterative way. This means that the tool should be re-evaluated after some changes to collect new insight from the target group for next improvement and so on.

The overall ease of use of the PQD is quite good based on the evaluation results seen in figure 20. The majority of the participants agree that the PQD is easy to use. However, the composition of quality level is not easy to understand. This could be improved by giving a visual explanation rather than just a text inside a tooltip. Also, the descriptive texts could be organized in a better way. One of the participants said that he/she would have wanted that the quality characteristics of a selected release would have been visible right away. Currently, it requires that the user makes additional clicks to display the quality characteristics of a release when they first come to the detailed view.

The quality level calculation could be configurable for each product. Currently, it is hardcoded that each characteristic has the same weight in the overall quality level. The calculation could be made individual for each product with a rule that describes how the calculation should be done. The rule should be configurable by the user on the UI, and it should be saved into the database, where it is associated with the product.

The usefulness of the PQD can be improved based on the evaluation results seen in figure 19. Some of the POs want to be able to have a deeper look into the details regarding the quality. It can be done by giving the link to SonarQube analysis result like it is already done with the Jira tickets. It is also useful to extract more information from SonarQube and to extend the PQD to connect with more ASATs next to SonarQube. Jira tickets could also be linked to a release. Currently, the tickets are from an active sprint, but this is a bit inaccurate association with the release info because there can be multiple releases in one sprint. Jira tickets have fields that can be connected with a release, and that information could be highlighted on the PQD user interface.

While investigating the feedback of the POs, it was seen that two of them referred to a similar concept. They implied that they would like to have a new concept of data collection in the PQD. This concept would provide continuous updates on the last release. This additional new functionality could be called PQD continuous release info collection (CRIC). Currently, the PQD fetches the information once the release is done, and then it waits idly for the next release. Events that happen between the releases will not affect the existing release info in the PQD. The current way is called release info collection (RIC).

The business value of the CRIC is based on the following: the POs wanted to have the user feedback and the monitoring data linked with the currently active release. This kind of data is created continuously when the software is up and running. The user feedback can give information regarding functionality and usability. The monitoring systems could give information about functionality, reliability, and performance efficiency. For example, Plumbr detects errors that happen during the use of the software that are linked to user activities.

The CRIC could be implemented in different ways. One way, which is the lesser option, is to have the systems send the data directly to the PQD. Another way, a more correct way, is to have a trigger that would start so-called *release info recollection* that would collect additional information on top of the existing release info. This way, there is less effort for the users to set up the PQD and the CRIC. Also, it would be consistent with the current system architecture that the PQD collects the data by itself and does not need to have the data sent to it. It might be useful in some contexts to open an endpoint on the PQD that would allow some data to be sent on the last release info. For example, when the external tool does not provide API, but it can send out data.

On a more technical side, the PQD could be updated to use messaging queues to control the RIC and CRIC. There are benefits of the messaging queues on the systems like improved performance and reliability. RabbitMQ or Kafka could be used for this.

7 Conclusion

A working open-source system has been delivered in this thesis. The system is called Product Quality Dashboard (PQD). The PQD is meant for the Product Owners (POs) to provide a straightforward overview of the quality level on each release. The current version of the PQD collects pieces of information from SonarQube and Jira, visualizes the information, and calculates a quality level for each release. The PQD is built using Hexagonal architecture for easier future extensions. The PQD is meant to be used as an automatic tool next to a release pipeline.

The PQD was designed, implemented, tested, and evaluated on the POs during this thesis. The design involved creating the high-level idea and requirements. The implementation was about building the system based on the requirements. The testing involved unit, integration, and system tests in confirming the functional correctness of the system. The automated unit and integration tests ensured that the components of the system worked as specified. The system test was performed manually to validate that the system satisfied the user stories and met the requirements, including non-functional requirements. The evaluation gave insight into the target group's perceived usefulness and ease of use about the product at the given state. The results from the evaluation showed that the system is easy to use, and some feel it useful, while others think that more functionalities are needed.

The future work primarily involves expanding the system to support connection with more tools. The PQD architecture supports expanding the application that is needed to connect the PQD with more external tools. More about the future work can be read from section 6.3.

References

[1] Mariana Falco, Ezequiel Scott, Gabriela Robiolo, “Overview of an automated framework to measure and track the quality level of a product,” in *2020 IEEE Biennial Congress of Argentina (ARGENCON)*, 2020 [in press].

[2] Ori Bendet, “5 tips for scaling agile in an enterprise environment,” 2020. https://techbeacon.com/app-dev-testing/5-tips-scaling-agile-enterprise-environment. Accessed on 10.10.2020.

[3] Gilad Maayan, “What software quality (really) is and the metrics you can use to measure it,” 2020. https://www.altexsoft.com/blog/engineering/what-software-quality-really-is-and-the-metrics-you-can-use-to-measure-it/. Accessed on 10.10.2020.

[4] Nicole Forsgren, “2015 state of devops report,” July 2015.

[5] A. Merey and H. Sneed, “Automated software quality assurance,” *IEEE Transactions on Software Engineering*, vol. 11, pp. 909–916, September 1985.

[6] Hrafnhildur Sverrisdottir, Helgi Ingason, Haukur Jonasson, “The role of the product owner in scrum-comparison between theory and practices,” *Procedia - Social and Behavioral Sciences*, vol. 119, pp. 257–267, March 2014.

[7] Michał Choraś, Rafał Kozik, Damian Puchalski and Rafał Renk, “Increasing product owners’ cognition and decision-making capabilities by data analysis approach,” *Cognition, Technology Work*, vol. 21, pp. 191–200, June 2018.

[8] iso25000.com, “Iso/iec 25010,” 2020. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010. Accessed on 04.10.2020.

[9] Mariana Falco and Gabriela Robiolo, *A Unique Value that Synthesizes the Quality Level of a Product Architecture: Outcome of a Quality Attributes Requirements Evaluation Method*, pp. 649–660. 11 2019.

[10] V. Nanda, *Quality Management System Handbook for Product Development Companies*. Taylor & Francis, 2005.

[11] Richard Mark Soley, Bill Curtis, “The Consortium for IT Software Quality (CISQ),” in *Software Quality. Increasing Value in Software and Systems Development* (Dietmar Winkler, Stefan Biffl, Johannes Bergsmann, ed.), (Berlin, Heidelberg), pp. 3–9, Springer Berlin Heidelberg, 2013.

[12] Consortium for Information & Software Quality, "Measuring code quality," 2020. `https://www.it-cisq.org/standards/code-quality-standards/`. Accessed on 10.10.2020.

[13] Digital.ai, "14th annual state of agile report," May 2020.

[14] Ken Schwaber, Jeff Sutherland, "The scrum guide," November 2020.

[15] John Noll, Mohammad Abdur Razzak, Julian M. Bass, Sarah Beecham, "A Study of the Scrum Masterś Role," in *Product-Focused Software Process Improvement* (Michael Felderer, Daniel Mendez Fernandez, Burak Tuhan, Marcos Kalinowski, Federica Sarro, Dietmar Winkler, ed.), (Cham), pp. 307–323, Springer International Publishing, 2017.

[16] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, Harald C. Gall, "Context is king: The developer perspective on the usage of static analysis tools," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 38–49, March 2018.

[17] Stanislav Mõškovski, *Building a tool for detecting code smells in Android application code*. Master's thesis, University of Tartu, 2020.

[18] Hammad Khalid, Meiyappan Nagappan, Ahmed E. Hassan, "Examining the relationship between findbugs warnings and app ratings," *IEEE Software*, vol. 33, no. 4, pp. 34–39, 2016.

[19] Alexander Trautsch, Steffen Herbold, Jens Grabowski, "A longitudinal study of static analysis warning evolution and the effects of pmd on software quality in apache open source projects," *Empirical Software Engineering*, vol. 25, pp. 5137–5192, 2020.

[20] Diego Marcilio, Rodrigo Bonifacio, Eduardo Monteiro, Edna Canedo, Welder Luz, Gustavo Pinto, "Are static analysis violations really fixed? a closer look at realistic usage of sonarqube," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pp. 209–219, 2019.

[21] Mika Ohtsuki, Kazuki Ohta, Tetsuro Kakeshita, "Software engineer education support system alecss utilizing devops tools," in *Proceedings of the 18th International Conference on Information Integration and Web-Based Applications and Services*, iiWAS '16, (New York, NY, USA), p. 209–213, Association for Computing Machinery, 2016.

[22] Kristín Fjóla Tómasdóttir, Aniche Maurício, Arie Van Deursen, "The adoption of javascript linters in practice: A case study on eslint," *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 863–891, 2020.

[23] SonarSource S.A, “Sonarqube,” 2020. https://www.sonarqube.org/. Accessed on 10.10.2020.

[24] Vijay Vaishnavi, Bill Kuechler, Stacie Petter, and Gerard De Leoz, “Design science research in information systems,” 07 2019.

[25] The PostgreSQL Global Development Group, “Postgresql: The world’s most advanced open source relational database,” 2021. https://www.postgresql.org/. Accessed on 13.04.2021.

[26] Facebook Inc., “React - a javascript library for building user interfaces,” 2021. https://reactjs.org/. Accessed on 13.04.2021.

[27] CoreUI, “Coreui react components library,” 2021. https://github.com/coreui/coreui-react. Accessed on 12.03.2021.

[28] Richard North, Sergei Egorov, and others, “Testcontainers,” 2021. https://www.testcontainers.org/. Accessed on 21.03.2021.

[29] Nikola Marangunic, Andrina Granic, “Technology acceptance model: a literature review from 1986 to 2013,” *Universal Access in the Information Society*, 03 2015.

[30] Len Bass, Paul Clements, Rick Kazman, *Software Architecture in Practice*. SEI Series in Software Engineering, Pearson Education, 2003.

[31] Orlenys López Pintado, Luciano García-Bañuelos, Marlon Dumas, “Enterprise system integration,” 2020. https://courses.cs.ut.ee/2020/esi/spring. Accessed on 01.03.2021.

[32] Sandeep Singh Shekhawat, “Onion architecture in asp.net core mvc,” 2020. https://www.c-sharpcorner.com/article/onion-architecture-in-asp-net-core-mvc/. Accessed on 31.03.2021.

[33] Bart De Win, Frank Piessens, Wouter Joosen, Tine Verhanneman, “On the importance of the separation-of-concerns principle in secure software engineering,” in *Workshop on the Application of Engineering Principles to System Security Design*, pp. 1–10, Citeseer, 2002.

[34] Ben Morris, “The problem with tiered or layered architecture,” 2014. https://www.ben-morris.com/the-problem-with-tiered-or-layered-architecture/. Accessed on 31.03.2021.

[35] Robert Cecil Martin, *Clean Architecture: A Craftsman’s Guide to Software Structure and Design*. Prentice Hall, 2018.

[36] M. Ehsan Khalil, Kamran Ghani, Wajeeha Khalil, "Onion architecture: a new approach for xaas (every-thing-as-a service) based virtual collaborations," in *2016 13th Learning and Technology Conference (L T)*, pp. 1–7, 2016.

[37] Alistair Cockburn, "Hexagonal architecture." https://alistair.cockburn.us/hexagonal-architecture/. Accessed on 05.03.2021.

[38] choquero70, "Onion architecture compared to hexagonal," 2018. https://stackoverflow.com/questions/50039019/onion-architecture-compared-to-hexagonal/50051803#50051803. Accessed on 02.04.2021.

Appendix

I. Repositories

The PQD is available in two different repositories:

1. PQD-BE repository - the repository for the PQD-API, an API that holds the business logic and interacts with the PQD-DB. The repository holds the migration files for the database as well. The repository can be accessed from: `https://github.com/Kert944/PQD-BE`. Last accessed 24th April 2021.

2. PQD-Front-React repository - the repository that holds the PQD-Front, a user interface of the PQD system that interacts with the PQD-API. The repository can be accessed from: `https://github.com/Kert944/PQD-Front-React`. Last accessed 24th April 2021.

II. System test report

This appendix describes the system test performed on 28. February 2021. Note that, the system received additional developments and fixes after the tests. The tests were performed on a local machine with database and PQD-Front running in Docker containers. PQD-API and SonarQube were running directly on the local testing machine. Jira was only application that was running on the web. The local machine was running on Windows OS, PQD system version was 26.02.2021, browser was Chrome version 88.0.4324.190. PQD-API was running on port 8080, PQD-Front was running on port 3000, database was running on port 5432, SonarQube server was running on port 9000, Jira was running on the web.

Table 17 describes the steps performed to verify the functionality, described in use case 1 (see table 3), was present in the system. The test regarding use case 1 passed and therefore user story 1 (see table 2) was satisfied. Figures 22, 23 and 24 show the key views of the use case during the test.

Table 18 describes the steps performed to verify the functionality, described in use case 2 (see table 4), was present in the system. The test regarding use case 2 passed with one minor, low impact, missing functionality. Alternative flow 1, in table 18, specified that failure notice should be shown if the saving had failed, but instead no message was shown and the user stayed at the same view. This is a low-impact flaw, which can happen if there are issues with network connection. The validators would not allow faulty request to be made - save button would be disabled. Figures 26, 27, 29 and 30 show the key views of use case 2.

Functionality described in use case 3 (see table 5 was missing from the system. Use case 3 had a priority of "nice to have". Therefore it did not impact the overall system test result.

Table 19 describes the steps performed to verify the functionalities, described in use cases 4, 5, 6, 7 and 8 (see tables 6, 7, 8, 9 and 10), were present in the system. The tests regarding use cases 4, 5, 6, 7 and 8 passed. Use cases 4, 5, 6 and 8 satisfied user stories 2, 3, 4 and 5 (see table 2). One low impact fault was found regarding release time minute section - release times are missing “0” from minutes if the minutes block contains minutes under 10. For example “25. Jan 2021, 20:6” should be “25. Jan 2021, 20:06”. Figures 31, 32, 33 and 34 show the key views of use cases 4, 5, 6, 7 and 8. Figure 31 shows a view that satisfies use cases 4, 6 and 8. Figure 33 shows a view that satisfies use case 7. Figures 33 and 34 show views that satisfies use case 5.

Table 20 describes the steps performed to verify that the system meets with the non-functional requirements specified in table 11. All the non-functional requirements were satisfied.

Table 17. System test case steps of use case 1.

STEP	TEST STEP / INPUT	EXPECTED RESULTS	ACTUAL RESULTS	PASS / FAIL
View list of PQD products				
1.	User logs in	Redirect to product list view	Redirect to product list view	PASS
2.	Wait for products to be loaded	Loader is shown	Loader was shown	PASS
3.	View list of products	See list of PQD products	Products were shown (see figure 22)	PASS
Alternative Flow 1: Products does not exist				
4.	Step 3: Products does not exist	Notice saying no products exist	Notice shown (see figure 23)	PASS
Alternative Flow 2: Loading Failed				
5.	Step 3: Fetching products failed	Display error notice	Error notice was shown (see figure 24)	PASS

Table 18. System test case steps of use case 2.

STEP	TEST STEP / INPUT	EXPECTED RESULTS	ACTUAL RESULTS	PASS / FAIL
Add PQD product				
1.	Click on “Add Product” at the product list view	Adding new product modal opened	Modal opened (see figure 26)	PASS
2.	Verify it is understood how to add a product	Input fields understood and external links provided for SonarQube and Jira documentations	Placeholders, instructions and links present	PASS
3.	Insert data	Invalid and valid input is indicated	Invalid data is shown in red, valid data is shown in green (see figure 27)	PASS
4.	Test tool connection	Success and failure notice accordingly to connection test result	Success notice shown if connection was successful (see figure 27); Error notice shown if connection was not successful (see figure 28)	PASS
5.	Click on “save”	Loader is shown	Loader was shown	PASS
6.	See success information	Success notice and information about the product is shown	Notice and information shown (see figures 29 and 30)	PASS
7.	Product is saved	Product is saved and can be accessed from the product list	User has to re-login and the product can be seen from the product list	PASS
Alternative Flow 1: Saving failed				
8.	Step 6: see failure notice	Failure notice shown	No failure notice, user stays at the same view and can press “save” button again	FAIL

Table 19. System test case steps of use case 4, 5, 6, 7 and 8.

STEP	TEST STEP / INPUT	EXPECTED RESULTS	ACTUAL RESULTS	PASS / FAIL
Detailed view of PQD product				
1.	Click on a product at the product list view	Redirected to product details view	Redirected to product view	PASS (UC4)
2.	View history of quality levels on a graph	Graph displayed at the top of the page	Graph displayed (see figure 31)	PASS (UC8)
3.	Select release from history of releases	Dropdown showing releases; Can select release from dropdown	Dropdown exists; Can select release and information of the release displayed	PASS (UC5)
4.	View overall quality level	Quality level displayed	Quality level displayed (see figures 31, 33 and 34)	PASS (UC6)
5.	Select "Sonarqube" from "Inspect details from" block	Details about Security, Reliability and Maintainability displayed	Details displayed (see figure 33)	PASS (UC6, UC7)
6.	Select "Jira" from "Inspect details from" block	Information about Jira sprint active at the release time displayed; Issues in sprint displayed	Active Jira sprint at the release time displayed; External link to open Jira exists; Issues in sprint displayed (see figure 34)	PASS
Alternative Flow 1: Release info does not exist				
7.	Step 2: view notice of no data exists	Notice shown	Notice shown (see figure 32)	PASS
Alternative Flow 2: Sonarqube not added to the PQD product				
8.	Step 5: Select "Sonarqube"	Sonarqube cannot be selected	Sonarqube button missing	PASS
Alternative Flow 3: Jira not added to the PQD product				
9.	Step 6: Select "Jira"	Jira cannot be selected	Jira button missing	PASS

Table 20. System test case steps of non-functional requirements

STEP	TEST STEP / INPUT	EXPECTED RESULTS	ACTUAL RESULTS	PASS / FAIL
NFR-1: Desktop and mobile view				
1.	Use application in desktop view	The layout fits the desktop	The layout fits the desktop (see figure 22	PASS
2.	Use application in mobile view	The layout fits the mobile screen	The layout fits the mobile screen (see figure 35)	PASS
NFR-2: User authentication and authorization				
3.	Make HTTP request with invalid JWT	Unauthorized message received	Unauthorized message received	PASS
4.	Ask product and release info with valid JWT, but for unauthorized products	Unauthorized message received	Unauthorized message received	PASS
5.	Ask product and release info with valid JWT, and for authorized products	Ok response received	Ok response received	PASS
NFR-3: System responsiveness				
6.	Try out the entire application and look for freezes over one second	The application responds to user actions within 1 sec	The application responds to actions under 1 sec or shows loader (see figure 25	PASS
NFR-4: Architecture and documentation				
7.	Check the repositories for documentation	Readme exists with information about the application	Readme exists and contains information about the application	PASS
8.	Check for architecture that is scalable and maintainable	Architecture is specified	Hexagonal architecture specified	PASS

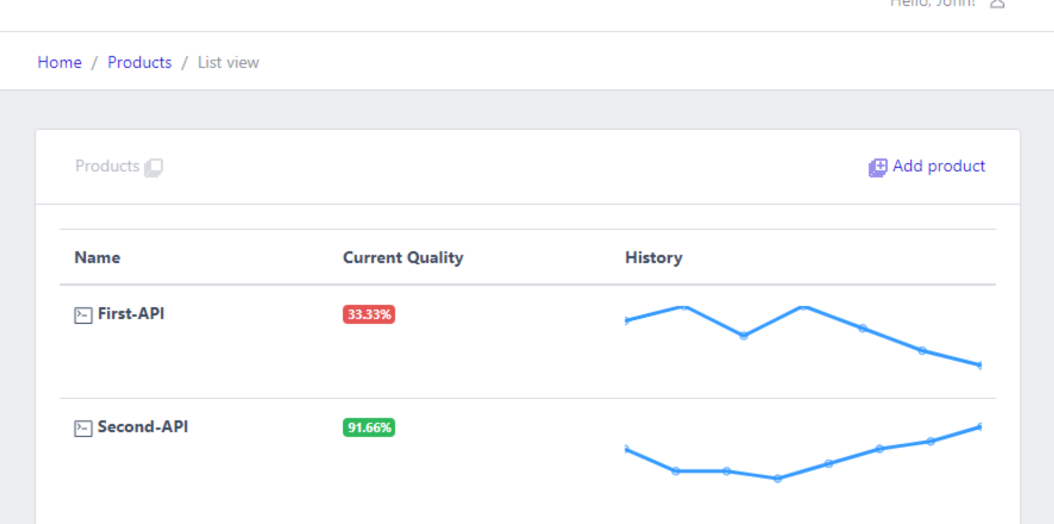

Figure 22. List of products

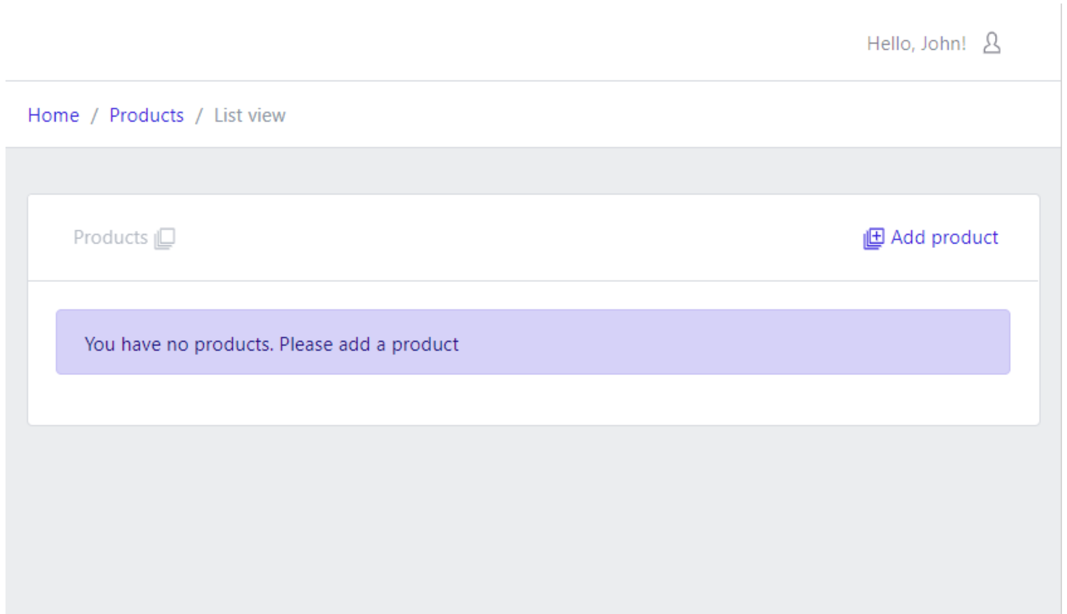

Figure 23. List of products with no products to display

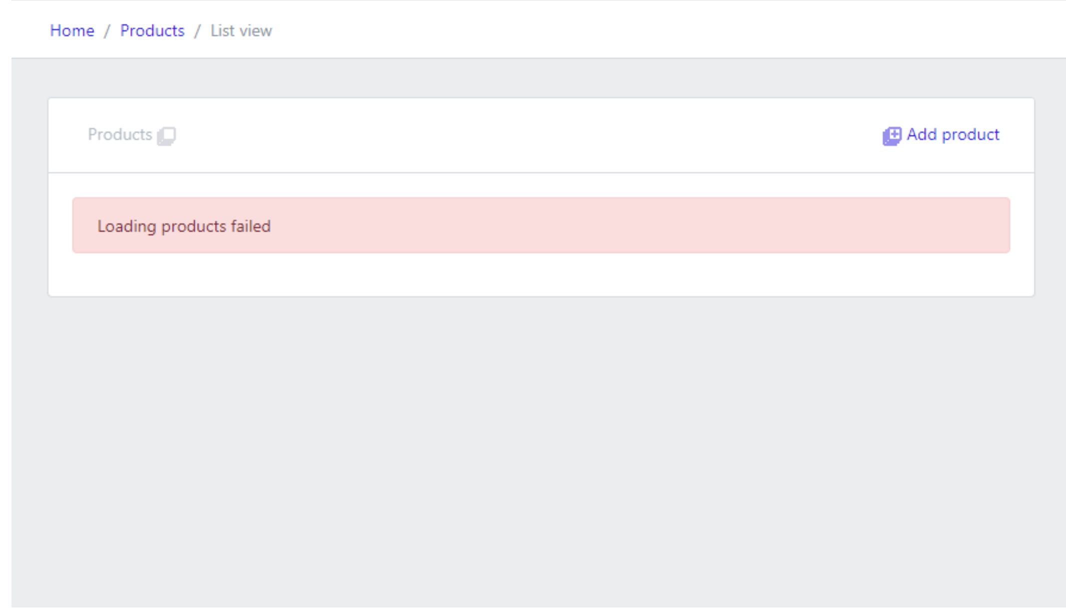

Figure 24. List of products with loading failed

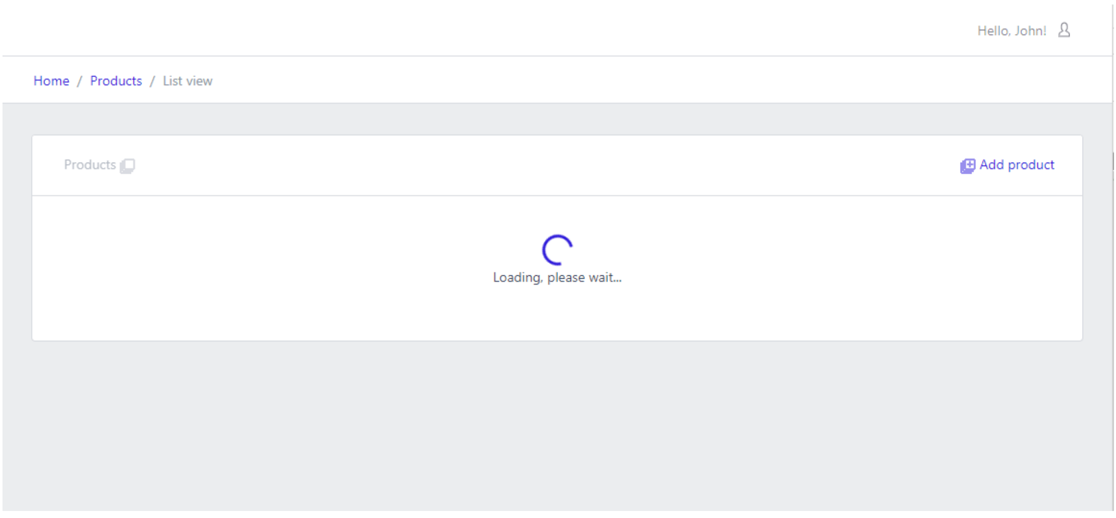

Figure 25. List of products during the loading

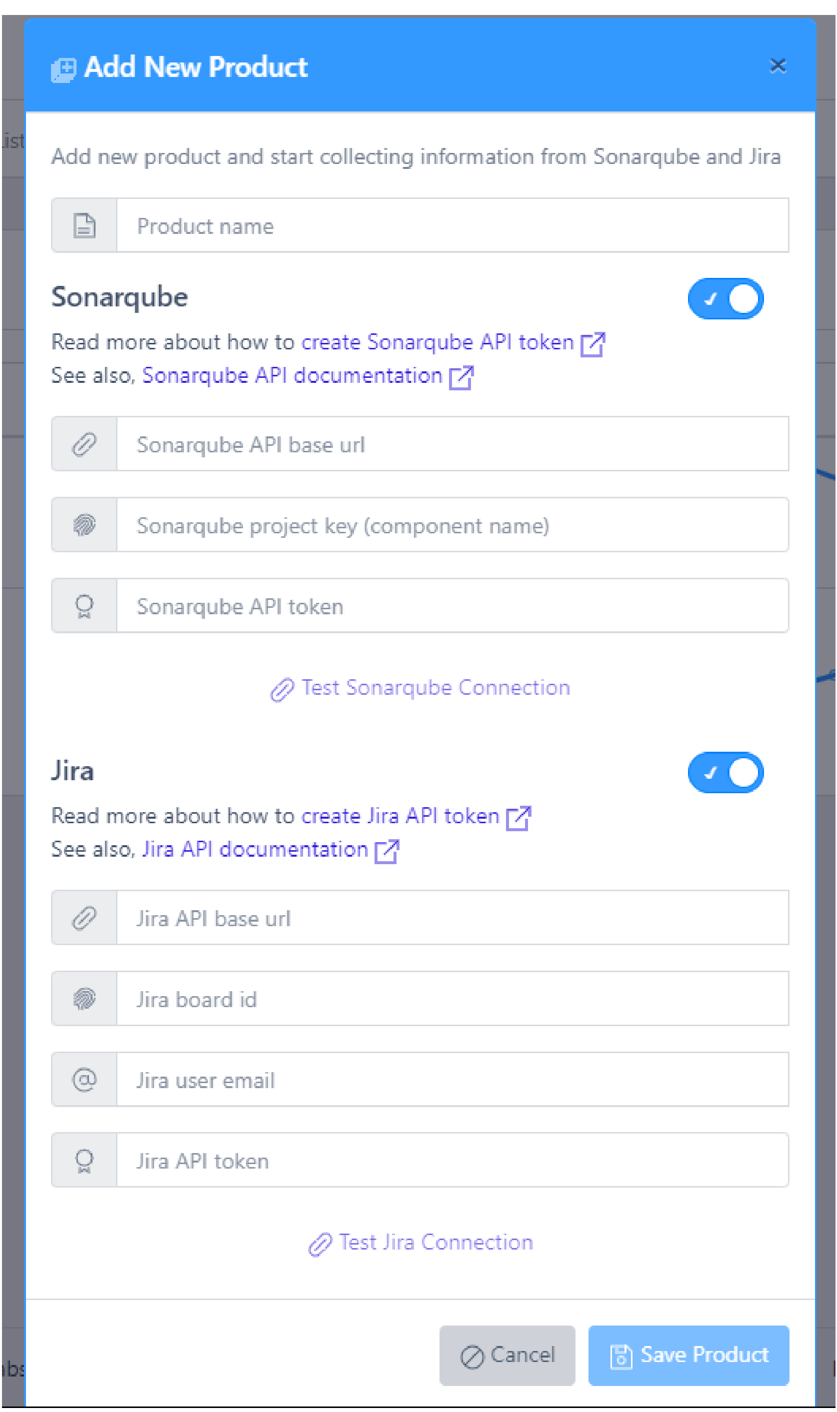

Figure 26. Product adding modal

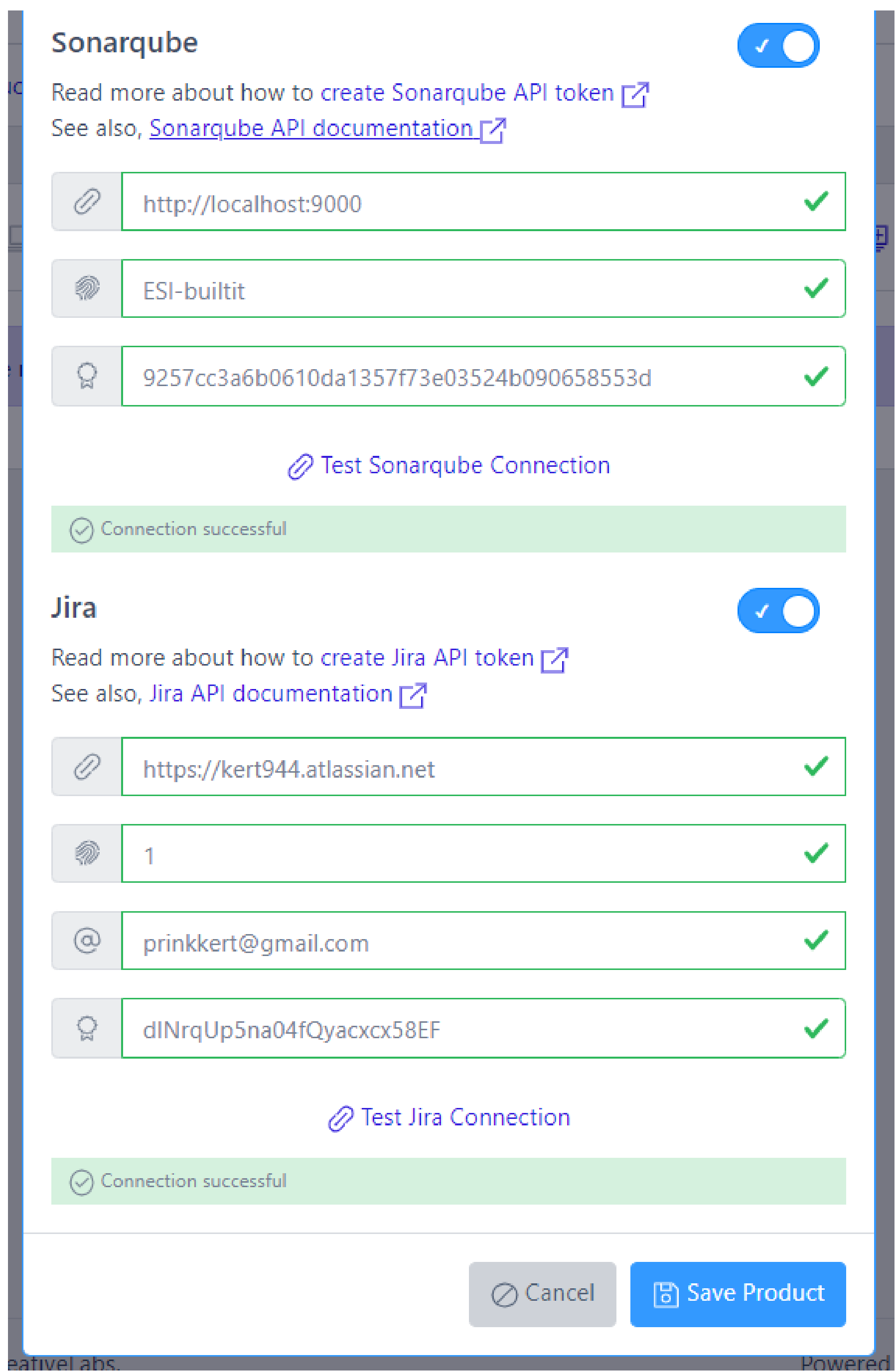

Figure 27. Valid input fields and successful connection tests

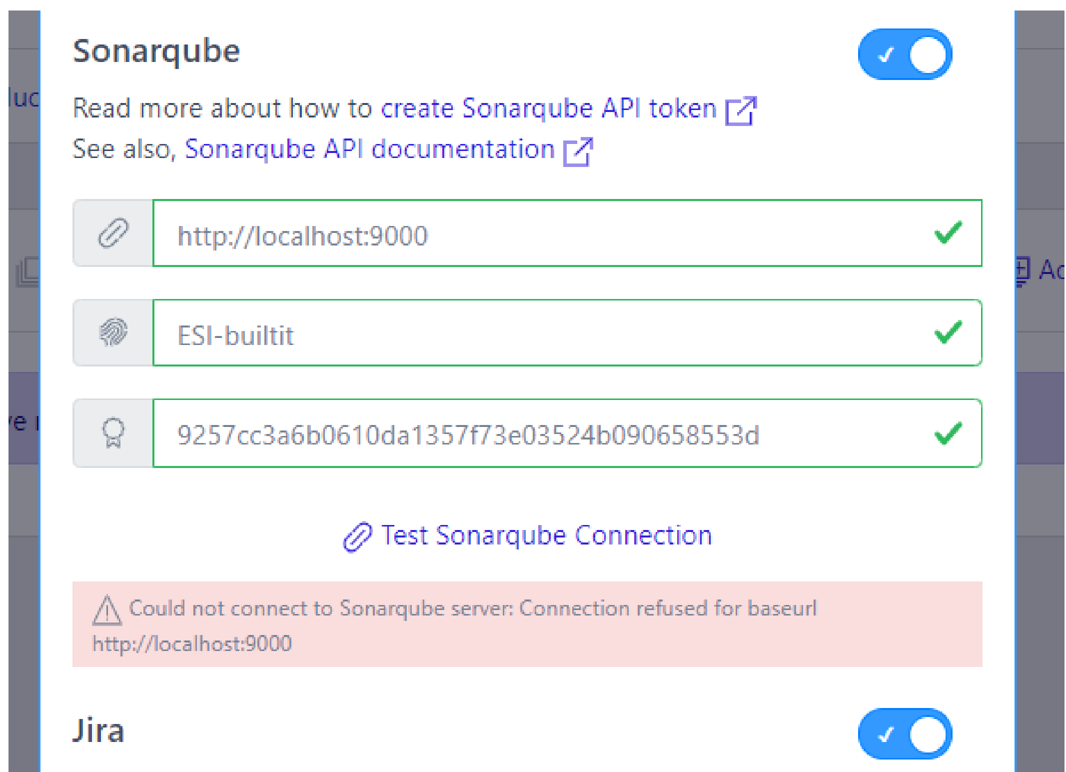

Figure 28. Failed connection test

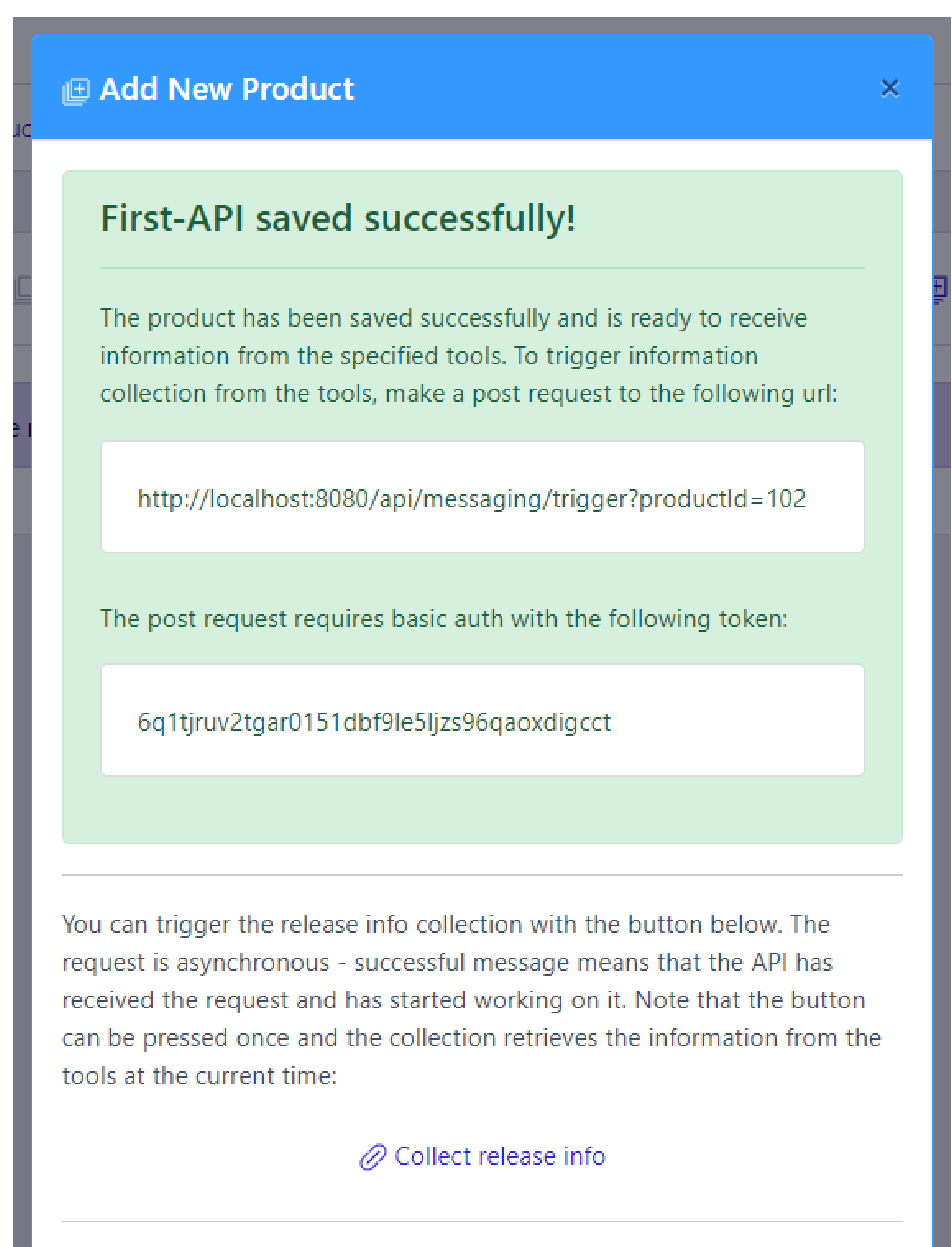

Figure 29. Product saving success message

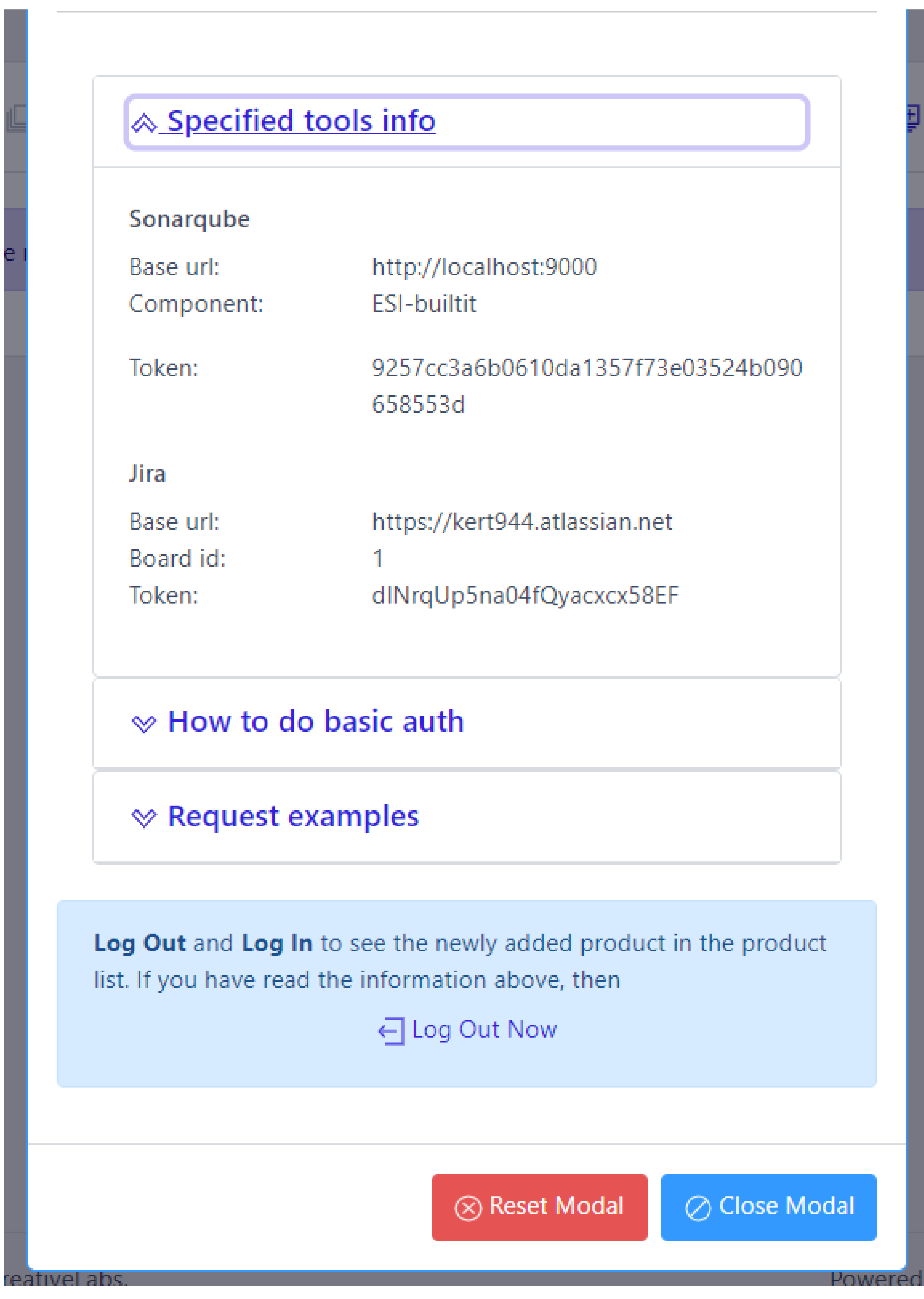

Figure 30. Information about saved product

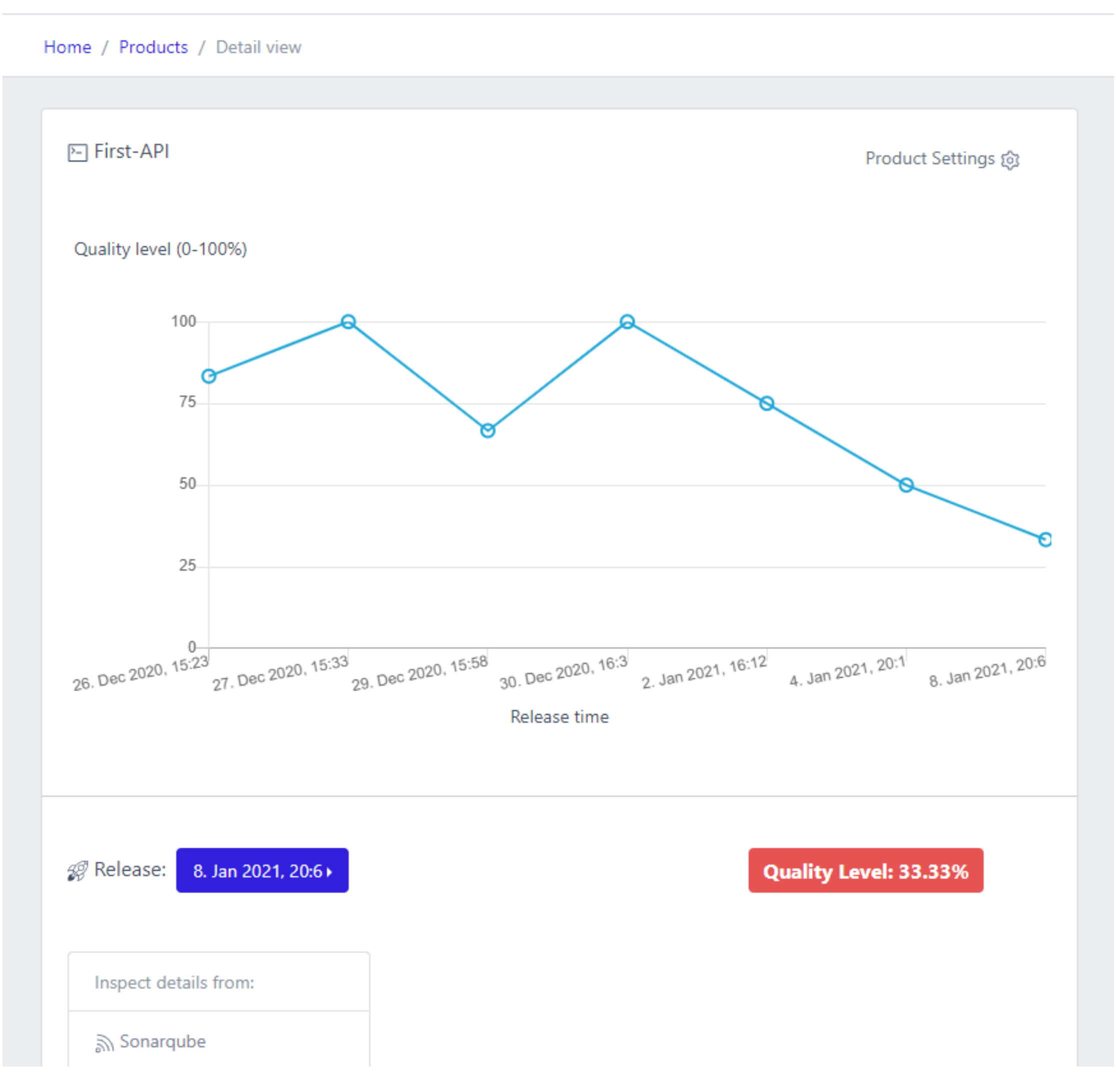

Figure 31. Graph of quality levels at the detailed view of a product

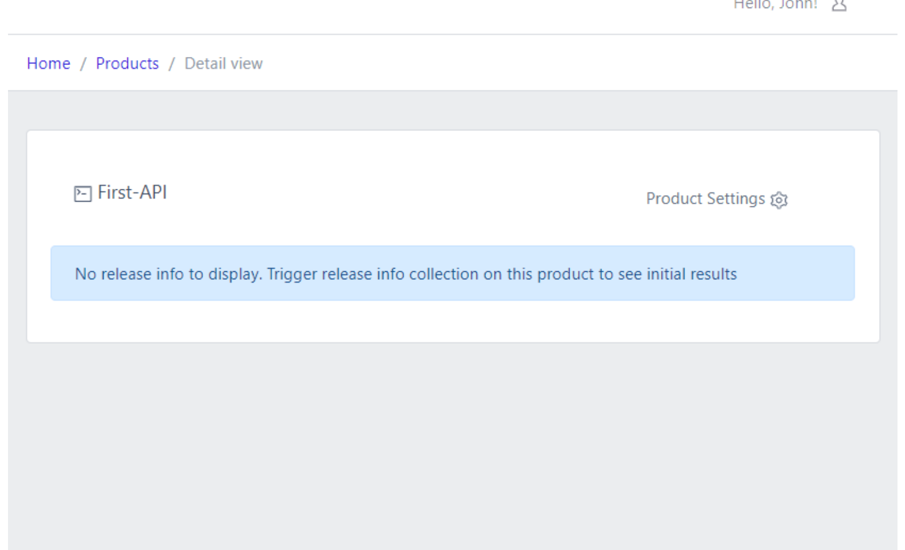

Figure 32. Notice of missing info at the detailed view of a product

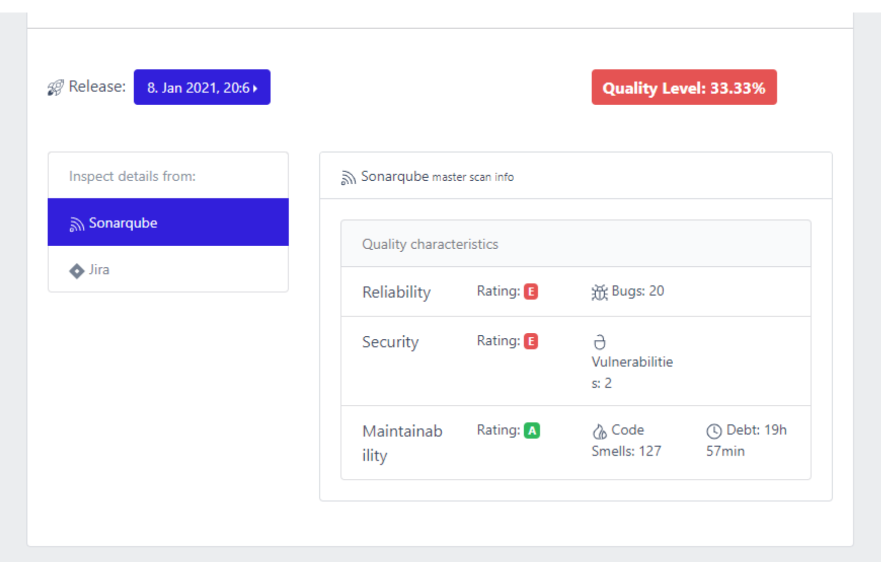

Figure 33. Sonarqube info block of a release at the detailed view of a product

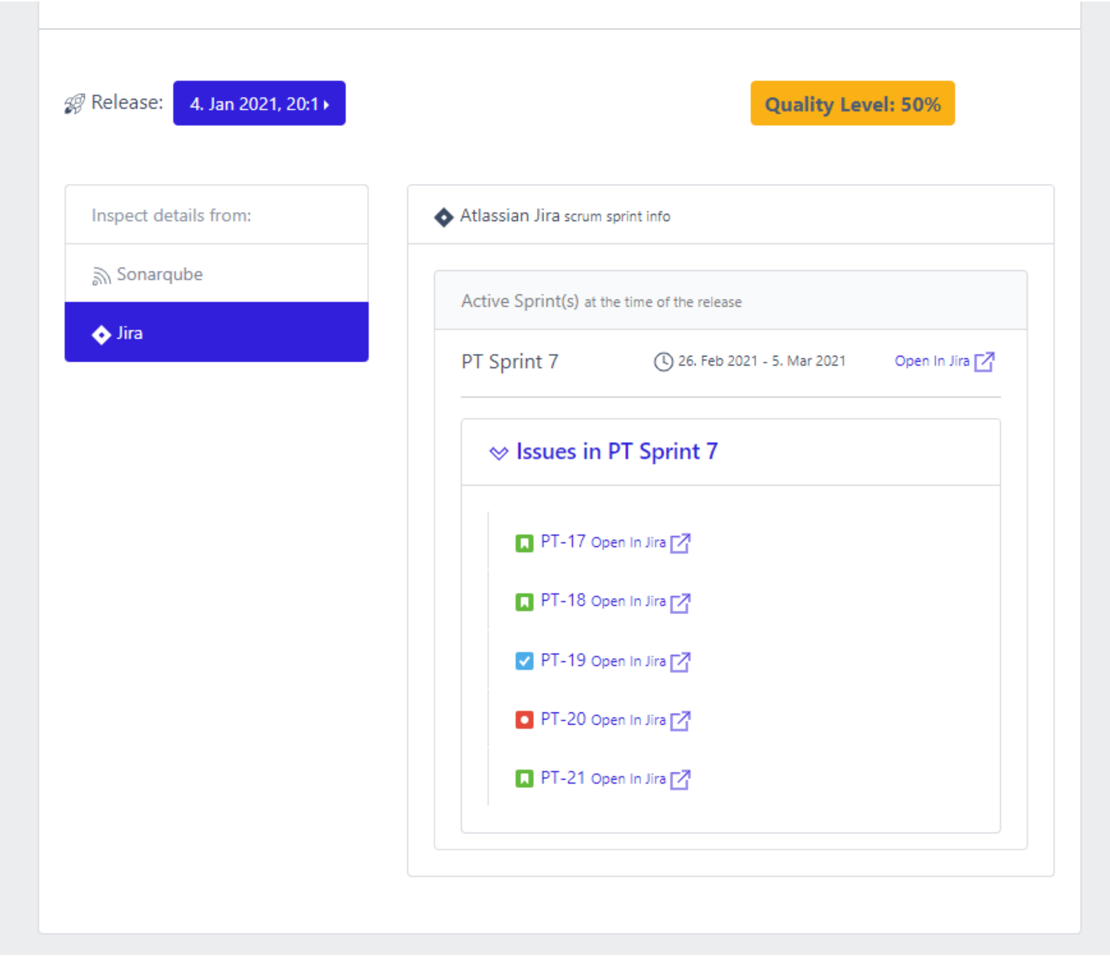

Figure 34. Jira info block of a release at the detailed view of a product

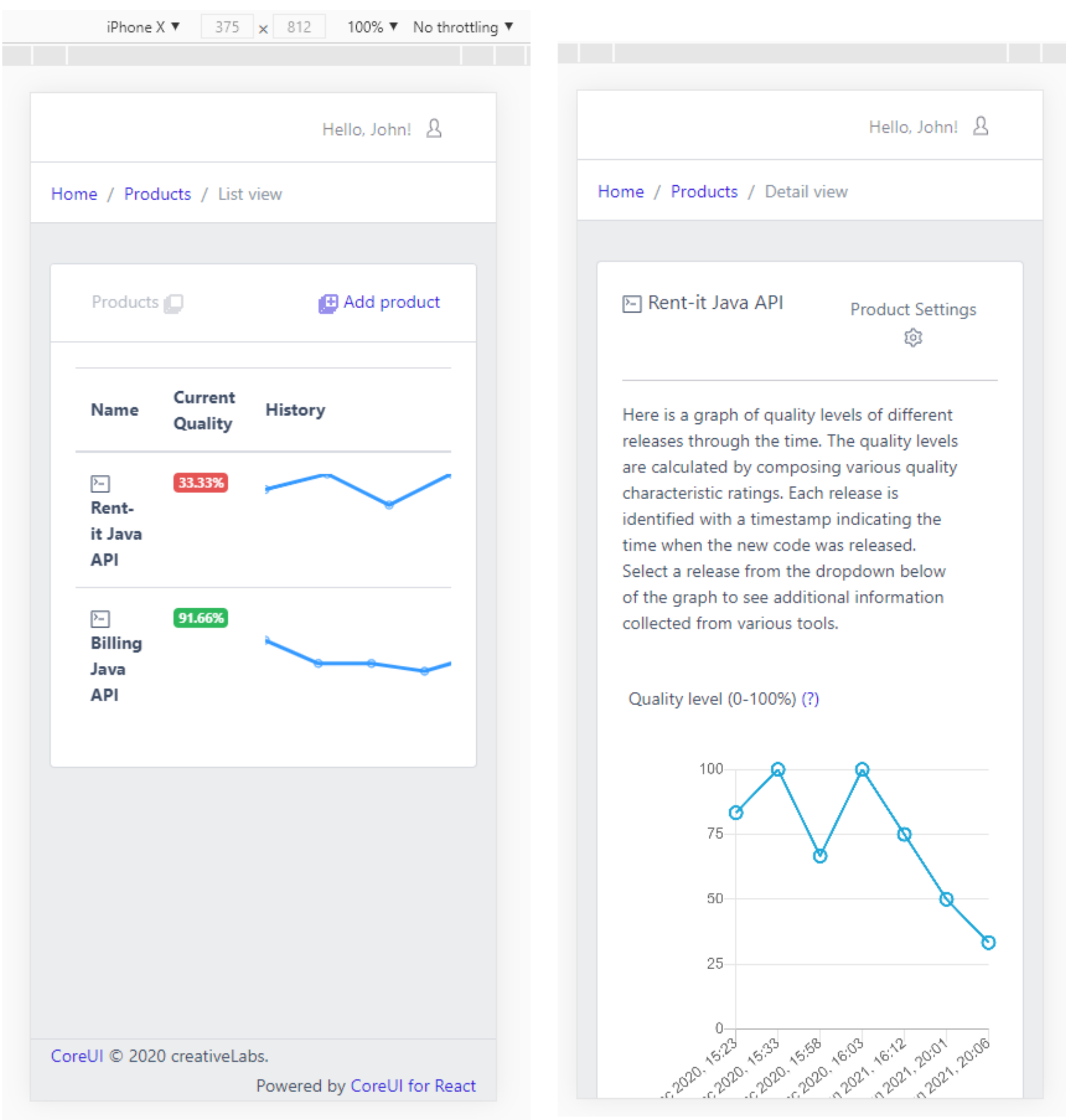

Figure 35. Mobile views of the PQD

III. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Kert Prink**,
(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   A dashboard to visualize the product quality level,
   (title of thesis)

   supervised by Ezequiel Scott.
   (supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Kert Prink
10.05.2021