

DAS Departamento de Automação e Sistemas
CTC Centro Tecnológico
UFSC Universidade Federal de Santa Catarina

Development using Software Quality Control Techniques of a Modern Application for the Shop Floor

*Report submitted to the Universidade Federal de Santa Catarina
as a requirement to approval on the subject:
DAS 5511: Projeto de Fim de Curso*

Rafael Scheffer

Berlin, November 2018

Development using Software Quality Control Techniques of a Modern Application for the Shop Floor

Rafael Scheffer

This paper was judged in the context of the subject:

DAS 5511: Projeto de Fim de Curso

and approved in its final form by the Course

Engenharia de Controle e Automação

Prof. Rômulo Silva de Oliveira

Examiner Committee:

Ricardo Grützmacher
Mentor at Rolls-Royce

Prof. Rômulo Silva de Oliveira
Mentor at UFSC

Hector Bessa Silveira
Responsible for the Course

Leonardo Martins Rodrigues, Committee member

Matheus Felipe Souza Valin, Committee member

Ígor Assis Rocha Yamamoto, Committee member

Abstract

Traditional companies, such as Rolls-Royce, are heavily investing in the digitalization of their manual and analog processes. In the department of Assembly and Test, a good example of this type of process is the monitoring of jet engines production in the shop floor, which is done by in-person verification of the module builds, Excel spreadsheets and paper-based systems. These methods are intrinsically slow, inefficient and error prone.

As a solution, the development of Shopino, a modern web application for monitoring and data visualization for the shop floor, was proposed. In order to lower costs and save time, the application was developed on top of an existing platform, called Engino. However, since Engino was not finished, it had to be improved and tested. For this purpose and to ensure that Shopino is a secure and high quality application, software quality control techniques were applied during the planning and development of the application. Furthermore, a Test-Driven Development methodology was used in combination with an agile methodology.

As a result, a initial version of the application was implemented and integrated with two shop floor systems, which are responsible for inspecting the quality of the built modules in the shop floor. These integrations were made available in a very short time and, thus, created immediate value for the company. Moreover, reviews and all levels of automated tests were implemented, ensuring the quality of the software produced.

Keywords: web-development, shop-floor, quality-control, tdd, elixir, react.

List of Figures

| | |
|---|----|
| Figure 1 – Rolls-Royce family day in Dahlewitz gathered 5000 people on 29/06/2017 [11]. | 15 |
| Figure 2 – View of one of the shop floors in Rolls-Royce Dahlewitz [10]. | 15 |
| Figure 3 – The four dimensions of quality according to Chemuturi [4, p. 26]. | 18 |
| Figure 4 – The PDCA cycle [19, p. 112]. | 22 |
| Figure 5 – Quality control loop [19, p. 113]. | 23 |
| Figure 6 – Landing page of Engino. | 32 |
| Figure 7 – Overview of some of the shop floor systems currently in place. | 32 |
| Figure 8 – Use Case Diagram. | 35 |
| Figure 9 – Simplified view of the current Engino database. | 36 |
| Figure 10 – Simplified view of the final database. | 38 |
| Figure 11 – Planned schedule. | 49 |
| Figure 12 – View of the development environment. | 51 |
| Figure 13 – Step-by-step guide to install the Oracle client. | 51 |
| Figure 14 – Comparison of the old frontend file structure (left) and the new one (right). | 52 |
| Figure 15 – New GraphQL schema structure. | 53 |
| Figure 16 – Permission screen. | 54 |
| Figure 17 – Test to check the view scope by the user access groups. | 56 |
| Figure 18 – Map importer test. | 57 |
| Figure 19 – Integration test for the login mutation request in the backend. | 58 |
| Figure 20 – End-to-end test for one of the authentication flows. | 59 |
| Figure 21 – Shop floor view management screen. | 60 |
| Figure 22 – Dialog to display shop floor systems data. | 62 |
| Figure 23 – Heat map displaying the amount of BAMs per shop floor area. | 62 |
| Figure 24 – Indicator showing the amount of Findings of each shop floor area. | 63 |
| Figure 25 – End-to-end test to check the application flow. | 65 |
| Figure 26 – Shopino main view. | 66 |
| Figure 27 – Frontend code coverage. | 67 |
| Figure 28 – Backend code coverage. | 68 |

List of Tables

| | |
|---|----|
| Table 1 – Test cases for the login resolver | 55 |
| Table 2 – Test case for the userIsAdmin HOC | 57 |
| Table 3 – Case coverage evaluation | 69 |

List of abbreviations and acronyms

| | |
|-------------|---------------------------------------|
| <i>PHP</i> | Hypertext Preprocessor |
| <i>SQC</i> | Software Quality Control |
| <i>SQA</i> | Software Quality Assurance |
| <i>QA</i> | Quality Assurance |
| <i>PDCA</i> | Plan, Do, Check and Act |
| <i>DRY</i> | Don't Repeat Yourself |
| <i>TDD</i> | Test Driven Development |
| <i>JSON</i> | JavaScript Object Notation |
| <i>DOM</i> | Document Object Model |
| <i>MVC</i> | Model-View-Controller |
| <i>API</i> | Application Programming Interface |
| <i>URL</i> | Uniform Resource Locator |
| <i>CI</i> | Continuous Integration |
| <i>CD</i> | Continuous Delivery |
| <i>HOC</i> | Higher Order Component |
| <i>LDAP</i> | Lightweight Directory Access Protocol |
| <i>CRUD</i> | Create, Read, Update and Delete |
| <i>ES6</i> | ECMAScript 6 |

Contents

| | | |
|----------|--|-----------|
| 1 | INTRODUCTION | 11 |
| 1.1 | The problem | 11 |
| 1.2 | Objectives | 11 |
| 1.2.1 | General objectives | 11 |
| 1.2.2 | Specific objectives | 12 |
| 1.3 | Methodology and structure | 12 |
| 2 | ROLLS-ROYCE PLC | 14 |
| 2.1 | Rolls-Royce Deutschland | 14 |
| 2.2 | Shop floor | 14 |
| 2.3 | Engino | 16 |
| 3 | THEORETICAL BACKGROUND AND TECHNOLOGIES | 17 |
| 3.1 | Quality | 17 |
| 3.2 | Software quality | 19 |
| 3.2.1 | Software quality dimensions | 20 |
| 3.2.2 | Coding best practices | 21 |
| 3.3 | Software quality control | 21 |
| 3.3.1 | Continuous software quality control | 21 |
| 3.4 | Software reviews | 23 |
| 3.5 | Software testing | 24 |
| 3.5.1 | Testing approaches | 24 |
| 3.5.1.1 | Black-box testing | 24 |
| 3.5.1.2 | White-box testing | 24 |
| 3.5.1.3 | Gray-box testing | 24 |
| 3.5.1.4 | Manual testing | 24 |
| 3.5.1.5 | Automated testing | 25 |
| 3.5.2 | Testing levels | 25 |
| 3.5.2.1 | Unit testing | 25 |
| 3.5.2.2 | Integration testing | 25 |
| 3.5.2.3 | System testing | 25 |
| 3.5.2.4 | Acceptance testing | 26 |
| 3.5.3 | Testing techniques | 26 |
| 3.5.3.1 | Regression testing | 26 |
| 3.5.3.2 | End-to-end testing | 26 |
| 3.5.4 | Testing disadvantages | 26 |

| | | |
|------------|--|-----------|
| 3.6 | Methodologies | 26 |
| 3.6.1 | Scrum | 27 |
| 3.6.2 | Test-Driven Development | 27 |
| 3.7 | Technologies | 27 |
| 3.7.1 | Single Page Applications | 27 |
| 3.7.1.1 | React | 28 |
| 3.7.2 | Phoenix | 28 |
| 3.7.2.1 | Elixir | 28 |
| 3.7.3 | PostgreSQL | 28 |
| 3.7.4 | GraphQL | 28 |
| 3.7.4.1 | Web Socket | 29 |
| 3.7.5 | Git and GitLab | 29 |
| 4 | THE PROBLEM | 30 |
| 4.1 | Engino platform | 31 |
| 4.2 | Shop floor systems | 31 |
| 4.3 | Shopino | 32 |
| 4.4 | Software quality | 33 |
| 5 | PROJECT PLANNING | 34 |
| 5.1 | Software Modelling | 35 |
| 5.1.1 | Requirements | 35 |
| 5.1.2 | Database | 36 |
| 5.1.3 | Technology stack | 38 |
| 5.1.3.1 | Frontend | 38 |
| 5.1.3.2 | Backend | 39 |
| 5.1.3.3 | Gitlab | 39 |
| 5.2 | Quality control plan | 39 |
| 5.2.1 | Quality model | 40 |
| 5.2.2 | Quality plan | 40 |
| 5.2.2.1 | Engino platform review plan | 40 |
| 5.2.2.2 | Shopino review plan | 41 |
| 5.2.2.3 | Engino platform test plan | 41 |
| 5.2.2.4 | Shopino Test plan | 42 |
| 5.3 | Engino platform development plan | 42 |
| 5.3.1 | Basic project documentation | 42 |
| 5.3.2 | Restructuring of the frontend file structure | 43 |
| 5.3.3 | Restructuring of the API schema | 43 |
| 5.3.4 | Modification of the communication protocol | 43 |
| 5.3.5 | User authentication system | 44 |

| | | |
|------------|---|-----------|
| 5.3.5.1 | Backend logic | 44 |
| 5.3.5.2 | Frontend logic | 44 |
| 5.3.6 | Permission system | 45 |
| 5.4 | Shopino development plan | 45 |
| 5.4.1 | Upload and manage shop floor layouts | 45 |
| 5.4.2 | Create, update and delete zones | 46 |
| 5.4.3 | Display shop floor systems data related to a shop floor area | 46 |
| 5.4.4 | Provide overlays on top of shop floor areas with different type of indicators | 46 |
| 5.5 | Shop floor systems analysis | 46 |
| 5.6 | Shop floor systems integration | 47 |
| 5.7 | Methodology | 48 |
| 5.8 | Schedule | 49 |
| 6 | IMPLEMENTATION AND TESTING | 50 |
| 6.1 | Development environment | 50 |
| 6.2 | Testing environment | 50 |
| 6.3 | Engino platform development | 50 |
| 6.3.1 | Basic project documentation | 51 |
| 6.3.2 | Restructuring of the frontend file structure | 51 |
| 6.3.3 | Restructuring of the API schema | 52 |
| 6.3.4 | Modification of the communication protocol | 52 |
| 6.3.5 | User authentication system | 53 |
| 6.3.6 | Permission system | 54 |
| 6.4 | Engino platform testing | 55 |
| 6.4.1 | Unit tests | 55 |
| 6.4.1.1 | Testing authentication API resolver in the back end | 55 |
| 6.4.1.2 | Testing login screen in the frontend | 55 |
| 6.4.1.3 | Testing permission persistence module in the back end | 55 |
| 6.4.1.4 | Testing scoping of views belonging to access groups in the back end | 56 |
| 6.4.1.5 | Testing the tiles importer | 56 |
| 6.4.1.6 | Testing access permission to admin areas | 57 |
| 6.4.2 | Integration tests | 58 |
| 6.4.2.1 | Testing the integration between the authentication modules in the front and back end | 58 |
| 6.4.3 | End-to-end tests | 58 |
| 6.4.3.1 | Testing the authentication flows | 59 |
| 6.5 | Shopino development | 59 |
| 6.5.1 | Create tables | 60 |
| 6.5.2 | Upload and manage shop floor layouts | 60 |
| 6.5.3 | Integration with BAMs and Findings database | 61 |

| | | |
|------------|---|-----------|
| 6.5.4 | Create, update and delete zones | 61 |
| 6.5.5 | Display shop floor systems data related to a shop floor area | 61 |
| 6.5.6 | Provide overlays on top of shop floor areas with different type of indicators | 61 |
| 6.6 | Shopino testing | 62 |
| 6.6.1 | Unit tests | 62 |
| 6.6.1.1 | Testing API resolvers | 63 |
| 6.6.1.2 | Testing shop floor views management screen | 63 |
| 6.6.1.3 | Testing map component | 63 |
| 6.6.1.4 | Testing zone component | 64 |
| 6.6.2 | Integration tests | 64 |
| 6.6.2.1 | Testing the integration between the modules in the front and back end | 64 |
| 6.6.3 | End-to-end tests | 64 |
| 6.6.3.1 | Testing the application flow | 64 |
| 7 | RESULTS | 66 |
| 7.1 | Test Coverage | 66 |
| 7.1.1 | Code coverage | 67 |
| 7.1.1.1 | Frontend | 67 |
| 7.1.1.2 | Backend | 68 |
| 7.1.2 | Case coverage | 69 |
| 8 | CONSIDERATIONS AND PERSPECTIVES | 70 |
| 8.1 | Result analysis | 70 |
| 8.2 | Future perspectives | 71 |
| | References | 72 |
| | APPENDIX A – PRODUCTION WIKI | 74 |
| | APPENDIX B – USER SOCKET MODULE | 76 |
| | APPENDIX C – AUTHENTICATION HOC | 77 |

1 Introduction

In the last decade, thanks to the advance of technology and the reduction of the cost of computers, internet and server infrastructure, software has become affordable as never before. Consequently, traditional industries are now seeking the digitalization and improvement of their processes through softwares, in order to save time, cut spending and increase productivity.

At the same time, web applications have gained increasing popularity in the field of software development, mainly because of its high flexibility, large reach and great compatibility. These characteristics are attractive for large and multinational companies, since it enables the application to be used in all their locations, in practically any device and with almost no setup.

Nevertheless, the widespread nature of a web application increases the risk of attacks and bug exploitation, specially when dealing with sensitive information or running in critical areas of a company, such as the shop floor. In these cases, the software development requires special attention regarding quality and security, which can be achieved by using techniques of Software Quality Control.

1.1 The problem

Currently, the monitoring of logistics and quality of the shop floor is done by in-person verification of the module builds, Excel spreadsheets and paper-based systems. These methods are intrinsically slow, inefficient and error prone.

As a solution, the Assembly and Test department chose to develop Shopino: a web application for the monitoring and visualization of the shop floor. In order to lower costs and save time, the application is going to be built on top of an already existing application, called Engino.

Engino is a modern web platform for marking engine views, developed by the department of Technical Data Services.

1.2 Objectives

1.2.1 General objectives

The general objectives tackled in this document are:

- Develop Shopino, a monitoring and visualization system for the shop floor;
- Analyze and choose the shop floor systems to be integrated with Shopino;
- Ensure that Shopino is a high quality software and has no critical bugs.

1.2.2 Specific objectives

In order to solve the problem and achieve the general objectives described above, this project aims to:

- Fulfill Shopino software requirements;
- Integrate shop floor systems without disturbing the production;
- Improve the software security and quality of the Engino platform;
- Develop Shopino using a Test-Driven Development methodology, meaning that all modules are covered by, at least, a unit test;
- Implement integration tests to assure that different parts of the application work correctly together;
- Develop end-to-end tests to assure that the final software system meets the specified quality requirements;
- Create regression tests so that new changes to the software don't break the application.

1.3 Methodology and structure

This project was developed using a mix of agile and test-driven development methodologies and is divided in the following 8 chapters:

- **Chapter 1:** Contextualizes the problem of the thesis and enumerates the objectives to be achieved;
- **Chapter 2:** Describes briefly the company and the environment where the project was developed;
- **Chapter 3:** Explains the theories of Software Quality Control and presents the methodologies and technologies used in the project;
- **Chapter 4:** Presents the problem of the project and the challenges needed to be overcome;

- **Chapter 5:** Contains the project planning and defines the methodology, schedule and metrics to be used;
- **Chapter 6:** Describes the implementation and the tests created;
- **Chapter 7:** Presents the result obtained in terms of quality of software, test coverage and metrics;
- **Chapter 8:** Contains the final considerations, a discussion about the results obtained and the future perspectives for Shopino.

2 Rolls-Royce plc

Initially established in 1884 by Henry Royce, Rolls-Royce first manufactured dynamos and electric cranes, until in 1904 it built its first motor car. In 1914, due to the beginning of the First World War, it started producing aircraft engines for the military. In 1953, after the end of the Second World War, Rolls-Royce entered the civil aviation market.

Today Rolls-Royce employs almost 50,000 people in 50 countries and it is the world's leading engine supplier for business aviation, powering over 3,000 aircraft in service today, with a 42% market share [13]. The company has customers in more than 150 countries, comprising more than 400 airlines and leasing customers, 160 armed forces, 4,000 marine customers including 70 navies, and more than 5,000 power and nuclear customers.

2.1 Rolls-Royce Deutschland

Rolls-Royce Deutschland is situated in Dahlewitz, near Berlin, and Oberursel, near Frankfurt/Main. It has invested more than 3.6 billion euros in the development of the BR700 engine family. It is the only officially approved jet engine manufacturer in Germany with development, manufacturing and maintenance license for modern civil and military turbine engines [14].

At the Dahlewitz location, it has been manufactured more than 7000 aircraft engines of small and medium size for the past 25 years. Currently, about 2800 people work in this location, where the shop floor is the area of the company that contains the highest number of workers and machines. Figure 1 displays an aerial view of part of the Rolls-Royce Dahlewitz facilities.

2.2 Shop floor

The production in Dahlewitz is divided in two facilities: one dedicated for the assembly of the big Trent XWB turbofan jet engines, that power the Airbus A350, and the other for development and final assembly of the smaller BR700 and the new Pearl 15 engines. When finished, one Trent XWB engine can weight over 7000 kg [18] and provide between 84200 and 97000 lbf [17] of thrust. In one of Rolls-Royce biggest sales, it was sold for about 35 millions of dollars each in 2007 [16].

Since Rolls-Royce is part of the aerospace industry, it must comply with numerous



Figure 1 – Rolls-Royce family day in Dahlewitz gathered 5000 people on 29/06/2017 [11].

regulations and quality checks. In order to be compliant, there are rigorous tests and quality assurance processes in place in the shop floor. Depending of the rework required to pass the quality tests and the availability of the parts, one engine can take several months to be built and assembled.

The department of Assembly and Test is responsible for assembling and testing the engine modules and where the project of this document was developed. Figure 2 shows one of the shop floors in Rolls-Royce Dahlewitz.



Figure 2 – View of one of the shop floors in Rolls-Royce Dahlewitz [10].

2.3 Engino

Engino is a web platform for visualization and marking of engine views developed in the department of Technical Data Services. It integrates several internal systems by importing data coming from Excel spreadsheets and databases into its own database or by providing a specific link to other systems. The first version of the application was a proof-of-concept designed in 2014 using PHP and quickly gained popularity among the Rolls-Royce engineers. Despite it, the system wasn't flexible and scalable enough, meaning that new integrations took a lot of time to be implemented and the code became hard to maintain.

The solution chosen for these issues was to recreate Engino from the scratch, using a modern architecture and flexible frameworks, such as React, Elixir and Ruby on Rails. Besides not being launched officially yet, Engino has already more features and systems integrated to the platform than before. Nevertheless, the productivity of the developers increased.

3 Theoretical Background and Technologies

In this chapter, the theoretical background about quality, software quality and software quality control is presented. An explanation of the methodologies and technologies used during the project development are also included.

3.1 Quality

The International Organization for Standardization (ISO) defines quality as "the degree to which a set of inherent characteristics fulfills requirements" [1]. Requirements are the product or service specifications, which can be determined by the customer, the producer of the product or an external body, such as the government or a standard body. The inherent characteristics are the never changing product properties that are responsible for delivering the requirements to the customer. Lastly, it can be inferred from the word *degree* that it is possible to evaluate how many and how good the requirements are fulfilled. In other words, that a product can have good, bad or excellent quality.

According to Chemuturi in [4], that definition of quality can lead to the ambiguous inference that "...quality is a continuum, beginning with zero and moving toward, perhaps, infinity" [4, p. 3]. Therefore, it wouldn't be clear who and how should measure the quality of the product. Instead, he argues that specifications are at the heart of quality and that quality should be defined from the standpoint of the provider, since the provider is the actor responsible for the development and assurance of the quality of a product.

Chemuturi concludes defining quality as "an attribute of a product or service provided to consumers that conforms in total to or exceeds the best of the available specifications for that product or service." [4, p. 4]. Thus, quality would be divided in four dimensions: specifications, design development and conformance, as shown in figure 3.

- **Specification quality:** defines the quality of the product specification, which is the first activity to be done and impacts all the other activities;
- **Design quality:** determines how well the product was designed to meet the requirements. If the design is not correctly done, the product may not fulfill its requirements, even if the specifications were well defined and the development was successful;
- **Development quality:** refers to the quality of the manufacture of the product and how well it conforms to the design document;

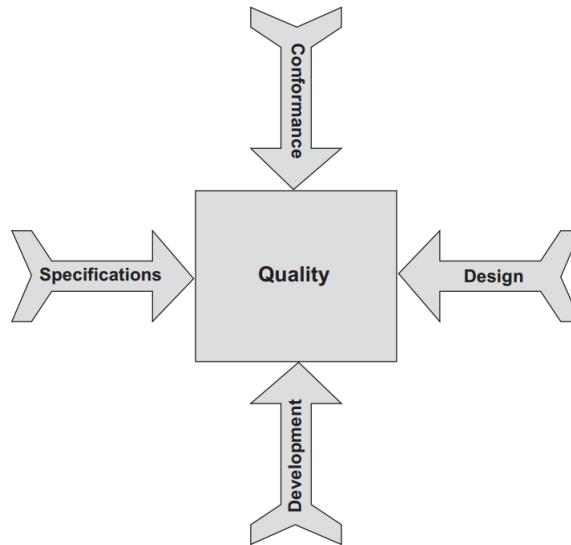


Figure 3 – The four dimensions of quality according to Chemuturi [4, p. 26].

- **Conformance quality:** defines the quality of product in comparison with the specifications and the product design. It uses measurements and metrics to evaluate the quality of the product and what needs to be fixed.

Both these definitions condition quality to the capacity of fulfilling **explicit** requirements. This could be problematic, because requirements can also be **implicit** or based on user expectations. Therefore, even if all requirements are fulfilled, it could be still possible that important requirements are missing or the product falls short of user expectations [19, p. 6]. As an alternative, Wagner proposes in [19] a way to define product quality by using Garvin's approaches to quality:

- **Transcendent approach:** captures the feeling that a product has high quality. It is not a concrete approach, but it helps us to remember that there is always a subjective and immeasurable aspect to quality.
- **User-based approach:** defines quality as a way to satisfy the user needs. This approach is useful for making us aware of the distinction between user satisfaction and quality. Which means that even a cheap product that is not well produced and, therefore, not considered of high quality could satisfy the user.
- **Value-based approach:** consists of assigning costs to conformance and non conformance requirements, comparing them to the benefits for the product and, then, calculating its value.
- **Product-based approach:** describes quality as the differences in the quantity of the desired product characteristics. This approach assumes that the desired product can always be described.

- **Manufacturing approach:** defines quality as the conformance to the product specifications. This approach assumes that the product can be always completely specified and has the problems commented before.

Garvin defends that different approaches are more suitable, depending on the stage of the product's life cycle. In the beginning of the life cycle, the more important approaches are the user-based and value-based, since they capture what is more important and valuable to the user. Then, the product-based approach would describe the more concrete product characteristics in form of a specification. Last, during the development stage, the manufacturing approach would ensure the product fulfills the specification.

3.2 Software quality

Software can be developed and offered as a service or a product, but this project is going to focus on the software as a product approach. A software has the following inherent characteristics that make it different from the other types of product:

1. Software is intangible and not subjected to physical wear. Because of that, it can last, in theory, forever;
2. The software environment is usually in continuous and fast change, whether is the programming language, the platform, the libraries or other softwares and their interfaces;
3. As a consequence of the point 1, a software can be always modified to add new features;
4. As a result of the point 2, a software must be always updated to keep up with the new changes to its environment and to have a decent performance, otherwise its quality decay.

Unlike to a part or an engine, that can be physically tested and its quality evaluated by objective indicators, a software, because of the attributes described above, is difficult to be tested and its quality evaluated. Therefore, the subjects of quality assurance and quality control are even more important for softwares.

Quality assurance is responsible for checking if all the processes related to the quality of the product, such as planning, development and testing, are working. In order to obtain objective evaluations, tools and techniques are used, such as process audits and product audits. Process audits verify the flow and the utility of a process, while product audits verify if the product was produced in conformance with the internal standards, such as forms, templates, coding guidelines, etc [15, p. 264].

In this project, however, the practices of software quality control are the ones being implemented, as discussed in the next section 3.3.

3.2.1 Software quality dimensions

When someone says a software is of high quality, the person is probably referencing a few quality dimensions. Nonetheless, there are hundreds of software quality dimensions and in several of them this software might be of low quality. Below are some of the most important dimensions:

- **Accessibility:** The degree to which software can be used by all kinds of people, including those who need assistive technologies;
- **Compatibility:** The capacity of software to be used in different environments, like operating systems, browsers, etc;
- **Concurrency:** The ability of software to answer multiple requests at the same time;
- **Efficiency:** The capacity of software to run without wasting resources, time or money;
- **Maintainability:** The ease with which software can be updated to add new features or fix bugs;
- **Reliability:** The degree to which software can run without any errors;
- **Security:** The ability of software to prevent unauthorized access, access to sensitive data, data theft, etc;
- **Localizability:** The ability of software to be used in multiple locations, in different languages and time zones;
- **Scalability:** The extent to which software can grow in demand without affecting performance;
- **Reusability:** The degree to which parts of the software can be reused in other applications;
- **Usability:** The capacity of software to be easy to use;
- **Testability:** The ease with which software can be tested.

3.2.2 Coding best practices

Coding best practices and conventions are created to improve code quality. Below is a list with the most popular ones:

- **DRY principle:** DRY stands for Don't Repeat Yourself and aims to avoid code duplication and increase software modularity;
- **Code refactoring:** is the practice of changing code without modifying its functionality, with the purpose of improving other dimensions of quality, such as readability, modularity, performance, etc;
- **Indentation and spacing:** good and consistent code indentation and spacing improve code readability and maintainability;
- **Style guidelines:** each programming language has its own coding conventions, which are created to also improve code readability and maintainability;
- **Deep nesting:** deep nesting is considered a bad coding practice, because it makes the code harder to read;
- **Naming conventions:** the most important ones are camel case (with the exception of the first word, the first letter of each word is capitalized) and underscores (words are separated by underscores);
- **Commenting and documentation:** both increase code readability, but commenting should be avoided when it's too obvious;
- **File and folder organization:** in order to improve code readability and navigation, big files should be divided in smaller ones and grouped into folders with files of the same functionality or context.

3.3 Software quality control

In contrast with quality assurance, that focus in the *prevention* of defects, quality control is a *reactive* process to evaluate the quality of the product and to identify defects. The tools and techniques used for quality control include peer reviews and the different levels of testing, which are presented in the following subsections, along with practices for continuous software quality control.

3.3.1 Continuous software quality control

Continuous quality control was originally created to provide methods for managing the quality of non-software products. It then became a standard approach for process-

oriented quality standards, such as the ISO 9001 [2]. One of the most popular methods is the PDCA cycle, which is composed of four stages: Plan, Do, Check and Act [19, p. 111], as seen in figure 4.

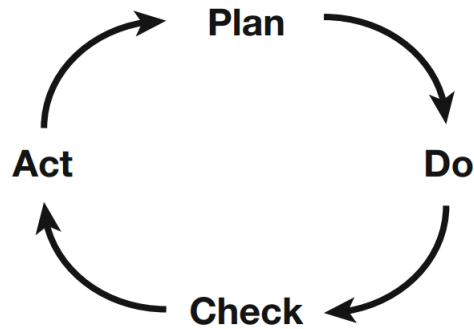


Figure 4 – The PDCA cycle [19, p. 112].

In the Plan stage, it's identified what can be improved and is created a plan to introduce these changes. In the Do stage, the plan is implemented in only a few processes in order to test it. In the Check stage, the results of the changes introduced are evaluated. If the results are satisfactory, then the plan is implemented for all processes in the Act stage.

As discussed in the previous section, a software is always suffering transformations, whether to add new features or to update its compatibility with new changes and technologies. This is such a common process that it received a name: software evolution. Although changes to software are crucial for increasing or maintaining user satisfaction, it can also lead to defects and quality issues. This usually happens because of three factors: short deadlines, lack of developers or resources and lack of training of the developers.

The result is copy and paste of previous codes, workarounds to fix problems, lack of documentation and lack of tests. These bad practices affect the software quality by introducing bugs and compromising efficiency, reliability and maintainability, which also raise the costs for future projects.

Therefore, after every software change, the software should be always reassessed in order to maintain a high quality. In order to obtain this continuous software quality control, Wagner proposes in [19] a feedback loop as an analogy of the PDCA cycle that can be applied to the software development process. Figure 5 shows the quality control loop proposed by Wagner.

First, a set of product goals are set and a quality model is created to specify the quality requirements. Then, the developers implement the first version of the software product, which is going to be evaluated by the SQC professionals or the developers themselves, through reviews, tests and analysis. Last, the results of the evaluations are fed

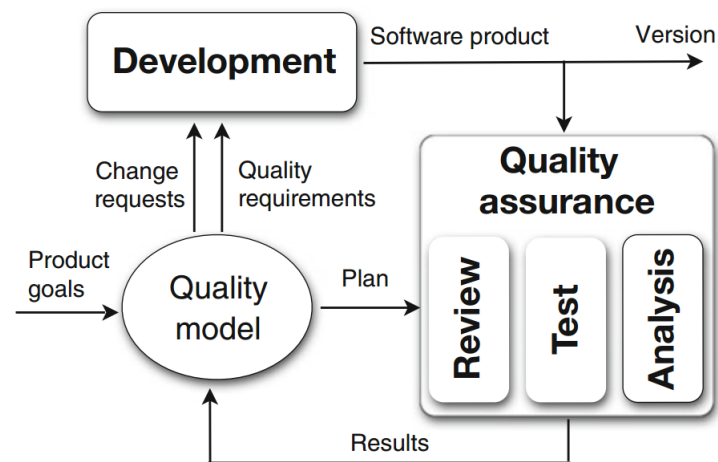


Figure 5 – Quality control loop [19, p. 113].

back to the quality model and compared to the quality requirements. If the quality of the software falls short of the specified, new changes to the software must be made and a new cycle begins. In order to not slow down the speed of the development cycle, long running tests and manual activities, such as code reviews, should be done only in the end of some milestones [19, p. 113].

3.4 Software reviews

Review is the most effective technique in SQC for identifying quality issues and preventing future problems of readability and maintainability. By identifying defects early, it saves time and money if compared to defects found by tests or users. Nevertheless, it's not frequently used in practice. According to a survey done by Cielkowski et al. (2003, pp. 46-51 apud Wagner, 2013, p. 121), only 28% of the respondents performed code reviews. The reasons gave by them included cost, time and lack of training.

There are mainly three types of software reviews: peer reviews, management reviews and audit reviews. This project focus in the implementation of peer reviews, which are divided in the following types:

- **Pair programming** is an activity where two programmers work on the same computer and code together;
- **Inspection** is done by professionals of QA, which examine the software for bugs and defects using a process;
- **Code Review** has the objective to fix bugs and remove vulnerabilities from a software, but it can be done individually and don't need a process;

- **Technical review** is done by a team of professionals of QA, which review the software to evaluate its usability;
- **Walkthrough** is an activity done by the team of developers to ask questions and make comments about the software defects and errors.

3.5 Software testing

Software testing is a risk management strategy, which can have different objectives, such as verify if requirements are met or identify defects and errors. In order to achieve these goals, many testing approaches were created. Some of them are described below.

3.5.1 Testing approaches

3.5.1.1 Black-box testing

Black-box testing, also known as functional testing, is a technique created to check the software's functionality without having information of the internal structure or logic of the program. The objective of this type of test is to evaluate if the specifications are met. The down side of black-box tests is that internal errors might not be detected.

3.5.1.2 White-box testing

White-box testing, or structural testing, consists of tests conditions designed to test the internal structure and logic parts of the software. The goal of these tests is to identify bugs and the logic of modules of the program, but they ignore the fulfillment of the specifications. Besides, another disadvantage of structural tests is that missing modules or data-sensitive errors might not be detected.

3.5.1.3 Gray-box testing

Gray-box testing is a combination of black-box and white-box testing, as it evaluates both the functionality and the logic of the application. The motivation is to save time by avoiding ambiguous tests, which test the same functionality.

3.5.1.4 Manual testing

Manual testing is a process done to identify bugs and defects manually. This usually requires the preparation of a test plan which describes the approach and test cases needed to test the whole application. Therefore, it is time consuming test and requires an exclusive person to do this task.

3.5.1.5 Automated testing

In order for developers to have a fast feedback of the quality of their code, tests need to be run frequently and consume little time. Manual tests are expensive and take too much time to be executed, so the most part of tests needs to be automated. The three phases of the test process that can be automated are: generation, execution and evaluation.

Generation is the most difficult part to automate, because it is the most intellectual demanding phase of the test process. On the other hand, execution is the easiest part to be automated, because there are a lot of testing frameworks that already do this for almost every programming language. Automation of the evaluation of the tests is usually included in the testing frameworks cited before. Yet, depending on the types of tests being created and the complexity of the system, simulation models need to be used in conjunction with the testing framework.

3.5.2 Testing levels

According to the phase of the development cycle, different types of tests are performed on the application. Below the four levels of testing are presented.

3.5.2.1 Unit testing

Unit test, also known as module or component test, evaluates the functionality of the smallest block of a software. Because it is the easiest and fastest type of test to be created, they are usually created first.

3.5.2.2 Integration testing

After all unit tests are created, the integration of the several modules of the application are tested through integration tests. There are different strategies to create these types of tests, such as bottom-up, top-down, big-bang and sandwich. Instead of testing for the functionality of the units, the interaction between different modules is tested to identify interface defects.

3.5.2.3 System testing

When the module integrations and integration tests are done, it means the software is complete. Then, system tests can evaluate the final dimensions of the software quality, such as functionality, performance, security, etc.

3.5.2.4 Acceptance testing

Acceptance test is a type of system and black-box test that is done by the users of the application. The main aspects of the software tested are usually functionality, usability and compatibility, but others can also be tested, depending on the application.

3.5.3 Testing techniques

This subsection describes some popular testing techniques that are used in one or several testing levels and were applied or mentioned during this project.

3.5.3.1 Regression testing

Regression testing is one of the most important types of tests. Its goal is to identify if new changes to software don't break already functioning parts of the application. Therefore, they are created and run after every software change and should be done on all test levels: unit, integration and system.

3.5.3.2 End-to-end testing

End-to-end testing is a type of system test that verifies the functionality of the application as a whole in a production scenario. This means that the application should connect to the same databases, networks and have the same hardware than the application that will run in production.

3.5.4 Testing disadvantages

Although tests can save a lot of valuable money and time by identifying bugs and non conformance to requirements before deploying the software to production, it can also slow down the productivity of the developers if too much time is spent doing duplicated or useless tests. Moreover, having many slow automated tests to run when using continuous integration and continuous testing tools, decreases the iterativity and speed of the development cycle. Thus, it is important to choose the right tests and their amount to be done, in order to avoid having a testing overhead.

3.6 Methodologies

In this section, the methodologies used during the project development are introduced.

3.6.1 Scrum

Scrum is part of the agile software development methodology, whose main principle is to focus on the development of the software itself, instead of documentation. This doesn't mean that agile methodologies don't create documentation, but they only create enough documentation for a new member of the team to understand the software.

In Scrum, the product is divided in small pieces and developed in iterations of 2 to 4 weeks, called sprints. Before each sprint, a planning meeting is done to plan the tasks that should be implemented. Then, after the sprint is concluded, the progress is tracked and the next plan is created in a new meeting. This process provides fast and continuous feedback for all members of the team about the current progress of the project and what needs to be improved.

3.6.2 Test-Driven Development

Test-Driven Development is a software development process that consists in repetitive and short development cycles. First, a test is written for the new functionality that needs to be implemented, then the minimum amount of code to make that test pass is created and, at last, the code is refactored. This process tries to improve the design of the application, making the developer think before writing any code. It also leads to more clean, modularized and flexible code.

3.7 Technologies

This section describes the technologies used during the development of the application.

3.7.1 Single Page Applications

Single Page Applications are web applications that receive only a single web page from the server together with JavaScript files. Then, these applications are themselves responsible for managing the navigation to other web pages via JavaScript. Therefore, after the first response from the server, every other request is made with only the needed information for that specific operation, thus, making the communication between the front and back end very fast. For these type of requests, lightweight data formats are used, such as JSON.

When the JavaScript scripts receive the response from the server, they can change the user interface, by modifying the Document Object Model (DOM).

3.7.1.1 React

React is one of the most used libraries for developing Single Page Applications and building user interfaces. It was developed and is maintained by Facebook. The advantages of React to other similar frameworks, such as Angular and Vue.js, are:

- Easy to learn;
- High flexibility and responsiveness;
- One way data binding, meaning that child elements cannot affect their parents data;
- Virtual DOM, which improves performance by only updating the real DOM with the modified objects;
- Lots of open source libraries.

3.7.2 Phoenix

Phoenix is a MVC framework that does not compromise speed and maintainability [9]. One of its main advantages is the real time layer that works through channels and concurrent web sockets. Phoenix is optimized to have multiple connections in real time, being able to handle up to two millions connections at the same time. It is written in the functional programming language Elixir.

3.7.2.1 Elixir

Elixir is a dynamic, functional language designed for building scalable and maintainable applications [6]. It was built on top of Erlang and is run on the Erlang virtual machine (BEAM), which is known for running low-latency, distributed and fault-tolerant systems. It also provides compile-time metaprogramming with macros and polymorphism via protocols.

3.7.3 PostgreSQL

PostgreSQL is a powerful, open source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads [12]. It also allows custom definitions of data types, index types, functional languages and others.

3.7.4 GraphQL

GraphQL is an open-source data query and manipulation language for APIs. While REST APIs require loading from multiple URLs, GraphQL APIs get all the data needed

in a single request. Furthermore, only the requested data structure is sent in the response, making the communication much more efficient and faster. GraphQL requires the creation of a schema with the API data types and supports reading, writing (mutating) and subscribing to changes to data (realtime updates) [8].

3.7.4.1 Web Socket

Web Socket is a communication protocol that enables interaction between client and server with lower overheads, facilitating real-time data transfer. Unlike HTTP, Web Socket provides full-duplex communication, meaning that both client and server can send messages at any time. HTML5 Web Sockets can provide a 500:1 or, depending on the size of the HTTP headers, even a 1000:1 reduction in unnecessary HTTP header traffic and 3:1 reduction in latency [20].

3.7.5 Git and GitLab

Git is a version-control system for tracking changes in files and coordinating work when files are modified by multiple people. It is primarily used for source-code management in software development. Git was created by Linus Torvalds in 2005 for development of the Linux kernel, with other kernel developers contributing to its initial development [3].

GitLab is a web-based Git repository manager that provides several features, such as wiki, issue tracking and CI/CD pipeline. It is the first single application built from the ground up for all stages of the DevOps life cycle for Product, Development, QA, Security, and Operations teams to work concurrently on the same project [7].

4 The problem

Currently, the process of monitoring the shop floor is done by:

- Physical dashboards that are manually filled in;
- Disconnected data coming from papers, Excel spreadsheets and legacy systems;
- In-person verification of the module builds.

These methods are intrinsically slow, inaccurate and error prone. In order to solve these problems, a web application called Shopino was proposed. Comparatively, a web application could provide the following advantages:

- Less time of engineers and production managers spent to obtain data about the current state of the shop floor, resulting in gain of productivity;
- Low cost of integration with other shop floor systems, which could provide more information and facilitate the decision-making;
- Data visualization and customized graphic interface;
- Storage of data in databases, assuring consistency and standardization of data;
- Low cost to generate reports about the production state, improving the identification of issues and bottlenecks;
- Flexibility to add new data sources, including legacy systems;
- Access of thousands of packages in the web, allowing for fast addition of new functionalities and speeding up the development of the application;
- More accurate view of the current state of the production, leading to better resource allocation, less delays and increase of the company productivity.

Yet, developing a software application in a big company like Rolls-Royce is not a easy task. First, the company has a high demand for new softwares, but the number of developers is limited. Second, the company is part of the Aerospace industry and must comply with a large number of strict regulations and, therefore, its IT department has also internal and strict rules regarding its softwares, databases and networks.

In such a restricted environment, cooperation plays an important role in reducing costs and saving time to develop a software application.

4.1 Engino platform

A brief description and history of the application was given in chapter 2.3.

The high flexibility, high productivity and easy maintainability of Engino platform attracted attention of other departments inside Rolls-Royce, such as the Assembly and Test. By using Engino's architecture and platform, Shopino can have a head start and be in an scalable and flexible environment to grow.

Nevertheless, Engino platform is still a work in progress and has yet to be released officially. Because it was developed by only two developers and in a short time frame, it has almost no tests and needs to be improved in terms of security and software quality. The biggest issues that need to be solved in the current state of Engino platform are the following:

- Lacks basic project documentation;
- Part of the frontend file structure is hard to navigate;
- API schema is a monolith;
- Communication between the frontend and the backend is not optimized for real-time updates;
- There is no authentication system;
- There is no permission system;
- Lack of tests.

Figure 6 shows the current landing page of Engino, which has no authentication system.

4.2 Shop floor systems

The shop floor systems currently in place need to be analyzed and the most suitable ones should be chosen to be integrated into Shopino. The following aspects should be considered in order of importance when choosing the systems that are going to be integrated:

1. Disturbance level of the production;
2. Cost and time required for the integration;
3. Data quality and importance for the monitoring of the production.

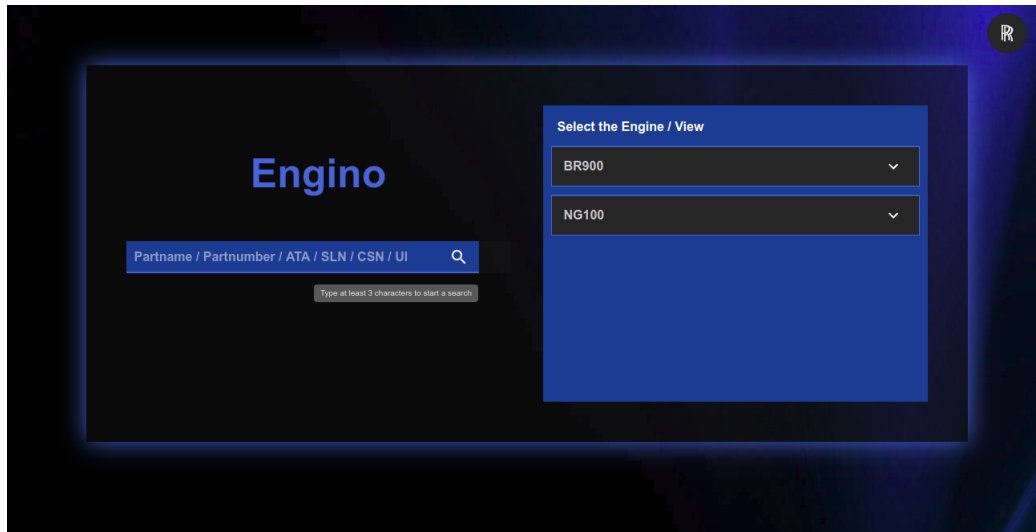


Figure 6 – Landing page of Engine.

Furthermore, the systems shall be grouped in two groups: logistics and quality. The first group needs to contain the systems related to the flow management of people, resources and product. While the second group has to include the systems responsible for determining the quality of the product.

Figure 7 shows an overview some of the shop floor systems currently in place.

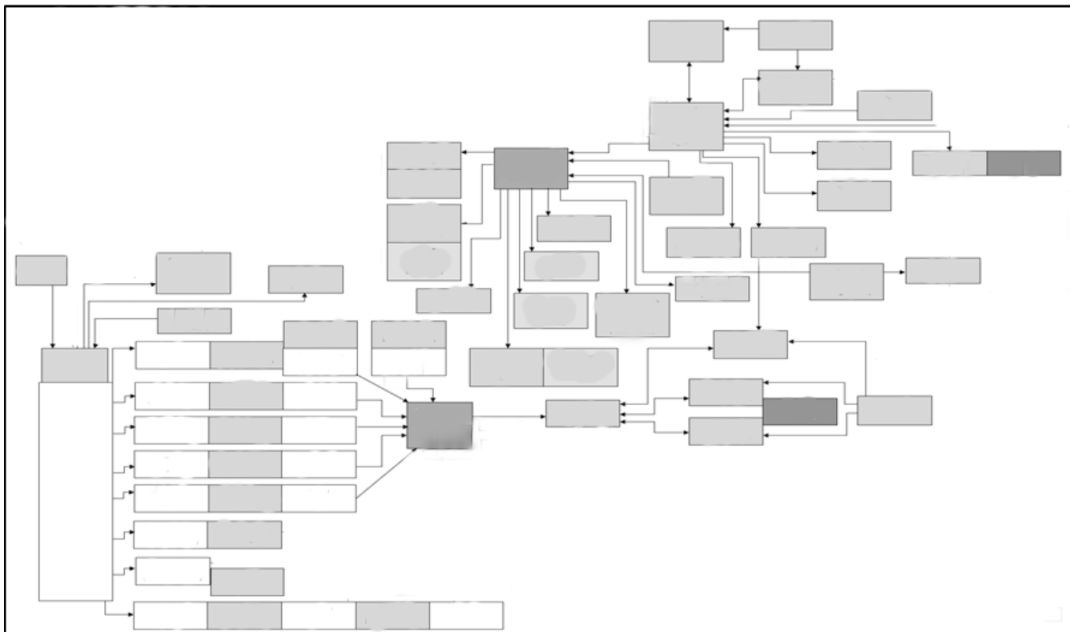


Figure 7 – Overview of some of the shop floor systems currently in place.

4.3 Shopino

In order to improve the monitoring of the shop floor and assist the decision-making of production managers and engineers, Shopino is going to be developed. The idea of

Shopino is to have a similar user interface as Engino, but adapted for the shop floor.

Therefore, instead of displaying engine views in the background, Shopino would present the layout of the shop floor. Instead of marking parts on top of the engine view, it would be drawn rectangles that represent the shop floor areas and workstations. Then, the data coming from the integrations with the shop floor systems could be displayed when the user selects an area or as indicators and heat maps on top of the areas.

4.4 Software quality

As mentioned in the chapter 2.2, a turbine engine costs millions of dollar, takes months to be produced and years to be developed. More important than that, it deals with people lives. A small defect in an engine could cause an accident and cost hundreds of lives. Because of that, all systems and processes in the shop floor must have the highest standards of quality.

Although Shopino is not going to have a direct impact on the engines, it could indirectly lead to wrong decisions of the production managers and engineers regarding the production plans and cost valuable time and money. Therefore, the software application needs to have also high standards of quality, which means it complies with the requirements, has no critical bugs and is secure and reliable.

5 Project planning

In this chapter, the planning to solve the problems described in the previous chapter is defined. In order to better tackle the problems, the plan was divided into the following topics:

1. Software modelling;
2. Quality control plan;
3. Engino platform development plan;
4. Shopino development plan;
5. Shop floor systems analysis;
6. Shop floor systems integration;
7. Methodology;
8. Schedule;
9. Metrics.

First, the software modelling of Shopino was made, describing the requirements of the application, the model of the database and the technologies to be used. Then, a quality control plan was designed, using as reference the loop seen in chapter 3.3.1. The third step was to plan how Engino platform should be improved in order to fulfill Shopino requirements and be compliant with the quality model designed, thus solving the problems listed in chapter 4.1. Next, a plan to develop the remaining requirements of Shopino was created. In the fifth step, the shop floor systems were analyzed and some of them were chosen to be integrated into Shopino. Then, a discussion about how to integrate the selected systems is done.

The seventh step selects the methodologies to be used during the development of the project. After that, a schedule is proposed and the metrics to be measured during and after the project development are chosen.

All the plans discussed in this chapter were created by the author with the supervision of the project manager. They are the result of many meetings and interviews with production managers and engineers.

5.1 Software Modelling

In this section, a software model is proposed for Shopino. It will be divided in four topics: requirements, database and technology stack.

5.1.1 Requirements

The problem described in chapter 4 of improving the monitoring of the shop floor was generic and open to multiple interpretations. Therefore, many meetings were done with production managers and engineers, in order to raise the requirements for Shopino. In order to facilitate the generation of the requirements, a simple Use Case Diagram, as seen in figure 8, was created.

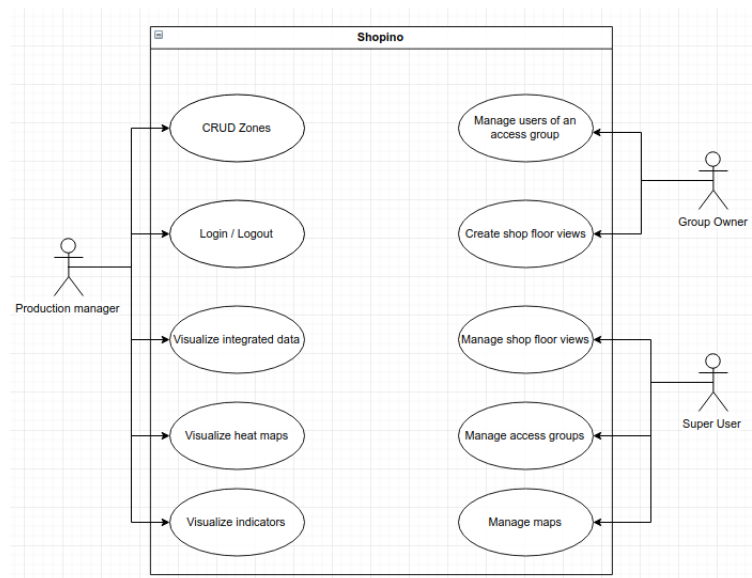


Figure 8 – Use Case Diagram.

Below are the final requirements established:

- User authentication system;
- Permission system;
- Upload and manage shop floor layouts;
- Create, update and delete zones;
- Display shop floor systems data related to a shop floor area;
- Provide overlays on top of shop floor areas with different type of indicators.

User authentication system will be responsible for controlling the access of the application, with the objective of maintaining a session of who is logged in and ensuring

that contractors and guests can't access the application. The permission system will allow certain parts of the application only for user groups, which will be created and managed by super users and the group owners. Both these functionalities will be developed for the whole platform of Engino, since all other applications can benefit from it. The Engino platform development plan is described in section 5.3.

Shop floor layouts are the images of the shop floor, which will be the background of the application and facilitate the visualization and navigation of the data. Zones are overlays that will be drawn on top of the shop floor layout and will be associated to a shop floor area. Finally, overlays in form of heat maps, numbers or lists should be available to be selected and, then, be displayed on top of shop floor areas, in order to improve the visualization of key shop floor indicators. A development plan that covers the implementation of these functionalities is presented in section 5.4.

5.1.2 Database

The current Engino database is a PostgreSQL database and contains 49 tables from different projects, such as SAS, TDM and Engino itself. A simplified view of the database is presented in figure 9.

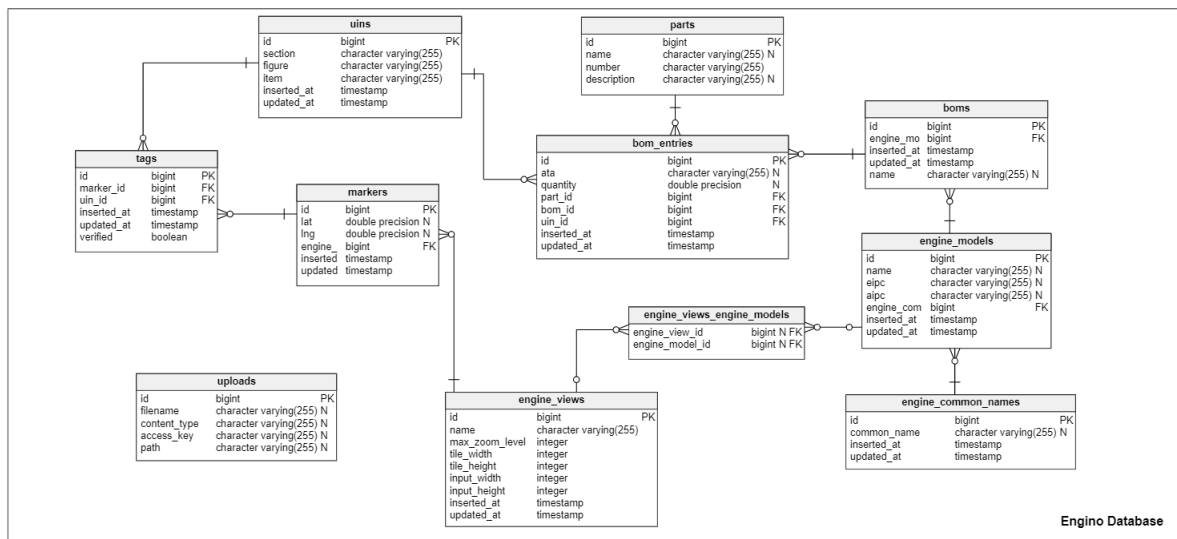


Figure 9 – Simplified view of the current Engino database.

In order to share common tables and save resources, the same database will be used for Shopino.

The most complicated modification of the current database could be the addition of the table *shop_floor_views*, responsible for storing the data of the uploaded shop floor layouts. The reason for it is that there are two ways of implementing this table. The first, and easiest one, would be to mimic the *engine_views* table, so that it would contain information about the tiles uploaded. The second option, that could cause conflicts, would

be to create a new generic table, called maps, that would store information about the image and tiles uploaded and, then, the tables *engine_views* and *shop_floor_views* would only store the name of the view and a foreign key to reference the map that contains the tiles to be used. The advantages of this last approach is:

- Reuse the same tiles for different views;
- Multiple views could share the same map, but have different markers;
- Views with the same map could have different access permissions, meaning, for instance, that each department could have its own view and mark only the parts they are interested in;
- Possibility to mark in the same map parts in one view and zones in another.

Nevertheless, this approach would require to modify the attributes of table *engine_views* and change the logic of the tiles importer and other parts of the application related to engine views, such as the engine view management screen. These would have to be done without interfering with the already uploaded tiles of every engine view.

After discussing this issue with the project manager of Engino, it was decided to take the second approach, because of the advantages presented and the possibilities that are being open for future features and projects. In the next chapter, the implementation of this approach will be discussed in section 6.5.1.

For the implementation of the permission system, 4 new tables will have to be created:

- **access_groups**: contains basic information about the group, such as: name, slug, description and owner;
- **users_access_groups**: cross-reference table to insert users into groups. This is done by relating a *group_id* to a user number;
- **engine_views_access_groups**: cross-reference table to insert engine views into groups. This is done by relating a *access_group_id* to a *engine_view_id*;
- **shop_floor_views_access_groups**: cross-reference table to insert shop floor views into groups. This is done by relating a *group_id* to a *sf_view_id*.

Finally, a table *zones* will be created to store information about the map coordinates of the rectangle drawn and a foreign key to reference the respective shop floor view.

The final and simplified view of the database with the added tables can be seen in figure 10.

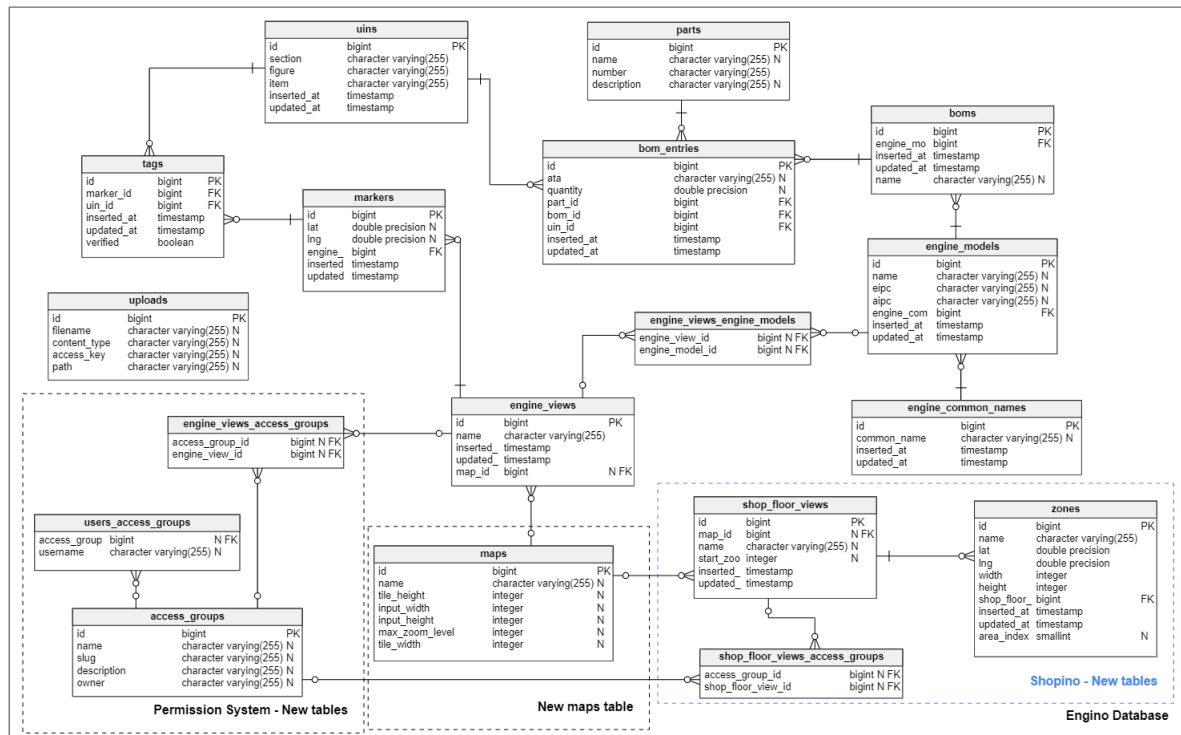


Figure 10 – Simplified view of the final database.

5.1.3 Technology stack

In order to reuse the already existing code in the Engino platform, the same technologies will be used in Shopino. However, some testing libraries will be installed, such as Jest, Enzyme and Puppeteer in the frontend, and ExMachina in the backend.

For the communication between the frontend and the backend, a GraphQL API will be also implemented for Shopino.

Below are the most important technologies of the front and back end.

5.1.3.1 Frontend

The technologies that are going to be used in the frontend are:

- **React:** for building the user interface;
- **Redux:** for managing the application state;
- **Leaflet:** to create interactive maps;
- **Apollo Client:** to provide a GraphQL client;
- **React Apollo:** provides Apollo Client integration for React;
- **Material-UI:** user interface library that implements Google's Material Design;

- **Jest**: to test JavaScript code without setup;
- **Enzyme**: for testing React components;
- **Puppeteer**: to create automated end-to-end tests in a headless browser.

5.1.3.2 Backend

In the backend, the following technologies will be used:

- **Elixir/Phoenix**: as the server side framework;
- **PostgreSQL**: as the main database;
- **Absinthe**: for handling the GraphQL requests and responses;
- **ExMachina**: for easily generating test data.

5.1.3.3 Gitlab

Gitlab will be used for code control and project managing. It provides nice features, such as a task board, issue tracking, time tracking and built-in continuous integration.

5.2 Quality control plan

In this section, the quality model and quality plan of the software quality control loop described in chapter 3.3.1 are created for Shopino. The first step of the loop is to set the product goals:

- Successfully launch version 1 of Shopino;
- Integrate with at least 1 shop floor system before launch;
- Test most use cases;
- Test at least 70% of the Shopino backend code;
- Test at least 50% of the Shopino frontend code.

Then, a quality model has to be created.

5.2.1 Quality model

Below the quality requirements for the product level and the code level are listed, as well as the actions required to fulfill them:

- Product Level Quality
 - **Functionality**: provide the specified functionalities and create end-to-end tests;
 - **Compatibility with Internet Explorer 11 and big monitors**: test the development code in a Windows virtual machine and use the grid system from Material-UI;
 - **Security**: don't allow access of unauthorized users;
 - **Reliability**: test the overall application;
 - **Documentation**: document critical modules of the software.
- Code Level Quality
 - **Reusability**: reuse already existing code of the Engino platform;
 - **Maintainability**: use coding best practices;
 - **Testability**: write modules that are easy to test.

5.2.2 Quality plan

With the objective of achieving the quality characteristics specified in the quality model, a quality plan is going to be created. The quality plan consists of a review plan and a test plan, which will define what parts of the application should be reviewed and tested and what they should be checked and tested for.

In the next subsections, the review and test plans for Engino platform and Shopino are presented.

5.2.2.1 Engino platform review plan

Chapter 3.4 describes the most used types of review.

As the first activity to be done, in order to identify the issues in Engino platform, a manual code review was chosen, because it is quicker and can be done individually by a developer. Below are the parts of the application picked to be reviewed and the reason behind it:

- Top level components of the frontend, which are responsible for the setup of the commonly used libraries. Objectives: maintainability and reusability;

- File structure of the front and back end. Objectives: maintainability;
- Project documentation available. Objectives: documentation and maintainability;
- API schema in the backend. Objectives: maintainability and reusability;
- Authentication modules in the front and back end. Objectives: security and reusability.

The review identified the issues listed in the chapter 4.1, that should be solved in the next development phase of the software quality control loop. In section 5.3, it is presented a plan to fix these issues.

5.2.2.2 Shopino review plan

During the development of Shopino, a manual code review will also be done after each development cycle, with the objective of identifying defects and improving code readability. More about the development plan of Shopino is described in chapter 5.4.

5.2.2.3 Engino platform test plan

Chapter 3.5 describes several testing approaches, the four levels of testing and some testing techniques.

In reason of the short time frame of this project, manual tests won't be executed, because they require too much time to be performed. Therefore, only automated tests will be created.

From the characteristics listed in the quality model in subsection 5.2.1, security and functionality are the most important of them, so a focus will be given to them. Security can be improved by testing the authentication and permission modules and data flows. In Engino platform, functionality is represented by the use of shared modules, such as the tiles importer. Therefore, the following automated tests should be implemented:

- Unit tests
 - Test authentication API resolver in the back end;
 - Test login screen in the frontend;
 - Test permission persistence module in the back end;
 - Test scoping of views belonging to access groups in the back end;
 - Test if admin and regular user have access to admin areas;
 - Test the tiles importer.

- Integration tests
 - Test the integration between the authentication modules in the front and back end.
- End-to-end tests
 - Test the authentication flows with valid and invalid token and credentials.

5.2.2.4 Shopino Test plan

For the code aspect of Shopino, the most important characteristic of the quality model to be tested is functionality. It's represented by the following requirements: CRUD of shop floor layouts, CRUD of zones, display data of integrated shop floor systems and present overlays with indicators on top of shop floor areas.

- Unit tests
 - Test API resolvers in the back end;
 - Test the map component in the frontend;
 - Test the zone component in the frontend;
 - Test shop floor views management screen in the frontend.
- Integration tests
 - Test the integration between the modules in the front and back end.
- End-to-end tests
 - Test the application flow of accessing a shop floor view, creating a zone, clicking on it to display the data from the integrated systems and, then, deleting the created zone.

5.3 Engineo platform development plan

In chapter 4.1, it is listed the biggest issues that Engineo platform has currently. Below a solution for each issue is presented.

5.3.1 Basic project documentation

Currently, the project documentation in Gitlab doesn't explain how to set up the Oracle client in the rails server of the development environment. As a result of not installing the Oracle client, it is not possible to complete the installation of the required

gems and, thus, to start the rails server. In order to fix this, a step-by-step guide will be created, with links to the pages where the required applications can be downloaded from.

Other issue is that there is no documentation of the paths and commands to deploy, run and debug the production server. This caused a problem that only one developer knew how to do these activities, thus creating a dependency. To solve this problem, all the necessary information will be acquired from the developer and summarized into a wiki page in Gitlab.

5.3.2 Restructuring of the frontend file structure

The folder located in `frontend/app/components/presentational` has 16 ungrouped files, which makes the frontend code hard to navigate, reducing the productivity of the developers. As a solution, the files shall be grouped in folders accordingly to their function or context used in the application.

Furthermore, the folder `frontend/app/components/` contains screens, containers and presentational files mixed together. A better file structure would be if they were separated in three different folders: screens, components and containers.

5.3.3 Restructuring of the API schema

Engino platform uses GraphQL as its API system. As described in section 3.7.4, GraphQL requires a schema with types, queries and mutations to be declared. At the moment, one single file, a monolith, located in `lib/engino_web/schema/` contains all the schema declarations. As a result, the readability and maintainability of the API is not ideal, also decreasing the productivity of the developers.

To solve this issue, the file shall be split into smaller files that group the resources of same domain.

5.3.4 Modification of the communication protocol

Currently, the communication between the frontend and the backend uses the HTTP protocol, which is not optimized for real-time updates. In order to fix this issue, it needs to be implemented a web socket protocol. Web socket is much faster than the HTTP protocol and compatible with the GraphQL API (see 3.7.4.1).

For security reasons, the backend will only connect to authenticated web sockets, that is, web sockets that contain a valid token in their header. For non authenticated users, the default communication protocol will be the HTTP, but only the authentication module will be available.

5.3.5 User authentication system

An authentication system is crucial for restricting access of unauthorized users to Shopino and other applications developed in the Engino platform. This issue can be decomposed into four smaller issues. In the backend, the user has to be authenticated and the access to the API restricted to only authenticated users. In the frontend, restrict access and obtain user credentials.

5.3.5.1 Backend logic

For better user experience, a token will be created and stored in the browser cookies, so that the user don't have to log in every time it access the application. The following procedures shall be implemented in the backend to run when the user logs in for the first time:

1. Check if the user credentials (user number and password) are valid through the internal LDAP server;
2. If the credentials are valid, create a token with the user number encoded in it;
3. Send the token to the frontend.

The next time this user will access the application again, the frontend will check if there is a token stored in the browser cookies. If so, it will send the token to the backend for validation. Then, the backend just needs to check if the token is valid and the decoded value has the format specified. If so, the backend sends a :ok message to the frontend.

The same process will happen when the backend receives an API request. The header of the web socket needs to be checked for a valid token. If so, the backend sends a response for the request.

5.3.5.2 Frontend logic

The frontend shall capture the user number and password in the log in page and send it to the backend for verification using the HTTP protocol. If the credentials are valid, the frontend will receive a token and store it in the browser cookies. Then, the setup of the web socket that is going to connect to the backend needs to be done and the token stored in the web socket header.

Engino platform is composed of several application and most of them require authentication, such as: Engino, SAS, TDM, Shopino and Admin. In order to avoid code duplication, a Higher Order Component¹ shall be implemented in the frontend and be

¹ A Higher Order Component (HOC) is a technique in React for reusing component logic. A HOC is a function that receives a component and returns a new component.

used in all those application. This HOC will serve as authentication middleware and block the access of non authenticated users, which means, it will:

1. Get the token from the cookies and check it validity;
2. If it is valid, set up the web socket with the token in the header and redirect the user to the requested page;
3. If it is not valid or there is no token, redirect the user to the log in page.

5.3.6 Permission system

The objective of a permission system is to restrict the access of parts of the application to certain groups of users. In the frontend, a screen to create and manage access groups and their users needs to be implemented. Then, certain screens can be restricted to only particular groups, such as a screen to manage the views uploaded or the screen to manage the permission groups itself. This can be implemented by saving the groups of the authenticated user in the redux session of the frontend and then comparing them to the permitted groups of each screen.

In the backend, the modules responsible for the CRUD of access groups, user access groups and views access groups have to be implemented, as also the GraphQL schema for the API to communicate with the frontend. Furthermore, scopes need to be created, so that queries will return only the views that the respective user has access to.

5.4 Shopino development plan

The Shopino development plan will be divided in four topics, according to the requirements specified in section [5.1.1](#).

5.4.1 Upload and manage shop floor layouts

In the frontend, two screens have to be created: one to create and other to manage shop floor layouts. Both can reuse components used in the respective screens of engine views. Furthermore, because it was decided in section [5.1.2](#) that a new table maps is going to be created, a new screen to manage maps needs also to be implemented. For now, there won't be a screen to upload a map without a shop floor view or engine view, thus a map is always going to be created during the importation of a new view.

In the backend, the tiles importer logic must be updated and basic CRUD modules have to be created for maps and shop floor views. The corresponding GraphQL schema needs also to be implemented.

5.4.2 Create, update and delete zones

A button to create a zone has to be added on top of the shop floor layout in the frontend. After created, other button has to be added so that the user can edit or delete the selected zone.

In the backend, the CRUD modules should be created and added to the GraphQL schema.

5.4.3 Display shop floor systems data related to a shop floor area

The data coming from the integrated systems shall be displayed in the context of a shop floor area. When the user clicks on a shop floor area, a dialog will open and the data should be presented in a table. The systems should be grouped in tabs on top of the dialog and the user should be able to navigate through the tabs by sliding the screen left and right. The idea behind it is to facilitate the navigation in big screens, such as the big monitors in the shop floor.

In the backend, the queries have to be created in the GraphQL schema. More about the integration with the shop floor system in section 5.6.

5.4.4 Provide overlays on top of shop floor areas with different type of indicators

In the frontend, a button will be created on the top of the screen, so that an overlay type can be selected. Then, the corresponding module of the selected overlay should be rendered. Therefore such modules have to be created.

In the backend, the queries to provided indicators must be created and added to the GraphQL schema.

5.5 Shop floor systems analysis

Since there are a large number of systems to be analyzed and the time frame of this project is short, it was agreed between the Shopino stakeholders that only the systems related to quality are going to be focused in this project. The systems analyzed are described below:

- **Inspect:**
 - Defects found during the engine module build;
 - Generates reports with amount of defects, root causes and if it was accepted, repaired or scrapped;

- Provides heat map with amount of defects found on each module build area in the shop floor;
 - Stored in a database in Derby and managed by a third party company.
- **Measure Link:**
 - Collects data from production and measurement machines in real time;
 - Provides a process analyzer to check trends and prevent future issues;
 - System was created by an external company and can only be used with local connected hardware lock.
 - **BAM:**
 - Identifies deviations in the build process;
 - Manual document;
 - Raised BAMs are digitalized and inserted in a local MySQL database.
 - **Findings:**
 - Defects found during inspection;
 - Stored in the same database as BAM.

In order to integrate Shopino with the Inspect system, it is required to ask for access permission to the database in Derby. However, the database is managed by a third party company, which will probably take a long time to grant it and could also charge a fee. Therefore, because it has a high cost and time required for the integration and it doesn't fulfill criteria number 2, mentioned in chapter 4.2, this system won't be integrated.

Measure Link could be integrated by sending the collected data to an endpoint in Shopino and saving it in the database. Nonetheless, it was developed by an external company and can only run with a local connected hardware lock, thus making the necessary modifications in the software code impossible to be done without paying the software company. For this reason, this system won't be integrated with Shopino.

BAM and Findings systems could be integrated together, since they use the same local MySQL database. The only requirement for this integration is the access permission from the database owner. Therefore, they could be easily integrated with Shopino, without any additional costs. For these reasons, both systems will be integrated.

5.6 Shop floor systems integration

As described in the previous section, the selected systems to be integrated, Findings and BAMs, use the same database to store their data. Therefore, it needs to be created

a connection to this database, which is a MySQL database and will be called Findings database. This can be done by using a MySQL database adapter for Elixir/Ecto called Mariaex, that will enable having Ecto features in the same environment as the rest of the project.

Once the connection to the Findings database is done, there are two options of how to deal with the data:

1. Import the required tables to the Engino database. Advantages:
 - Faster queries, because the queried database will be in the same server as the Shopino server;
 - It won't add additional load to the Findings database;
 - Control of the data;
 - No dependency to an external system.
2. Query directly to the Findings database. Advantages:
 - Guarantee of live data;
 - No need for creating and managing new tables in the Engino database.

A test was conducted to compare the duration of similar queries to Engino and Findings databases. It resulted in not a big time difference between the two databases. Besides, Shopino is only going to be used by shop floor managers and engineers, so there won't be enough users to cause a heavy load on the database. Nevertheless, it was clarified with the database owner that the database can bear the Shopino load.

Therefore, the only remaining advantages of option number 1 are: control of the data and no dependency to a external system. But, since the Findings database is used in the shop floor, it must be reliable and stable. Thus, mainly for the guarantee of live data, the option number 2 will be implemented.

5.7 Methodology

The methodologies used were Scrum and Test-Driven Development. Scrum, an agile methodology, was chosen because of the short time frame of the project and the flexibility it provides. As described in chapter 3.6.1, in Scrum, tasks are implemented in periods no longer than one month, called sprints. Then, the progress is verified and a new plan is done for the next sprint.

In Test-Driven Development, first a small test that fails is created, then changes are made to the code until the test passes. This process is repeated until the desired feature

is implemented. TDD was chosen, because of the advantages it provides, such as fewer bugs, better documentation, less time spent on rework, cleaner code and better software quality. However, since the user interface is really hard to test, only some components of the frontend were implemented using TDD.

5.8 Schedule

For achieving the the goals set in chapter 1.2 in the time available, a schedule was created with the tasks described in this chapter. Figure 11 displays the planned schedule.

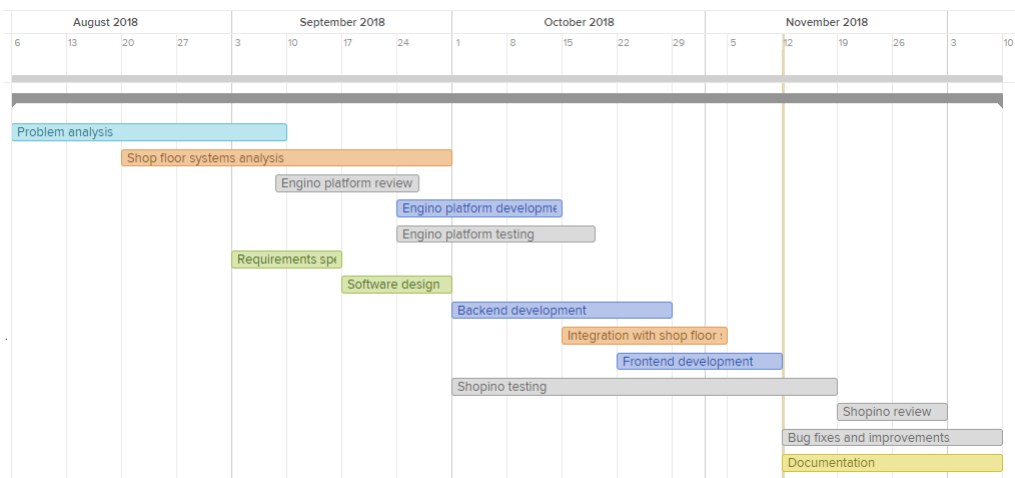


Figure 11 – Planned schedule.

6 Implementation and testing

In this chapter, it is described the implementation of the planning from chapter 5 and the modifications that had to be made.

Except for part of the Permission system and the integration with the LDAP system in the user authentication system, which were implemented by other developer, all activities reported in this chapter were done by the author.

6.1 Development environment

The development environment consists of a notebook with Ubuntu Desktop 18.04 as the operating system. The following softwares were installed:

- **Visual Studio Code**: was chosen as the code editor, because of its speed and vast quantity of libraries for Elixir and Javascript;
- **Yarn**: to add and manage the frontend packages;
- **Node.js**: with the purpose of running webpack-dev-server, so that the browser can be updated automatically every time a Javascript file has changed;
- **pgAdmin 3**: for managing the PostgreSQL database;
- **Oracle VM VirtualBox**: to run a Windows virtual machine inside the Linux environment and test the development changes in the Internet Explorer, the production browser.

The softwares needed to run the technologies listed in section 5.1.3 had also to be installed.

6.2 Testing environment

The testing environment will be the same as the development environment described in section 6.1. The testing libraries described in chapter 5.1.3 had also to be installed.

6.3 Engino platform development

In this section, the implementation of the plan discussed in chapter 5.3 is presented.

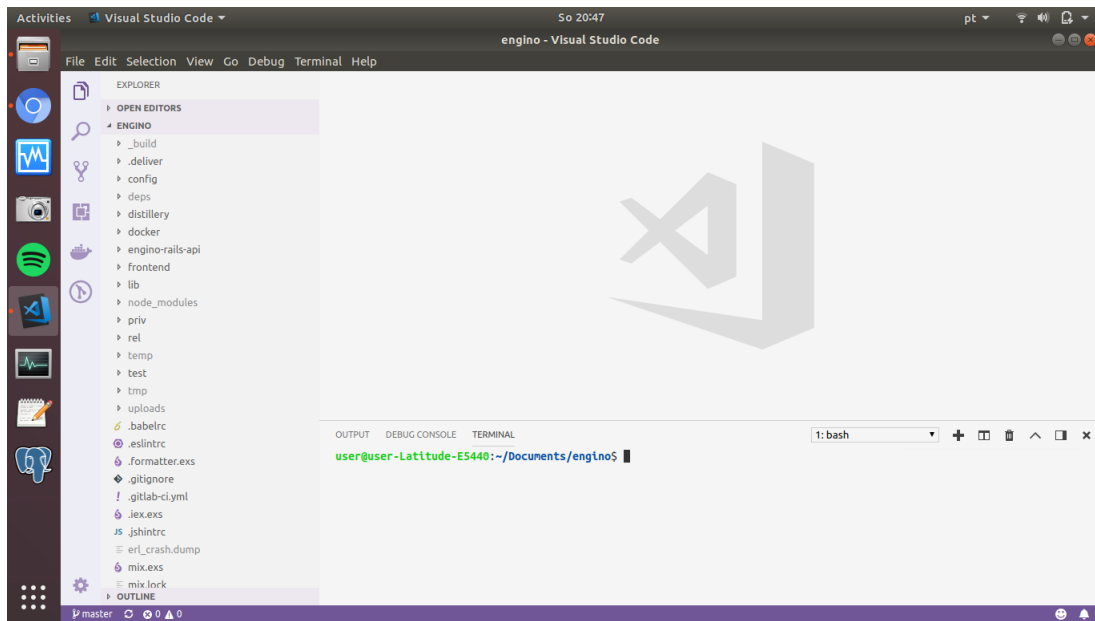


Figure 12 – View of the development environment.

6.3.1 Basic project documentation

A step-by-step guide of how to install the Oracle client was implemented in the README file of the Gitlab repository. After this installation, it's possible to run the rails server in the development environment. Figure 13 shows the view of the guide.

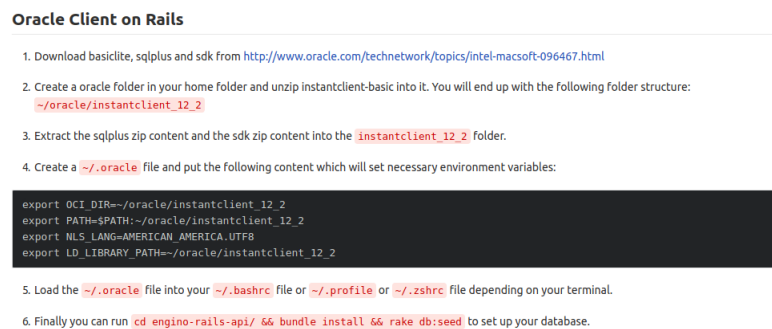


Figure 13 – Step-by-step guide to install the Oracle client.

For documenting the deployment path and commands, the Wiki section of Gitlab was used. The final document is displayed in appendix A.

6.3.2 Restructuring of the frontend file structure

First, two new folders were created in `frontend/app/`: `screens` and `containers`. Then, the screen files, `Home.jsx` and `View.jsx`, were moved to the folder `screens`, along with the `AppRouter.jsx` file, which is responsible for defining the frontend routes. After the application was tested and it was verified that everything still worked, the files from the `containers` folder located in `components/containers` were moved to the new location

and all file references were updated. Next, the application was tested again to verify its functionality.

Then, the files from the presentational folder were grouped by function and moved inside the components folder. At last, the references of these files were updated and the application was tested again.

Figure 14 shows a comparison between the old file structure and the new one.



Figure 14 – Comparison of the old frontend file structure (left) and the new one (right).

An `index.js` file was also created in each folder, so that multiple files can be imported in a single declaration by using ES6 import syntax.

6.3.3 Restructuring of the API schema

The big monolith file that contained the GraphQL schema was divided into smaller files by grouping the resource types of same domain. New folders were created to store the schema of other applications inside the Engino platform, such as Shopino and SAS.

In figure 15, it's displayed the new GraphQL schema structure.

6.3.4 Modification of the communication protocol

In the frontend, the library Apollo Client was configured to use web sockets and point to the backend endpoint.

In the backend, a user socket module to manage the connection of the users to the web socket had to be created. Furthermore, an endpoint was created to receive the web socket connections and point them to the user socket module.

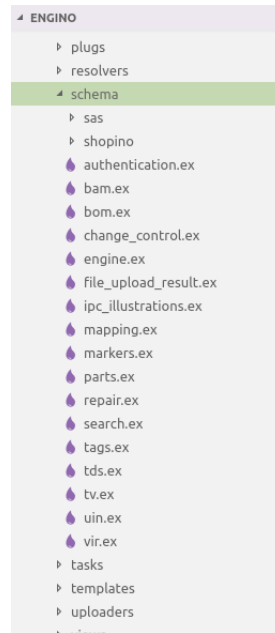


Figure 15 – New GraphQL schema structure.

The user socket module is presented in appendix B. Only connections with a valid token in the header are accepted. After a token has been successfully decoded into a user name, the user name is added to the Absinthe context, in order for the resolvers to perform operations with this information, such as query scoping and verifying data access permission.

Once the user connects to the web socket, the backend library Absinthe takes care of receiving the GraphQL queries and mutations requests and redirecting them to the appropriate resolvers.

For the non authenticated users, a special schema that uses HTTP as the communication protocol was created. Only the login mutation is available in this schema.

6.3.5 User authentication system

A mutation to login and a query to fetch the user from a token was implemented in the backend. The login mutation is resolved in a function that checks through the company LDAP system if the user credentials are valid and returns an OK message in positive case. Then, the token is created with one month of validity, by encoding the user name through the library Guardian, and sent back to the frontend. The query to fetch the user from the token also uses the library Guardian, but now to decode the token into the user name.

In the frontend, a login screen was implemented to collect the user credentials and send them to the backend through the login mutation. If the login is successful, the token received is saved in the browser cookies. The next time the user accesses the application, the authentication cookies are verified by an authentication HOC created and sent to the

backend. In case the token is valid, the query returns the user name of the token and the HOC permits the user to have access to the application. In case there are no authentication cookies, the user is redirected to the login screen.

Futhermore, another GraphQL schema was created for the queries and mutations that don't need authentication, such as the login mutation.

In appendix C, the authentication HOC created is shown. With it, every application inside the Engino platform can have the authentication system setup by wrapping the root component of the application in the HOC.

6.3.6 Permission system

For the permission system, the tables mentioned in chapter 5.1.2 were created. In the backend, the queries and mutations to read, create, update and delete access groups and its relationships were implemented. Modules for resolving the requests and for persisting the data were also created.

In the frontend, a screen to add and manage the permissions was implemented in the Admin application. For verifying if the session user has access permission for restricted pages, HOCs were created using the library *redux-auth-wrapper*. This package connects the session state from *redux* and provides it as prop for a condition function, that has to return a boolean value representing if the user has permission or not to access that page or component.

The permission screen was divided in half, so that in the left side the user can add and manage the access groups and, in the right part, users can be added or removed from the selected groups. Figure 16 shows the permission screen.

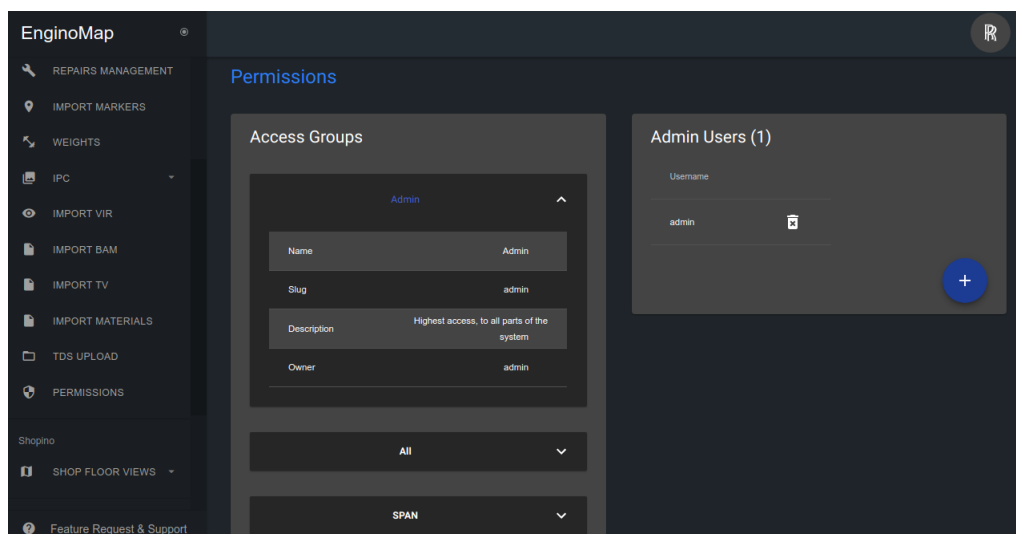


Figure 16 – Permission screen.

Table 1 – Test cases for the login resolver

| Function | login/2 | login/2 |
|------------------|---|--|
| Test Scenario | with valid credentials returns username and token | with invalid credentials returns error |
| Test Steps | pass username and password as argument | pass username and password as argument |
| Test data | username = "tester" password = "tester" | username = "" password = "" |
| Expected Results | {:ok, %{username: "tester", token: token}} | {:error, _} |

6.4 Engine platform testing

In this section, the implementation of the test plan discussed in chapter 5.2.2.3 is presented.

6.4.1 Unit tests

In this subsection, the implemented unit tests are presented.

6.4.1.1 Testing authentication API resolver in the back end

Tests were created for the authentication API resolver, which resolves user authentication queries and CRUD queries of access groups and user access groups.

In table 1 is displayed the test cases of the login resolver. During the application flow in production, the function `login/2` receives the username and password from the request and checks with the LDAP server if the credentials are valid. In the test and development environments, the credentials are tested against predefined credentials.

6.4.1.2 Testing login screen in the frontend

For the login screen in the frontend, four unit tests were done:

- Renders username and password inputs;
- Renders a submit button;
- Updates username input on change;
- Updates password input on change.

6.4.1.3 Testing permission persistence module in the back end

The permission persistence module in the back end was tested indirectly by testing the authentication resolver, since it calls the persistence module functions. Furthermore, the remaining functions that are not used by the authentication resolver were also tested.

6.4.1.4 Testing scoping of views belonging to access groups in the back end

Figure 17 shows the test created in the back end for the scoping of views belonging to the user access groups. In it, a group is created and associated a user to it. Then, in line 19 is created an association between engine view and access group. In lines 17 and 21 orphan views are created, so that there are more than one engine view in the database. Finally, in line 23 the function that applies the scope is called and it's asserted that it only returned the user associated engine view.

```
engine_view_test.exs x
You, 3 days ago | 1 author (You)
1 defmodule Engino.EngineViewTest do
2   use Engino.DataCase
3
4   alias Engino.Mapping.EngineView
5
6   describe "Engine View" do
7
8     test "scope/3 returns only the views from the user groups" do
9       access_group = insert(:access_group)
10      username = "u123456"
11      insert(
12        :user_access_group,
13        username: username,
14        access_group: nil,
15        access_group_id: access_group.id
16      )
17      insert(:engine_view)
18      engine_view = insert(:engine_view)
19      Engino.Mapping.update_engine_view_access_groups(engine_view.id, [access_group.id])
20
21      insert(:engine_view)
22
23      res = EngineView.scope(EngineView, nil, %{current_user: %{username: username}})
24      |> Engino.Repo.all
25
26      assert res == [engine_view]
27    end
28  end
29 end
30
```

Figure 17 – Test to check the view scope by the user access groups.

6.4.1.5 Testing the tiles importer

In order to test the tiles importer without overwriting the already uploaded files, a function was created in the importer module to check the environment of execution and choose accordingly the path to create new tiles.

Figure 18 shows the map importer test created. It starts by uploading an image and sending the map attributes as argument of the map importer function. Then, it is verified if the map was correctly created, by checking if the name of the map returned matches the sent one. After that, it is checked if the tiles folder of that map was indeed created and its subfolders for each zoom level. At last, every zoom folder is iterated to check if the correct amount of tiles were created for each level.

Once the test finishes, the temporary folder where the tiles were created is deleted.

```

6 describe "Map Importer" do
7   @map_attrs = {name: "Map", max_zoom_level: 3}
8   @maps_path = MapImporter.maps_path
9
10  setup do
11    on_exit fn ->
12      @maps_path
13      |> File.rm_rf!
14    end
15  end
16
17  test "import/1 with image uploaded creates map and tiles" do
18    upload_key = upload_image()
19    attrs = {upload_key: upload_key}
20    |> Map.merge(@map_attrs)
21
22    assert {ok: %Mapping.Map{} = map, success: true} = MapImporter.import(attrs)
23    assert map.name == @map_attrs.name
24
25    tiles_path = "#{@maps_path}/#{map.id}"
26    assert File.dir?(tiles_path)
27
28    level_zoom_folders = File.ls!(tiles_path)
29    assert length(level_zoom_folders) == @map_attrs.max_zoom_level
30
31    level_zoom_folders
32    |> Enum.each(fn zoom_level ->
33      tiles = File.ls!( "#{tiles_path}/#{zoom_level}" )
34      index = String.to_integer(zoom_level) - 1
35
36      assert length(tiles) == :math.pow(2, 4 + (2 * index))
37    end)
38  end

```

Figure 18 – Map importer test.

Table 2 – Test case for the userIsAdmin HOC

| Component | userIsAdmin HOC |
|------------------|--|
| Test Scenario | does not render admin screen if user is not admin |
| Test Steps | <ol style="list-style-type: none"> 1. mock user session state 2. mock redux store 3. pass mocked store as argument to component 4. mount admin screen wrapped by userIsAdmin HOC |
| Test data | <pre> accessGroupSlugs = ["verifier"] currentUser = { accessGroupSlugs } store = { session: { currentUser } } </pre> |
| Expected Results | renderedComponent != adminScreen |

6.4.1.6 Testing access permission to admin areas

Table 2 shows the test case for the HOC `userIsAdmin`, which is responsible for blocking the access of non admin users to restricted screens. Other similar test was also done, in order to check if admin users have access to admin screens.

6.4.2 Integration tests

In this subsection, the implemented integration tests are presented.

6.4.2.1 Testing the integration between the authentication modules in the front and back end

In the frontend, the login screen is the module responsible for sending the user credentials to the backend. In order to test the integration between this module and the backend, a mock of the mutation response was created. Then, after setting the username and password to match the ones configured in the response mock, it was checked if when the submit button is clicked, the function in the login screen to handle the successful response from the backend is called.

In the backend, the test was done by sending a GraphQL mutation to the login resolver endpoint, in order to simulate the request coming from the frontend. Then, it is checked if the response contains the sent username and a valid token. Figure 19 shows the test done for the login mutation in the backend.



```
69 test(
70   "login mutation returns username and token",
71   %{conn: _conn, username: _username}
72 ) do
73   conn = Phoenix.ConnTest.build_conn()
74   user_credentials = %{
75     username: "tester",
76     password: "tester"
77   }
78
79   query = """
80   mutation login {
81     login(
82       username: "#{user_credentials.username}",
83       password: "#{user_credentials.password}"
84     ) {
85       username
86       token
87     }
88   }
89   """
90   You, 7 days ago • Create authentication schema test
91   res = conn
92     |> post(@unauth_endpoint, %{query: query, operationName: "login"})
93     |> json_response(200)
94     |> Map.get("data")
95     |> Map.get("login")
96
97   assert res["username"] == user_credentials.username
98   assert String.length(res["token"]) > 10
99 end
100 end
```

Figure 19 – Integration test for the login mutation request in the backend.

6.4.3 End-to-end tests

In this subsection, the implemented end-to-end tests are presented.

6.4.3.1 Testing the authentication flows

The following end-to-end tests were created to test the authentication flows:

1. User logs in with valid credentials and logs out;
2. Logging in with invalid credentials returns error message;
3. Automatically logs in with valid token;
4. Logging in with invalid token causes a redirect to the login page.

In order to test them, five reusable functions were created: *assertIsLoginPage*, *assertIsHomePage*, *signIn*, *signOut* and *assertLoginError*. Therefore, the test for each authentication flow was easily built by simply calling the necessary functions in the right order. Figure 20 shows the test implementation for the authentication flow number 1.

The image shows a code editor window titled 'JS App.test.js'. It contains two Jest test functions. The first test, 'logs in with valid credentials and logs out', is an async function that launches a browser, navigates to APP_URL, asserts the page is a login page, signs in with valid credentials, asserts the page is the home page, signs out, and asserts the page is a login page again. The second test, 'logging in with invalid credentials returns error message', is an async function that launches a browser and navigates to APP_URL. The code is as follows:

```
65 test('logs in with valid credentials and logs out', async () => {
66   const browser = await puppeteer.launch({
67     headless: false
68   });
69   const page = await browser.newPage();
70
71   page.emulate(emulateOptions);
72
73   await page.goto(APP_URL);
74
75   await assertIsLoginPage(page);
76
77   await signIn(page, { username, password });
78
79   await assertIsHomePage(page);
80
81   await signOut(page);
82
83   await assertIsLoginPage(page);
84
85   browser.close();
86 }, 16000);
87
88 test('logging in with invalid credentials returns error message', async () => {
89   const browser = await puppeteer.launch({
90     headless: false
91   });
92   const page = await browser.newPage();
93
94   page.emulate(emulateOptions);
95
96   await page.goto(APP_URL);
97
```

Figure 20 – End-to-end test for one of the authentication flows.

6.5 Shopino development

In this section, the implementation of the plan discussed in chapter 5.4 is presented.

6.5.1 Create tables

As discussed in chapter 5.1.2, three tables were created: *shop_floor_views*, *zones* and *maps*. In order to not interfere with the already uploaded tiles of the engine views, a migration was created to copy the already existing engine views to the maps table.

Then, the attributes from the table engine views were drop, except for the attribute name, and a foreign key to reference a map was added. After that, the *map_id* fields were updated with their own id, in order to reference the copied map. At last, the id sequence of the table maps was updated to the highest existing id, so that new maps are not assigned to already existing ids. Tile references to engine views were also updated to reference maps.

6.5.2 Upload and manage shop floor layouts

A screen to upload shop floor layouts was created, as well as a screen to manage the shop floor views.

In the frontend, the uploader component was reused from the already existing engine view upload screen. For the shop floor view management screen, the same user interface of the engine view management screen was reused, with fields to rename, delete or reassociate a shop floor view to a different map or access group.

In the backend, the queries and mutations were created and associated with the respective resolvers. Figure 21 displays the final view of the shop floor management screen.

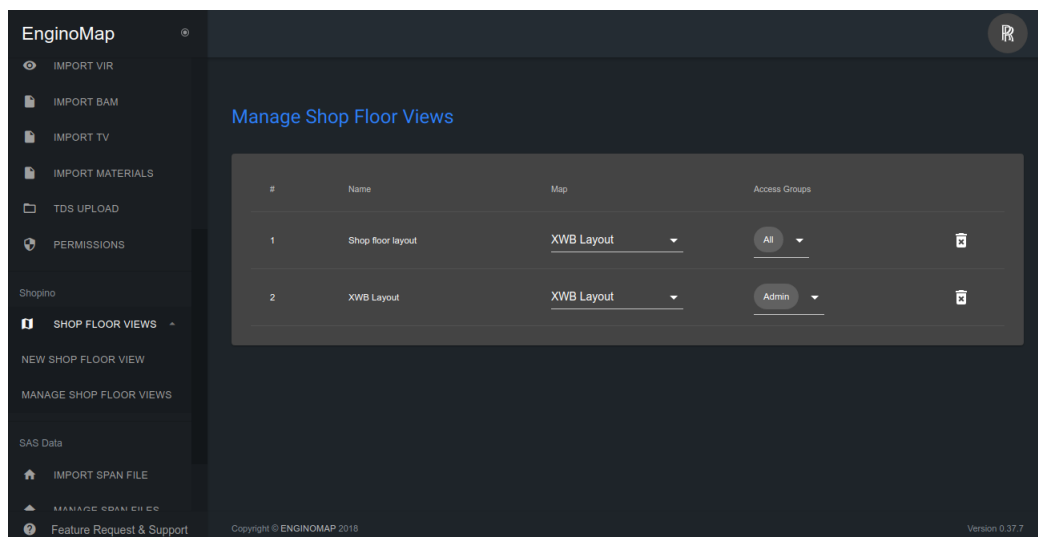


Figure 21 – Shop floor view management screen.

After the first shop floor layout was uploaded, the basic leaflet setup was done, in order to display the layout in the background of the Shopino home page.

6.5.3 Integration with BAMs and Findings database

In order to connect to the MySQL Findings database, the MySQL adapter Mariaex was installed in the backend. Then, a new Ecto Repo was configured and the models of the Findings database tables were created. For simulating the MySQL database in development, a docker image was created and run. In the production server, a environment variable with the URL, username and password was added.

6.5.4 Create, update and delete zones

In the backend, the CRUD functions were created in the persistence module, while the necessary queries and mutations were added to the GraphQL schema.

In the frontend, a button to create a new zone on the bottom right corner of the screen was added. Once clicked, the user can draw a rectangle to delimit the shop floor area. Then, a sidebar opens, so that the user can select a shop floor area, whose options come from the Findings database, and confirm the creation of the zone.

For editing, an edit button was added near the dialog title. When click, the dialog closes and the sidebar opens, giving the options to change the shop floor area or to resize the selected zone. Included in the sidebar is a button the delete the zone.

6.5.5 Display shop floor systems data related to a shop floor area

In order to display the data coming from the shop floor systems, tabs were implemented inside a dialog. The dialog opens when the user clicks on a zone and each tab represent an integrated shop floor system. For now, there are only the BAMs and Findings tabs. Inside each tag, a table was added to display the most important information of each system. On top of the table were also added a date and time selector, so that the user can filter the data by date.

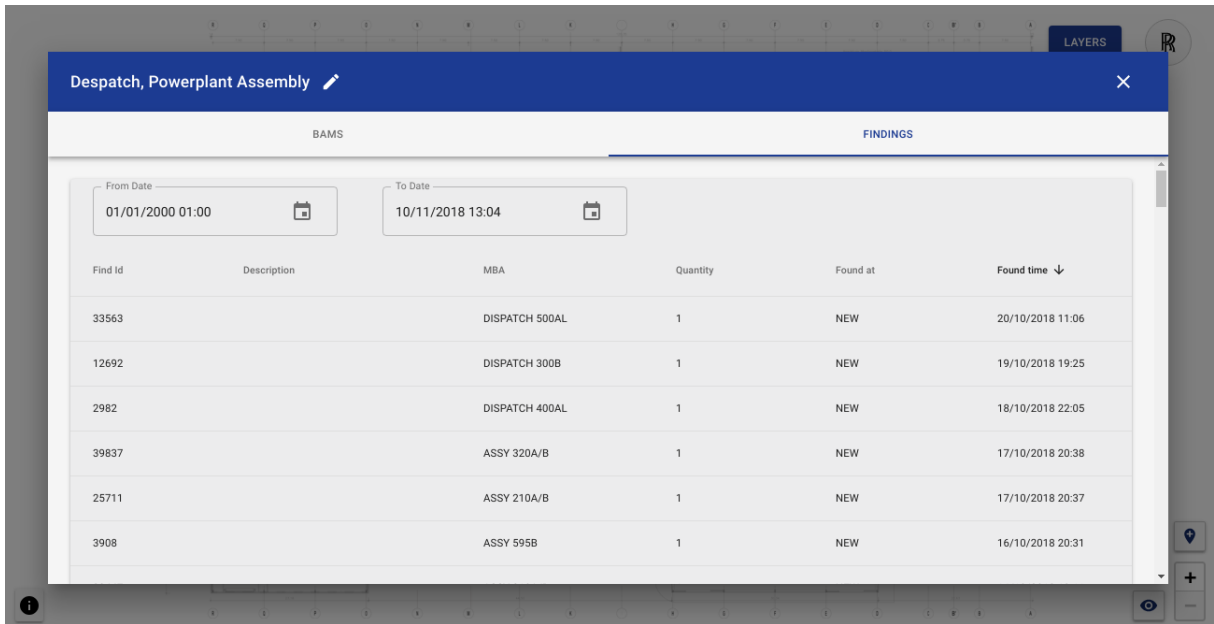
On the bottom of the table, pagination was added with the objective of increasing the speed with which the page loads. Furthermore, filters were implemented , so that when the user clicks on a table attribute, the table is filtered by that attribute. This is done in the backend by sending a new query with the selected sorting attribute.

Figure 22 shows a view of the mentioned dialog and table.

6.5.6 Provide overlays on top of shop floor areas with different type of indicators

Two types of overlays were implemented: a heat map to display the amount of BAMs for each area in a predefined period and a indicator to display the amount of Findings for each area.

Figures 23 and 24 show the view of the implemented heat map and indicator.



| Find Id | Description | MBA | Quantity | Found at | Found time ↓ |
|---------|-------------|----------------|----------|----------|------------------|
| 33563 | | DISPATCH 500AL | 1 | NEW | 20/10/2018 11:06 |
| 12692 | | DISPATCH 300B | 1 | NEW | 19/10/2018 19:25 |
| 2982 | | DISPATCH 400AL | 1 | NEW | 18/10/2018 22:05 |
| 39837 | | ASSY 320A/B | 1 | NEW | 17/10/2018 20:38 |
| 25711 | | ASSY 210A/B | 1 | NEW | 17/10/2018 20:37 |
| 3908 | | ASSY 595B | 1 | NEW | 16/10/2018 20:31 |

Figure 22 – Dialog to display shop floor systems data.

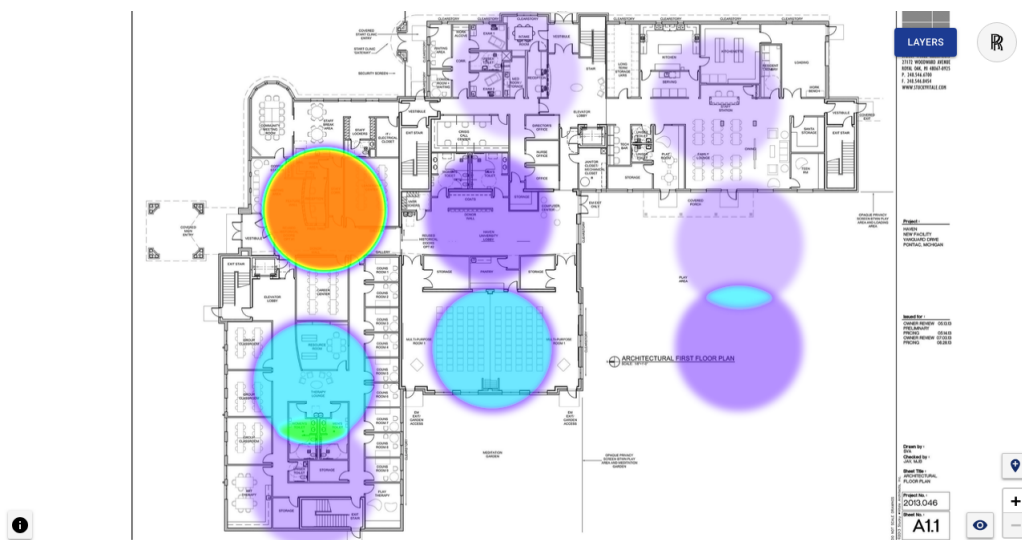


Figure 23 – Heat map displaying the amount of BAMS per shop floor area.

6.6 Shopino testing

In this section, the implementation of the plan discussed in chapter 5.2.2.4 is presented.

6.6.1 Unit tests

In this subsection, the implemented unit tests are presented.



Figure 24 – Indicator showing the amount of Findings of each shop floor area.

6.6.1.1 Testing API resolvers

Tests were created for the zone and shop floor view API resolvers, which are responsible, respectively, for resolving the CRUD queries and mutations of zones and shop floor views.

6.6.1.2 Testing shop floor views management screen

In order to test the shop floor view management screen, the queries for fetching the views had to be mocked. Then, the following test cases were created:

- Lists views and their map and access groups;
- Deletes view on click.

The first test checks if the name of every mocked shop floor view and its map and access groups are rendered in the screen. The second one tests if when the delete button is clicked, the view name is not rendered anymore.

6.6.1.3 Testing map component

Map component is responsible for rendering the tiles and zones in the screen. Since the rendering of the tiles is done by the library Leaflet, it is already tested. Therefore, only one test was created to check if it renders the zones correctly. In order to do it, zone components were passed as children to the map component and it was checked if the zone names were rendered in the screen.

6.6.1.4 Testing zone component

The zone component is responsible for rendering a rectangle in the map by calling the Marker component from Leaflet with the same size and position as it was created and the Tooltip component from Leaflet to display the name of the zone. Therefore, a test was created to check if the component passes the correct props to the Marker and Tooltip components.

6.6.2 Integration tests

In this subsection, the implemented integration tests are presented.

6.6.2.1 Testing the integration between the modules in the front and back end

For testing the integration between the modules in the front and back end, tests were created in the backend to test the GraphQL API schema. The queries to fetch a shop floor view and the view zones were chosen to be tested, as they are important for the application. For these tests, the ConnCase module of Phoenix was used, which sets up a new connection with a testing endpoint before each test is run. Before sending the connection to the test, a valid token was added to cookies of the connection, so that the tests can access queries that require authentication.

In the tests, a post request is sent to the API endpoint with the query and variables in the body. Then, it is verified if the response has a status code of 200 (OK) and if the response body contains the correct data.

In the frontend, tests were created for the map container and zones container, which are responsible, respectively, for sending the queries to fetch the shop floor view and the view zones. First the queries were mocked and then it was checked if the component had received the correct response from the query.

6.6.3 End-to-end tests

In this subsection, the implemented end-to-end tests are presented.

6.6.3.1 Testing the application flow

The application flow consists of accessing a shop floor view, creating a zone, clicking on it to display the data from the integrated systems and, then, deleting the created zone. In order to test it, small functions were created to perform each necessary task and assert that it was correctly concluded. Figure 25 displays the test implementation.

```
JS App.test.js x
154
155 test('application flow', async () => {
156   const browser = await puppeteer.launch({
157     headless: false
158   });
159   const page = await browser.newPage();
160   page.emulate(emulateOptions);
161
162   await page.goto(APP_URL);
163
164   await signIn(page, { username, password });
165   await assertIsHomePage(page);
166
167   const areaName = await addNewZone(page);
168   await assertZoneCreated(page, areaName);
169
170   await selectCreatedZone(page);
171   await assertDialogOpen(page, areaName);
172
173   await assertBamTableHasData(page);
174
175   await selectFindingsTab(page);
176   await assertFindingsTableHasData(page);
177
178   await clickEditZoneButton(page);
179
180   await deleteCreatedZone(page);
181   await assertZoneWasDeleted(page, areaName);
182
183   browser.close();
184 }, 16000);
185 });
186
```

Figure 25 – End-to-end test to check the application flow.

7 Results

The developed application solved the problems listed in chapter 4 by implementing all the specified requirements and integrating with two shop floor systems in short time. Figure 26 shows the main view of the application.



Figure 26 – Shopino main view.

In terms of quality, the best coding practices were used and tests were created to test the most important parts of the application in both the front and back end. In the next chapter, the test coverage of Shopino is evaluated.

7.1 Test Coverage

Test coverage is useful for identifying paths of the software that are not tested and improve the application adhesion to requirements. There are two types of test coverage:

- **Code coverage:** covers how much of the code is tested;
- **Case coverage:** verifies how many of the use cases are covered by test suites.

In the next subsections, both metrics are evaluated to measure the test coverage of Shopino.

7.1.1 Code coverage

Measuring code coverage requires special tools for each programming language evaluated. Therefore, the code coverage of the front and back end will be verified separately.

7.1.1.1 Frontend

Code coverage can be measured in the frontend using jest's `--coverage` and `--collectCoverageFrom` flags to specify where to look for the JavaScript files. Since many modules related to the user interface are hard or not worthy to be tested, only the most important modules will be verified. Figure 27 displays the file coverage for these modules.

| Permission HOCs | | | | | |
|----------------------------|---------|----------|---------|---------|-------------------|
| File | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s |
| All files | 67.5 | 0 | 7.14 | 70.27 | |
| index.jsx | 100 | 100 | 100 | 100 | |
| userCanAccessTds.jsx | 66.67 | 0 | 0 | 66.67 | 5 |
| userCanEditPermissions.jsx | 66.67 | 0 | 0 | 66.67 | 5 |
| userCanManageSas.jsx | 66.67 | 0 | 0 | 66.67 | 5 |
| userCanManageShopIno.jsx | 66.67 | 0 | 0 | 66.67 | 5 |
| userIsAdmin.jsx | 100 | 100 | 100 | 100 | |
| userIsVerifierOrElse.jsx | 50 | 100 | 0 | 100 | |

| Login files | | | | | |
|-------------|---------|----------|---------|---------|--------------------|
| File | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s |
| All files | 41.86 | 16.67 | 45.45 | 41.86 | |
| Login.jsx | 81.82 | 25 | 62.5 | 81.82 | 26,59,63,150 |
| app.jsx | 0 | 100 | 0 | 0 | ... 6,7,9,10,16,17 |
| main.jsx | 0 | 0 | 0 | 0 | ... ,9,17,20,21,22 |

| Shop floor view mutations | | | | | |
|---|---------|----------|---------|---------|-------------------|
| File | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s |
| All files | 100 | 100 | 100 | 100 | |
| DELETE_SHOP_FLOOR_VIEW.js | 100 | 100 | 100 | 100 | |
| IMPORT_SHOP_FLOOR_VIEW_MUTATION.jsx | 100 | 100 | 100 | 100 | |
| UPDATE_SHOP_FLOOR_VIEW.js | 100 | 100 | 100 | 100 | |
| UPDATE_SHOP_FLOOR_VIEW_ACCESS_GROUPS.js | 100 | 100 | 100 | 100 | |
| index.js | 100 | 100 | 100 | 100 | |

| Shop floor view management components | | | | | |
|---------------------------------------|---------|----------|---------|---------|--------------------|
| File | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s |
| All files | 19.7 | 19.05 | 28.57 | 18.64 | |
| ImportShopFloorView.jsx | 0 | 0 | 0 | 0 | ... 63,175,189,199 |
| ShopFloorViewList.jsx | 76.47 | 61.54 | 75 | 84.62 | 52,70 |
| index.jsx | 0 | 100 | 100 | 0 | 1,2,4,5,6,9,10,11 |

| Map and Zone components | | | | | |
|-------------------------|---------|----------|---------|---------|--------------------|
| File | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s |
| All files | 37.84 | 18.52 | 36.96 | 38.24 | |
| Map.jsx | 40.63 | 11.11 | 40.74 | 40 | ... 15,320,335,336 |
| MapButtons.jsx | 0 | 0 | 0 | 0 | ... ,89,94,100,108 |
| Zone.jsx | 68 | 50 | 54.55 | 63.64 | ... 37,138,141,142 |

Figure 27 – Frontend code coverage.

Unfortunately, jest only shows the percentage of the statements covered in each file and not the amount of relevant and missing statements. Therefore, in order to estimate the code coverage of the frontend, a simple average of the total percentage of statements

coverage of each module i will be used:

$$\begin{aligned} \text{CODE_COVERAGE} &= \frac{\sum_{i=1}^n \text{TOTAL_PERCENTAGE_STMTS}_i}{n} = \\ &= \frac{67,5 + 41,86 + 100 + 19,7 + 37,84}{5} = 53,4\% \end{aligned} \quad (7.1)$$

7.1.1.2 Backend

In the backend, the library ExCoveralls provides test coverage statistics for Elixir files. Figure 28 displays the file coverage for the most important modules created or modified during the development of Shopino.

| COV | FILE | LINES | RELEVANT | MISSED |
|--------|--|-------|----------|--------|
| 0.0% | lib/shopino/findings/bookdata.ex | 60 | 7 | 7 |
| 0.0% | lib/shopino/findings/finding.ex | 51 | 5 | 5 |
| 0.0% | lib/shopino/findings/findings.ex | 77 | 25 | 25 |
| 0.0% | lib/shopino/findings/valid_finding.ex | 13 | 1 | 1 |
| 0.0% | lib/shopino/inspections/inspection_reque | 68 | 6 | 6 |
| 0.0% | lib/shopino/inspections/inspections.ex | 67 | 17 | 17 |
| 50.0% | lib/shopino/mapping/mapping.ex | 263 | 22 | 11 |
| 83.3% | lib/shopino/mapping/shop_floor_view.ex | 52 | 6 | 1 |
| 100.0% | lib/shopino/mapping/zone.ex | 35 | 3 | 0 |
| 72.7% | lib/engino/authentication/access_group.e | 65 | 11 | 3 |
| 100.0% | lib/engino/authentication/authentication | 125 | 13 | 0 |
| 75.0% | lib/engino/authentication/user_access_gr | 22 | 4 | 1 |
| 66.7% | lib/engino/mapping/map.ex | 64 | 3 | 1 |
| 80.6% | lib/engino_web/workers/map_processor.ex | 101 | 31 | 6 |
| 100.0% | lib/engino_web/resolvers/shopino/finding | 11 | 0 | 0 |
| 100.0% | lib/engino_web/resolvers/shopino/inspect | 11 | 0 | 0 |
| 75.0% | lib/engino_web/resolvers/shopino/shop_fl | 16 | 4 | 1 |
| 66.7% | lib/engino_web/resolvers/shopino/zone_re | 38 | 6 | 2 |
| 81.8% | lib/engino_web/resolvers/authentication | 75 | 11 | 2 |
| 75.9% | lib/engino_web/importers/map_importer.ex | 168 | 54 | 13 |

Figure 28 – Backend code coverage.

The total code coverage of the Shopino backend can be evaluated by using the formula:

$$\text{CODE_COVERAGE} = 1 - \frac{\sum_{i=1}^n \text{MISSED_LINES}_i}{\sum_{i=1}^n \text{RELEVANT_LINES}_i} = 1 - \frac{102}{229} = 55,5\%$$

That's considering the first 6 files responsible for fetching data in the Findings Database, which weren't tested, because they require connection to an external database. If they are ignored, the code coverage of the backend results in:

$$\text{CODE_COVERAGE} = 1 - \frac{\sum_{i=1}^n \text{MISSED_LINES}_i}{\sum_{i=1}^n \text{RELEVANT_LINES}_i} = 1 - \frac{41}{168} = 75,6\%$$

Table 3 – Case coverage evaluation

| Use case | Covered by a test? | Score |
|---------------------------------|--------------------------------|------------|
| CRUD Zones | Yes | 1 |
| Login / Logout | Yes | 1 |
| Visualize integrated data | Yes | 1 |
| Visualize heat maps | No | 0 |
| Visualize indicators | No | 0 |
| Manage users of an access group | Yes | 1 |
| Create shop floor views | Partially, only in the backend | 0.5 |
| Manage shop floor views | Yes | 1 |
| Manage access groups | Yes | 1 |
| Manage maps | No | 0 |
| TOTAL | 65% | 6.5 |

7.1.2 Case coverage

Code coverage is a good metric to understand how many parts of the software are tested, but it doesn't give any information if the use cases were tested. For this, it needs to be measured the number of use cases covered by test suites.

Table 3 lists the Shopino use cases and if they are covered by a test or not. In the end of the table, the total case coverage is shown.

8 Considerations and Perspectives

With the launch of the first version of Shopino, it is possible to access data about the current state of production in a user friendly interface, through the integrated systems. In order to choose the right systems to integrate with and provide immediate value to the company, an effective investigation of the many shop floor systems in place was made.

For the planning of Shopino, many meetings with production managers and engineers were done with the objective of gathering the software requirements. Then, following software quality control techniques, a quality control plan was created, specifying the product goals and quality requirements of the application, as well as the reviews and tests that should be made. Furthermore, a development plan was designed to make sure the best solutions were given for each problem.

During the development of the application, coding best practices were used and tests were created to ensure the security and quality of the software. An authentication and permission system was implemented in the Engino platform, as well as the planned tests. Then, the development and test plan of Shopino were executed successfully.

8.1 Result analysis

Some results achieved with this project:

- Implementation of version 1 of Shopino using software quality control techniques, therefore increasing the quality of the software;
- Made first integration with shop floor systems available in a very short time and, thus, creating immediate value for the company;
- Improved the quality of the Engino platform by applying modern software development techniques.

The code coverage of the backend is 75,6%, ignoring the modules that require a connection to an external database, and 53,4% for the most important modules of the frontend. The frontend code coverage is not very accurate, not only because some modules have more statements than others, but also because some files that are displaying 0% of statements covered should be ignored, since they are used only for setup or to reexport the real components. Furthermore, in both evaluations, the end-to-end tests created weren't taken into account, because they were implemented using a different library, Puppeteer, and are not identified by the testing coverage tools used. Nonetheless, the code coverage

evaluations gave a reasonable estimate of the amount of tests produced, which were satisfactory and achieved the product goals set in chapter 5.2.

More important than the code coverage is the case coverage, that indicates how many use cases are being tested. According to the evaluation explained in last chapter, most use cases of Shopino were tested, as the case coverage of Shopino is 65%. This value could be higher, but the use cases that weren't tested were considered non critical for the application operation and, therefore, were ignored for testing. However, they should also be tested in the future.

In this project, because of the short time frame, it was given focus to quality control tools and techniques, but there are other areas of software quality that should be used more in depth in the future of Shopino, such as the improvement of software processes through Software Quality Assurance techniques.

8.2 Future perspectives

Shopino is a modern and flexible software for the shop floor, which enables a lot of possibilities of integration with shop floor systems and new data visualization features in the future. The following activities are in the future plans for Shopino:

- Integrate with more shop floor systems;
- Enable creation of BAMs inside Shopino;
- Add new heat maps and indicators;
- Create multiple real-time views for different teams and locations, such as management, support and office.

References

- [1] ISO 9000. *Quality Management Systems – Fundamentals and Vocabulary*. Geneva: ISO, 2015 (cit. on p. 17).
- [2] ISO 9001:2008. *Quality Management Systems – Requirements*. 2008 (cit. on p. 22).
- [3] *A Short History of Git*. English. URL: <https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git> (cit. on p. 29).
- [4] Murali Chemuturi. *Mastering software quality assurance: best practices, tools and techniques for software developers*. J.Ross Publishing, 2011 (cit. on pp. 17, 18).
- [5] M. Cielkowski, O. Laitenberger, and S Biffl. *Software reviews: The state of the practice*. IEEE Softw., 2003 (cit. on p. 23).
- [6] *Elixir*. English. URL: <https://elixir-lang.org/> (cit. on p. 28).
- [7] *GitLab*. English. URL: <https://about.gitlab.com/2018/11/08/gitlab-for-designers/> (cit. on p. 29).
- [8] *GraphQL*. English. URL: <https://facebook.github.io/graphql/June2018> (cit. on p. 29).
- [9] *Phoenix*. English. URL: <https://phoenixframework.org/> (cit. on p. 28).
- [10] *Picture of one of the shop floors in Rolls-Royce Dahlewitz*. German. URL: <http://www.maz-online.de/Lokales/Teltow-Flaeming/Rolls-Royce-feiert-Firmenjubilaeum-in-Dahlewitz> (cit. on p. 15).
- [11] *Picture of Rolls-Royce family day in Dahlewitz took on 29/06/2017*. German. URL: http://www.frank-haueis.de/arbeiten/referenzen.html?tx_sbportfolio2_items%5Bitem%5D=64&tx_sbportfolio2_items%5Baction%5D=single&tx_sbportfolio2_items%5Bcontroller%5D=Item&cHash=716e428e0847483c325c109951800c55 (cit. on p. 15).
- [12] *PostgreSQL*. English. URL: <https://www.postgresql.org/about/> (cit. on p. 28).
- [13] *Rolls-Royce Deutschland*. English. URL: <https://www.rolls-royce.com/media/press-releases/2018/28-05-2018-rr-celebrates-launch-of-new-pearl-engine-family.aspx> (cit. on p. 14).
- [14] *Rolls-Royce Deutschland*. English. URL: <https://www.rolls-royce.com/country-sites/deutschland/uberblick/rolls-royce-in-deutschland.aspx> (cit. on p. 14).
- [15] G. Gordon Schulmeyer. *Handbook of Software Quality Assurance*. Artech House, 2008 (cit. on p. 19).

-
- [16] *Trent XWB price*. English. URL: <https://www.flightglobal.com/news/articles/rolls-royce-inks-biggest-ever-sale-214880/> (cit. on p. 14).
- [17] *Trent XWB thrust*. English. URL: <https://www.rolls-royce.com/~media/Files/R/Rolls-Royce/documents/civil-aerospace-downloads/trent-xwb-infographic.pdf> (cit. on p. 14).
- [18] *Trent XWB weight*. English. URL: https://web.archive.org/web/201607251331%2010/https://www.easa.europa.eu/system/files/dfu/EASA%20E%20111%20TCDS_RR%20Trent%20XWB_issue%2003_%20201612004_1.0.pdf (cit. on p. 14).
- [19] Stefan Wagner. *Software Product Quality Control*. Springer, 2013 (cit. on pp. 18, 22, 23).
- [20] *Web Socket*. English. URL: <http://www.websocket.org/quantum.html> (cit. on p. 29).

APPENDIX A – Production Wiki

```

1  ## General Info
2
3  - Server host: `server-ajw`
4
5  - Engino path: `/engino`
6
7  - Environment variables path in production: `/engino/production/.env`
8
9  ## Elixir Application
10
11 - To start the server in staging:
12 `PATH="$PATH:/engino:/engino/ruby/bin:/engino/librsvg/bin:/engino/imagemagick/bin/:`
   ↳ /engino/oracle/instantclient_12_1/:`
   ↳ /engino/staging/engino-rails-api/gems/ruby/2.5.0/bin/" /engino/staging/bin/engino
   ↳ start`
13
14 - To start the server in production:
15 `PATH="$PATH:/engino:/engino/ruby/bin:/engino/librsvg/bin:/engino/imagemagick/bin/:`
   ↳ /engino/oracle/instantclient_12_1/:`
   ↳ /engino/production/engino-rails-api/gems/ruby/2.5.0/bin/"
   ↳ /engino/production/bin/engino start`
16
17 - To start a shell in production, like 'iex -S mix':
   ↳ `PATH="$PATH:/engino:/engino/ruby/bin:/engino/librsvg/bin/:`
   ↳ /engino/imagemagick/bin:/engino/oracle/instantclient_12_1/:`
   ↳ /engino/production/engino-rails-api/gems/ruby/2.5.0/bin/"
   ↳ /engino/production/bin/engino console`
18
19 - To connect to a running release in production: `/engino/production/bin/engino
   ↳ remote_console`
20
21 - To stop the server in production: `/engino/production/bin/engino stop`
22
23 - Locally, to test the application: `MIX_ENV=prod mix release` and then
   ↳ `_build/prod/rel/engino/bin/engino console`
24
25 ## Rails Application
26
27 **Important! To apply file modifications the server needs to be restarted**
28

```

```
29 - To start the server in production: `cd /engino/production/engino-rails-api &&`
    ↳ `PATH="$PATH:/engino:/engino/ruby/bin:/engino/librsvg/bin/:`
    ↳ `/engino/imagemagick/bin/" env $(cat ../.env | xargs) bundle exec puma -C`
    ↳ `./config/puma.rb`
30
31 - To stop the server:
32
33 1. Find the rails port by reading the environment variable `RAILS_PORT`
34 2. Find the PID of the `bundle` processes running on that port: `lsof -i`
    ↳ `:$RAILS_PORT`
35 3. Stop the `bundle` processes running on that port: `kill -9 $PID`
36
37 ## Deploying a release manually using PuTTY SCP Client
38
39 1. Download the release file from Amazon S3 bucket
40 2. Copy the release file to the external drive D in: `D:\Releases`
41 3. Open a Command Prompt and set the path of the [PuTTY SCP
    ↳ Client](https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html): `set`
    ↳ `PATH=D:\Releases;%PATH%`
42 4. Copy the release file to the server with the user number `${USER}`: `pscp`
    ↳ `D:\Releases\engino.gz ${USER}@server-ajw:/engino/staging`
43 5. Use PuTTY SSH Client to access the server and change directory to where the
    ↳ release file was copied
44 6. Stop the elixir application
45 7. Extract the release file: `tar -xzf engino.gz`
46 8. Start the elixir application and restart rails application if necessary
47
48 ## Manually running migrations
49
50 1. Open a remote console: `/engino/production/bin/engino remote_console`
51 2. Run the command: `Enum.each([Engino.Repo], &Ecto.Migrator.run(&1,`
    ↳ `Path.join([:code.priv_dir(:engino), "repo", "migrations"]), :up, all: true))`
```

APPENDIX B – User socket module

```
1 defmodule EnginoWeb.UserSocket do
2   use Phoenix.Socket
3   use Absinthe.Phoenix.Socket, schema: EnginoWeb.Schema
4   alias Engino.Guardian
5
6   transport :websocket, Phoenix.Transports.WebSocket
7
8   def connect(%{"token" => token}, socket) do
9     with {:ok, claims} <- Guardian.decode_and_verify(token, %{}),
10        {:ok, username} <- Guardian.resource_from_claims(claims)
11     do
12       {:ok, Absinthe.Phoenix.Socket.put_options(
13         socket,
14         [
15           context: %{current_user: %{username: username, token: token}}
16         ]
17       )}
18     else
19       _error -> :error
20     end
21   end
22   def connect(_, socket), do: :error
23
24   def id(_socket), do: nil
25 end
```

APPENDIX C – Authentication HOC

```
1 import React from "react";
2 import { connect } from 'react-redux';
3 import gql from 'graphql-tag';
4 import Cookies from 'universal-cookie';
5 import {
6   compose,
7   lifecycle,
8   branch,
9   renderComponent,
10 } from "recompose";
11 import { ApolloProvider } from 'react-apollo';
12
13 import newApolloClient from "shared/configs/apolloClient";
14 import * as actions from "shared/actions";
15 import PageLoading from "shared/components/PageLoading";
16
17 const CURRENT_USER_QUERY = gql`
18   query {
19     currentUser {
20       username
21       accessGroups {
22         id
23         slug
24         name
25         owner
26       }
27       ownedAccessGroups {
28         id
29         slug
30         name
31       }
32     }
33   }
34 `;
35
36 const mapDispatchToProps = dispatch => ({
37   createApolloClientAction: apolloClient => dispatch(
38     actions.createApolloClient(apolloClient)
39   ),
40   setCurrentUserAction: currentUser => dispatch(
41     actions.setCurrentUser(currentUser)
42   ),
43 });
```

```
44
45 export const connectApolloClient = connect(
46   ({ session: { apolloClient, currentUser } }) => ({
47     apolloClient,
48     currentUser,
49   }),
50   mapDispatchToProps,
51 );
52
53 const fetchCurrentUser = lifecycle({
54   componentDidMount() {
55     const cookies = new Cookies();
56     const token = cookies.get("authToken");
57
58     if (token) {
59       const apolloClient = newApolloClient(token);
60       this.props.createApolloClientAction(apolloClient);
61     } else {
62       window.location = `${window.location.origin}/login`;
63     }
64   },
65
66   componentWillReceiveProps({ setCurrentUserAction, apolloClient }) {
67     if (!this.props.apolloClient && apolloClient) {
68       apolloClient.query({
69         query: CURRENT_USER_QUERY,
70       })
71         .then(({ data: { currentUser } }) => setCurrentUserAction(currentUser))
72         .catch(({ graphQLErrors, networkError }) => {
73           if (graphQLErrors || networkError) {
74             window.location = `${window.location.origin}/login`;
75           }
76         });
77     }
78   }
79 });
80
81 const renderSpinnerWhileLoading = branch(
82   ({ currentUser }) => !currentUser,
83   renderComponent(
84     ({ apolloClient }) => (
85       apolloClient
86       ? (
87         <ApolloProvider client={apolloClient}>
88           <PageLoading />
89         </ApolloProvider>
90       )
```

```
91         : <PageLoading />
92
93     )
94 )
95 );
96
97 export const withAuthentication = compose(
98     connectApolloClient,
99     fetchCurrentUser,
100    renderSpinnerWhileLoading,
101 );
102
103 export default withAuthentication;
```