

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE AUTOMAÇÃO E SISTEMAS
ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Jamal Musa Rahman Filho

Development of an Application for Supervision of Concrete Quality Control

Florianópolis
2019

Jamal Musa Rahman Filho

Development of an Application for Supervision of Concrete Quality Control

Relatório submetido à Universidade Federal de Santa Catarina como requisito para a aprovação na disciplina DAS 5511: Projeto de Fim de Curso do curso de Graduação em Engenharia de Controle e Automação.
Orientador(a): Prof. Felipe Gomes de Oliveira Cabral

Florianópolis
2019

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Musa Rahman Filho, Jamal

Development of an Application for Supervision of
Concrete Quality Control / Jamal Musa Rahman Filho ;
orientador, Felipe Gomes de Oliveira Cabral, 2019.
86 p.

Trabalho de Conclusão de Curso (graduação) -
Universidade Federal de Santa Catarina, Centro Tecnológico,
Graduação em Engenharia de Controle e Automação,
Florianópolis, 2019.

Inclui referências.

1. Engenharia de Controle e Automação. 2. Web
development. 3. Quality control. 4. Phoenix and elixir. 5.
Functional programming. I. de Oliveira Cabral, Felipe
Gomes. II. Universidade Federal de Santa Catarina.
Graduação em Engenharia de Controle e Automação. III. Título.

Jamal Musa Rahman Filho

Development of an Application for Supervision of Concrete Quality Control

Esta monografia foi julgada no contexto da disciplina DAS5511: Projeto de Fim de Curso e aprovada na sua forma final pelo Curso de Engenharia de Controle e Automação.

Florianópolis, 27 de novembro de 2019

Banca Examinadora:

Rafael Jung

Orientador na Empresa
Jungsoft

Felipe Gomes de Oliveira Cabral

Orientador no Curso
Universidade Federal de Santa Catarina

Fabio Baldissera

Avaliador
Universidade Federal de Santa Catarina

Mateus Giovanni Ewert Bonet

Debatedor
Universidade Federal de Santa Catarina

Murilo Peruch Nunes

Debatedor
Universidade Federal de Santa Catarina

Acknowledgements

The principal acknowledgement ought to be directed to my parents Jamal and Rose, for they have provided me the true essence of love and support through my life path. Education and knowledge were exposed by them as the key to achieve anything, as well as kindness, respect and discipline to thrive in this modern world. Agradeço imensamente por tudo que me foi ensinado, sem vocês eu seria apenas um composto orgânico complexo de probabilidade existencial ínfima, amo vocês da maneira mais pura e real possível. Muito obrigado por tudo. Also to my brother Mussa, with whom I've grown up and evolved together. Plenty of good moments (and more to come) and a great partnership are carved in my memory, 'preciated it my man.

To my friend and engineer colleague Victor Lopes (aka Surfista) for the opportunity to work in such a project that represents the essence of the ECA program. Many thanks to the backend engineer Rafael Scheffer at Jungsoft for all the Elixir support throughout the development of this project, without him the path to completion of the application would be much more painful.

A special thanks to my life brothers Gabriel, Murilo and João that the engineering program has provided, you were the spirit boost that I needed to finish this phase of my life, I heartily thank you for everything. I would also like to extend this thanks to my friend and roommate Francisco for the three and a half years living together, thank you for all the great conversations and teachings primão.

To the many friends I've made in UFSC and LVA, the latter in which I've achieved three years of personal development alongside great people and scientists, thank you all. Wouldn't forget my brothers from the ODM higher mentality group, Criciúma and Floripa would be far less interesting without your vast group knowledge. Só agradece família.

Resumo

A tecnologia alcançou um nível de evolução incessável, permitindo que uma grande quantidade de dados possa ser compartilhada e facilitando uma cooperação global, o que incentiva o desenvolvimento de projetos das mais variadas áreas. Com uma estrutura *online* solidificada, projetos de serviços e produtos baseados na rede *web* começam a surgir e dominar o mercado. A Jungsoft é uma empresa que desenvolve softwares e possui um projeto de automação para centrais de concreto chamado Kartrak, no qual o autor pôde cooperar e aumentar o conhecimento na área de desenvolvimento web. A plataforma Kartrak não possui uma área para supervisionar e controlar os estados iniciais do controle de qualidade do concreto e o presente projeto busca solucionar tal problema.

Uma aplicação web moderna chamada Kartrak Laboratory foi proposta para atacar esse problema de supervisão. Devido ao curto espaço de tempo fornecido para desenvolver o projeto e pelo fato do autor não ter experiência prévia na área de programação funcional e desenvolvimento web, o programa foi construído em cima da plataforma de automação Kartrak. Uma vantagem é que a manutenção do aplicativo será facilitada devido à mesma estrutura estar sendo utilizada. Metodologias ágeis e baseadas em teste foram utilizadas de modo a obter um melhor gerenciamento do tempo. Para atingir um alto nível de qualidade, técnicas de controle de software foram aplicadas durante o desenvolvimento do projeto.

As principais funções *backend* do software, isto é, funcionamento do servidor, foram implementadas, obtendo assim uma aplicação funcional para controlar e registrar todas as etapas do ciclo de vida do corpo de prova. Para garantir um nível de confiança e qualidade, vários testes unitários e de ponta-a-ponta foram desenvolvidos e implementados.

Palavras-chave: desenvolvimento web, controle de qualidade, desenvolvimento baseado em testes, elixir, phoenix, programação funcional.

Abstract

Technology has reached a non-stop pace of evolution, allowing data sharing and global cooperation to boost the development of projects from the most vast areas. With a solid online structure, web-based services and products are beginning to emerge and conquer the market. Jungsoft is a company that develops softwares and has a project for the automation of concrete batching plants named Kartrak, in which the author had the opportunity to cooperate and learn. The Kartrak platform doesn't have a supervision feature to control the early stages of concrete quality and this project targets that problem.

Kartrak Laboratory, a modern web-application, was proposed to counteract that problem. Due to short deadline and no previous experience in functional programming and web-development, it was built on top of the already existing Automation platform. An advantage is that maintainability will be enforced since the same structure will be used. Agile and test-driven-development methodologies were pursued in order to have a better management of time. To attain a high level of quality, software quality assurance and control techniques were applied during the application development.

The main backend functionalities of the application's server-side were implemented, thus achieving a working feature to control and register the specimen life cycle. To ascertain a level of confidence and quality, several unit tests and an end-to-end test were designed and implemented.

Keywords: web-development, quality control, test-driven-development, elixir, phoenix, functional programming.

List of Figures

Figure 1 – Real-time storage overview.	26
Figure 2 – Plant supervisory layout matches the real batching plant.	26
Figure 3 – PDCA cycle.	32
Figure 4 – Quality assurance components.	32
Figure 5 – Quality control loop.	33
Figure 6 – The test-driven-development step cycle: design a failing test, implement minimum code to pass the test and improve the design via refactoring.	37
Figure 7 – Slump test procedure.	41
Figure 8 – Kartrak web platform screens.	44
Figure 9 – A common location flow of the specimen, from creation to rupture.	45
Figure 10 – Kartrak <i>Laboratory</i> modal screens.	46
Figure 11 – Rupture modal.	47
Figure 12 – Simplified <i>Kartrak Laboratory</i> database diagram and its cardinalities.	54
Figure 13 – End-to-end queries, from the GraphiQL interface, used for the integra- tion testing.	78
Figure 14 – Screen holding general information about shipments and collections.	87
Figure 15 – Screen containing information about the total percentage of shipments with specimens moulded.	88
Figure 16 – Screen with information about ruptures scheduling.	88

Listings

5.1	Specimen migration file.	55
5.2	Collection migration file.	55
5.3	<i>Kartrak Laboratory enum types.</i>	56
5.4	List specimens.	57
5.5	Get specimen by ID.	57
5.6	Specimen creation.	57
5.7	Specimen update.	57
5.8	Specimen context file.	59
5.9	Collection context file.	59
5.10	Specimen schema file.	60
5.11	Collection schema file.	61
5.12	Simplified GraphiQL endpoint implementation.	62
5.13	Specimen GraphQL schema: scaffold.	63
5.14	(Simplified) Specimen GraphQL schema: objects definition.	63
5.15	Query and mutation definition.	64
5.16	Resolver implementation.	64
5.17	All specimens query.	65
5.18	Create specimen mutation.	65
5.19	GraphQL query interpretation of 5.17.	66
5.20	Response data from query 5.17.	66
5.21	Corporation default settings update mutation.	67
5.22	Specimen settings creation.	68
5.23	Specimen settings update transaction.	69
5.24	Specimen batch creation based on the default settings of a corporation.	70
5.25	Recursive specimen creation.	70
5.26	Function to update a specimen batch.	71
5.27	Function to register a collection batch.	72
5.28	Function to update a collection.	72
5.29	Collection resolver private fuction to update the specimen situation.	72
5.30	Function to register a rupture.	73
5.31	Collection verifier function.	73
5.32	f_{ck} calculation function.	73
5.33	Asserting that creating a rupture with invalid data generates an error.	74
5.34	Rupture creation unit test to assert that valid data creates a rupture.	75
5.35	Possible testi to assess the integration between modules.	76
6.1	Test coverage result.	77

List of Tables

Table 1 – Code Management terminology. 36

List of abbreviations and acronyms

ABNT	Associação Brasileira de Normas Técnicas
API	Application Programming Interface
CD	Continuous Delivery
CI	Continuous Integration
CRUD	Create, Read, Update and Delete
DRY	Don't Repeat Yourself
ECA	Engenharia de Controle e Automação
ERP	Enterprise Resource Planning
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
NBR	Norma Brasileira
PDCA	Plan, Do, Check and Act
REST	Representational state transfer
SQA	Software Quality Assurance
SQC	Software Quality Control
SQL	Structured Query Language
TDD	Test Driven Development
UI/UX	User Interface/User Experience
UML	Unified Modelling Language
URL	Uniform Resource Locator

Contents

	Listings	13
1	INTRODUCTION	23
1.1	Motivation	23
1.1.1	General Objectives	23
1.1.2	Specific Objectives	24
1.2	Monograph's Structure	24
2	JUNGSOFT & KARTRAK	25
2.1	Jungsoft	25
2.2	Kartrak	25
3	THEORETICAL BACKGROUND REVIEW	27
3.1	Requirements Engineering	27
3.2	System Modelling	28
3.3	Quality Management	29
3.3.1	Software Quality	29
3.3.2	Software Quality Assurance & Control	31
3.4	Software Testing	33
3.5	Configuration Management	35
3.6	Methodology	36
3.6.1	Agile software development	36
3.6.2	Test-driven development	37
3.7	Technologies	37
3.7.1	Phoenix	37
3.7.2	Elixir	38
3.7.2.1	Functional Programming	38
3.7.3	PostgreSQL	38
3.7.4	GraphQL	39
3.7.4.1	Absinthe	39
3.7.5	Git	39
3.7.5.1	GitLab	40
3.8	Quality Control of Concrete	40
3.8.1	Slump Test	41
3.8.2	Compressive Strength Test	41
3.8.3	Standardisation Review	42

3.8.3.1	NBR 12655:2015	42
3.8.3.2	NBR 5738:2015	42
3.8.3.3	NBR 5739:2019	42
4	PROJECT APPROACH	43
4.1	Current State	43
4.1.1	Kartrak's Web Platform	43
4.2	Planned Solutions	44
4.3	Software Modelling	45
4.3.1	Requirements	46
4.3.2	Database	48
4.3.3	Tech stack	48
4.4	Quality Management	49
4.5	Kartrak <i>Laboratory</i> development plan	50
4.5.1	Create, Read, Update and Delete zones	51
4.5.1.1	Default Settings	51
4.5.1.2	Specimens	51
4.5.1.3	Collection	51
4.5.1.4	Ruptures	51
4.5.2	Queries	52
4.6	Methodology	52
5	IMPLEMENTATION AND TESTING	53
5.1	Development environment	53
5.2	Database tables and model creation	53
5.2.1	Migrations	55
5.2.2	Context & Schema	56
5.3	GraphQL API	61
5.3.1	Queries & Mutations	62
5.4	CRUD Zones	67
5.4.1	Corporation and Specimen default settings	67
5.4.2	Specimens	69
5.4.3	Collections	71
5.4.4	Ruptures	72
5.5	Testing	73
5.5.1	Unit Tests	73
5.5.2	Integration End-to-end Tests	75
5.5.3	Code Coverage Statistics	76
6	DEVELOPMENT ANALYSIS AND RESULTS	77

6.1	Test Coverage	77
7	CONSIDERATIONS AND PERSPECTIVES	79
	BIBLIOGRAPHY	81
	ANNEX	85
	ANNEX A – OVERVIEW PAGES	87

1 Introduction

Technology has reached a non-stop pace of evolution as a result of global cooperation in the last years. Since the beginnings of the internet in 1969 with the introduction of ARPANET (Advanced Research Projects Agency Network), ways of sharing information escalated in an unprecedented manner. Associated with that and the decrease in computer prices, old companies rush to keep up with the ongoing evolution whereas a new wave of modern companies build their system's structure entirely with the latest technologies.

As the foundation for sharing and storing data online gets solidified, web-based services and products begin to emerge. The growing interest in web applications by software developers is mainly due to its high flexibility, ease of access and low cost IT resources. These are attractive benefits for any business size, ranging from local or regional companies, as it is easy to scale up (or down), to multinationals. The application can be easily deployed in multiple physical locations around the world with just a few clicks, providing a lower latency and better experience for customers at minimal cost.

In order to develop and deploy a working and *bug-free* software, assurances regarding software quality and security must be taken into account during the development phase. In order to do so, we follow the procedures and techniques of *Software Quality Control*.

1.1 Motivation

A concrete batching plant must follow strict regulations defined by the Brazilian National Standards Organisation (*ABNT* in portuguese-br) in order to be able to commercialise its product. The current Kartrak's web-platform offers a vast range of enterprise management functionalities, such as an integrated Enterprise Resource Planning (*ERP*) and real-time Internet of Things (IoT) system (embedded + mobile + web) to automate the concrete plant, but it lacks a thorough supervision of the technological control of concrete.

1.1.1 General Objectives

This project aims at the *Technological and Quality Control* stage, in which samples of the mixed concrete must be moulded and supervised until its compression test in accordance to Brazilian Standards (NBR in portuguese-br). The objective is to develop a high-quality application to manage the test specimens¹, from creation to rupture, and integrate with the full web-platform.

¹ Cylindrical-shaped (or cubic-shaped) concrete test samples.

1.1.2 Specific Objectives

- Development of backend functionality for an application to manage the life cycle of specimens without interfering with the *in-production* software;
- Implement unit and integration tests to ensure that different parts of the application work correctly together;
- Develop end-to-end tests addressing several use cases to assure that the specified quality requirements are met;

1.2 Monograph's Structure

This monograph is divided into seven chapters, with the present one being an introduction to the project context. A brief presentation of the Jungsoft company and the *Kartrak Automation* web-platform project is given in Chapter 2. In Chapter 3 the most fundamental concepts to understand the project development are explained. A development plan is exposed and the requirements are raised in Chapter 4. The code implementations are shown and described in Chapter 5. The results are exposed in Chapter 6. Lastly, considerations and perspectives are discussed in Chapter 7.

2 Jungsoft & Kartrak

2.1 Jungsoft

Jungsoft is a web and mobile software development consultancy company founded by an ECA alumnus. With a team of 20 employees, distributed between the offices of Berlin and Brazil, Jungsoft has multiple projects in production, with their latest projects being:

- **Woodspoon:** Meal kit delivery e-commerce platform;
- **Sppyns:** Real time Crypto Investments Marketplace powered by the Ethereum blockchain;
- **Tailwind:** System for processing user feedback and generating KPIs about the company's customer experience;
- **Engino:** A tool to integrate multiple aerospace engineering systems into a fast and easy-to-access web software;
- **Kartrak:** Cloud-based web platform for enterprise management with a real-time IoT system to automate cement factories, in which the author could cooperate with the present monograph.

Jungsoft focus on providing the best experience for both the clients and end users by delivering the best solutions through the latest technologies available.

2.2 Kartrak

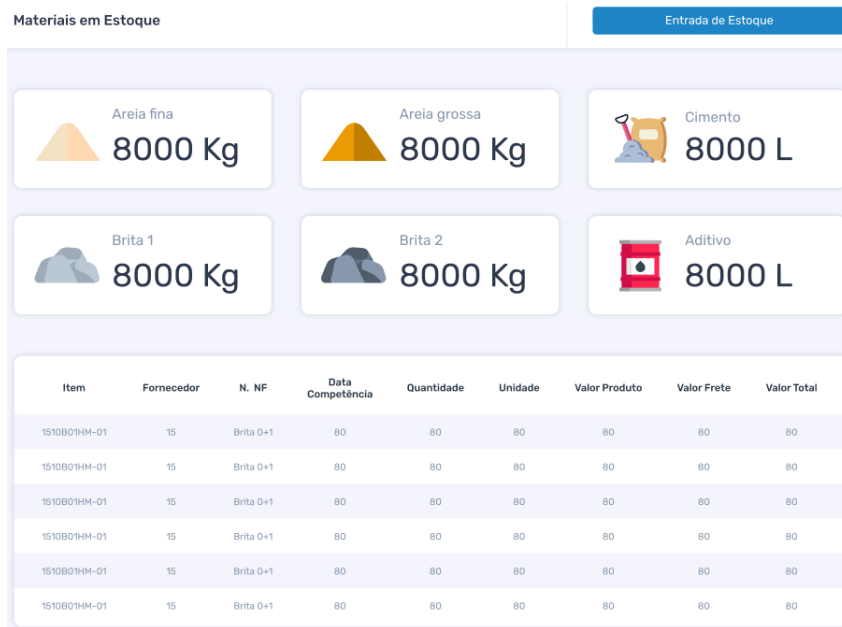
The Kartrak Automation Concrete offers a vast range of functionalities, integrating the entire batch plant supervision in a single platform accessible through any device, locally or abroad.

Its Enterprise Resource Planning and Internet of Things system offers a total management of the company resources while being able to supervise the production in real-time through the mobile app. The client can have full control of the company by means of performance indicators provided instantaneously in the platform.

Local servers cease to become a hindrance as all data is centralised and accessed through the *cloud*. The [Kartrak 2015](#) platform makes available remote shipments, automatic updates and remote support without the need of in situ visits, as well as several other

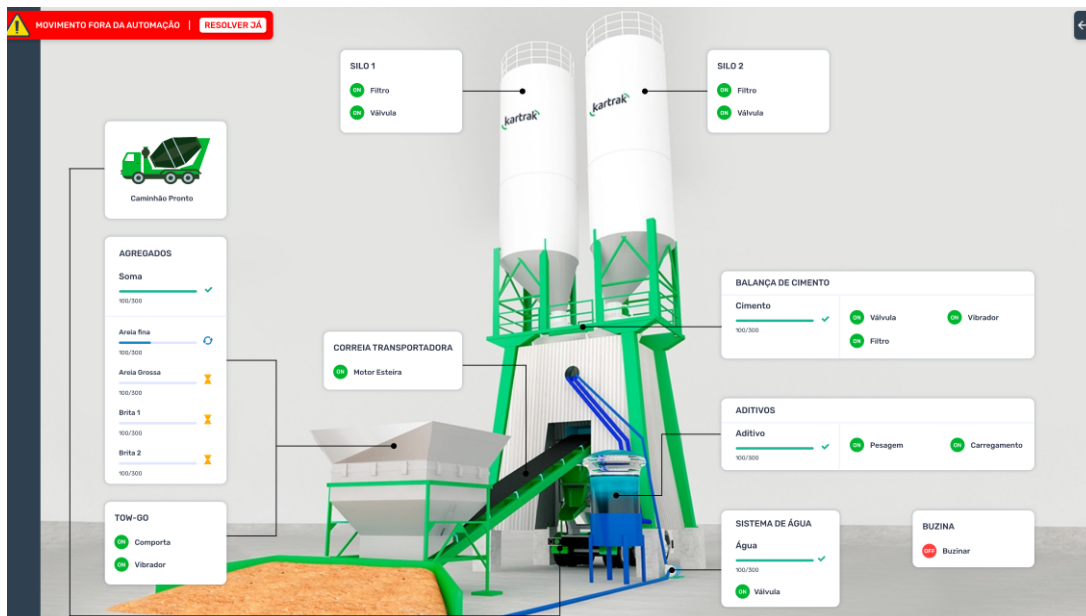
benefits. Figure 1 presents the material storage in real-time and Figure 2 shows the supervisory screen with an alert popping up informing that some operation error occurred.

Figure 1 – Real-time storage overview.



Source: Kartrak.

Figure 2 – Plant supervisory layout matches the real batching plant.



Source: Kartrak.

3 Theoretical Background Review

The purpose of this chapter is to familiarise the reader with concepts necessary to comprehend the present project in its full extent. Basic concepts of Requirements Engineering are presented in Section 3.1, which will be necessary to understand the subsequent sections. Software Quality definitions are exposed in Section 3.3 to introduce the testing techniques in the subsequent section. The methodology and technologies used are presented in Sections 3.6 and 3.7, respectively. The final section is a review of the followed standardisations defined by the ABNT.

According to [Ian 2015](#), a software is both the computer program and its associated documentation, where they may be developed for a general market or a particular customer. Presently however, these system product types are merging into a single idea, which is to build systems with a generic product as a foundation and then adapt them to suit the requirements of a customer. As an example, Enterprise Resource Planning (ERP) systems, such as the present project, are becoming more common in the market.

Essential attributes of good software are [[Ian 2015](#)]:

- **Acceptability:** it must be acceptable to the type of users for which it is designed. Meaning that it should be understandable, usable, and compatible with other systems that are in use.
- **Dependability and security:** software dependability includes a range of characteristics including reliability, security, and safety. Dependable software should not cause physical or economic damage in the event of system failure. A software has to be secure so that malicious users cannot access or damage the system.
- **Efficiency:** it should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, resource utilisation; in sum, seeking for an optimal management.
- **Maintainability:** it should be written in such a way that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment that endeavours to stay up-to-date with the latest technologies.

3.1 Requirements Engineering

In accordance to [IEEE 1990](#), a requirement is:

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.
3. A documented representation of a condition or capability as in 1 or 2.

Requirements are a collection of needs arising from the user and various other stakeholders, e.g., general organisation, community, government bodies and industry standards, all of which must be met. Requirements must show “*what*” the system must do rather than “*how*” it should be done [Wagner 2005]. That idealisation, however, has no logic in reality since “*what*” and “*how*” are subjective concepts.

Ian 2015 distinguishes requirements by referring to high-level abstraction as *user requirements* and detailed description of what the system should do as *system requirements*. It is then defined as:

- *User requirements* are statements, in a natural language with the aid of diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate. The user requirements may vary from broad statements of the system features required, to detailed, precise descriptions of the system functionality.
- *System requirements* are more detailed descriptions of the software system’s functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented. It may be part of the contract between system buyer and software developers.

3.2 System Modelling

At the modelling stage, abstract models of a system are developed with each model presenting a different view or perspective of that system. A common representation is using graphical notation based on diagram types in the Unified Modelling Language (UML) [Rumbaugh 2004].

Models are used during the requirements engineering process to help derive the detailed requirements for a system, during the design process to enlighten the developers, and after post-implementation phase for documentation purposes.

A system model is not to be understood as a complete representation of the system, it leaves out details for an easier understanding. A model is an abstraction of the system being studied rather than an alternative representation of it.

When representing a system, all information about the entity should be maintained. An abstraction deliberately simplifies a system design and chooses the most prominent characteristics. Different models can be developed to represent the system from many perspectives, for example [Rumbaugh 2004]:

1. An external perspective, where the context or environment of the system is modelled.
2. An interaction perspective, where the interactions between a system and its environment, or between the components of a system are taken into account.
3. A structural perspective, where the organisation of a system or the structure of the data processed by the system is used in the model.
4. A behavioural perspective, where the dynamic behaviour of the system and how it responds to events is modelled. This was the main model approached when designing the system.

3.3 Quality Management

Software quality management is concerned with ensuring that developed systems should meet the needs of their users, perform efficiently and reliably while also being delivered on time and within budget.

The relationship between the quality of a product and the organisation responsible for the development of that product is multidimensional. According to Schulmeyer 2007, this relationship depends upon many factors such as the business strategy and business structure of the organisation, available talent and resources needed to produce the product.

Minimising documentation and any process that doesn't encompass the development of code is a directive followed by agile methods of software engineering, where the team's synergy and communication are emphasised. Common practices in agile development, such as refactoring (restructuring internally a chunk of code without altering its external behaviour) and test-driven development (Subsection 3.6.2) are used to achieve a high code quality.

3.3.1 Software Quality

The quality of a product is well established in the manufacturing industry. Parting from the premise that products could be clearly specified and procedures set up to assert if the product conforms to those specifications, with some tolerance allowed, a definition of quality was then created. However, in the software domain, the sharpness in checking against those tolerances does not exist, as stated by Ian 2015.

ISO/IEC 25010:2011 2011 defines the following eight product qualities:

- **Functional Suitability:** measures if a set of functions covers and facilitates the accomplishment of specified tasks and objectives, besides providing the correct results with the needed degree of precision.
- **Performance Efficiency:** measures the response and processing times and the amounts/types of resources used by a product system when performing its functions.
- **Compatibility:** a product can perform its functions efficiently while sharing a common environment with other products and exchange/use information in a cooperative manner.
- **Usability:** degree to which a product can be easily used and understood by people with wide range of personal characteristics.
- **Reliability:** a product system can be accessed and operated despite ongoing faults and can recover from them.
- **Security:** the ability of a system, product or component to prevent unauthorised access to, or modification of sensitive data.
- **Maintainability:** the ease to add new functionalities to the system and modify some component with minimal impact over other components, as well as the effectiveness of test criteria that can be established.
- **Portability:** degree to which a product can replace another software product with the same purpose in the same functionality, while efficiently adapting to evolving hardware and software.

Aiming at writing a more readable code, best practices were pursued. The most common coding best practices and conventions are listed below:

- **Code consistency:** when a style or convention is chosen at the beginning of implementation, it must be adopted for the entire code.
- **Indentation and spacing:** it increases code readability and maintainability, since what is written becomes clearer.
- **Function grouping:** same function with different input parameters should be grouped together without separating blank lines, in every other case a blank line should be added.

- **Naming:** Camel case (capitalisation of the first letter of each word, except for the first one, e.g., `camelCase`) should be used for module names, and snake case (words separated by underscores, e.g., `snake_case`) should be used for module attributes, functions, macros and variable names.
- **Commenting:** should be avoided when the subject of comment is too obvious.
- **Refactoring opportunities:** action of rewriting internal organisation of code without interfering with external behaviour (e.g., refactor unnecessary multi-nested logic).
- **DRY principle:** **D**on't **R**epeat **Y**ourself, to avoid code duplication and information repetitiveness.
- **Folder organisation:** big files should be shrunk into smaller ones and separated into their correspondent folders.
- **Pitfalls:** such as lines of code used for debugging should be removed from the production code (e.g., console logging).

3.3.2 Software Quality Assurance & Control

Often confused with Quality *assurance*, quality *control* is the process of counteracting the decay in quality during software evolution. Understanding that a software must change throughout the years, a continuous quality control must be applied. Quality control is inside the scope of Quality assurance.

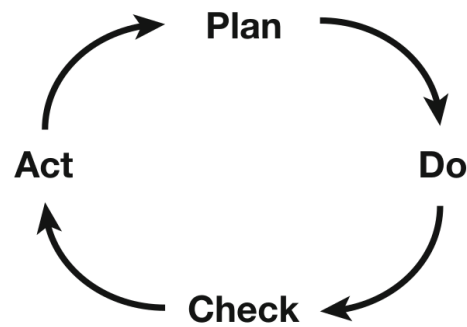
SQA is the collection of support activities and functions used to supervise and control a software project so that specific objectives are achieved with the desired level of confidence. It establishes guidelines for quality control to ensure the software withholds the integrity of its functionalities. Quality assurance is determined via consensus of the entire development team.

Continuous improvement processes is, currently, a standard approach for process-oriented quality standards such as the ISO 9001 [ISO 9001:2015 2015]. A popular model of continuous improvement is the PDCA cycle (Figure 3), it enables an organisation to ensure that its processes are adequately resourced and managed.

The name comes from stages of the cycle: **P**lan, **D**o, **C**heck and **A**ct. In the Plan phase, a plan is developed to improve the current state of the process, then the plan is implemented with sufficiently few lines of code in the Do phase. In the Check phase, the results are evaluated and, if succeeded, implemented for all processes in the Act phase.

Normally, quality assurance processes include software testing (Section 3.4), with verification and validation of code, software configuration management (Section 3.5), for

Figure 3 – PDCA cycle.



Source: Wagner 2005.

keeping track of modifications, and quality control. These components can be seen in Figure 4. It can be concluded that quality *assurance* is a managerial tool oriented toward preventing defects, whereas quality control is a corrective tool designed to detect and fix them.

Figure 4 – Quality assurance components.



Source: Lewis 2005.

As mentioned in the beginning of this subsection, all softwares tend to undergo some changes as time evolves, and that process is called software *evolution*. Hardware, user requirements, market, technologies, everything changes; and the modifications that goes along with these changes must be made with awareness of constant transformation.

Tight deadlines, low budget, inexperienced developers, poor team communication can all end up blurring that state of awareness, with direct effects on overall quality. Previous code being copied over, workaround to fix problems instead of solving them, these bad practices degrade the code while generating many bugs that reduce maintainability, reliability and performance efficiency.

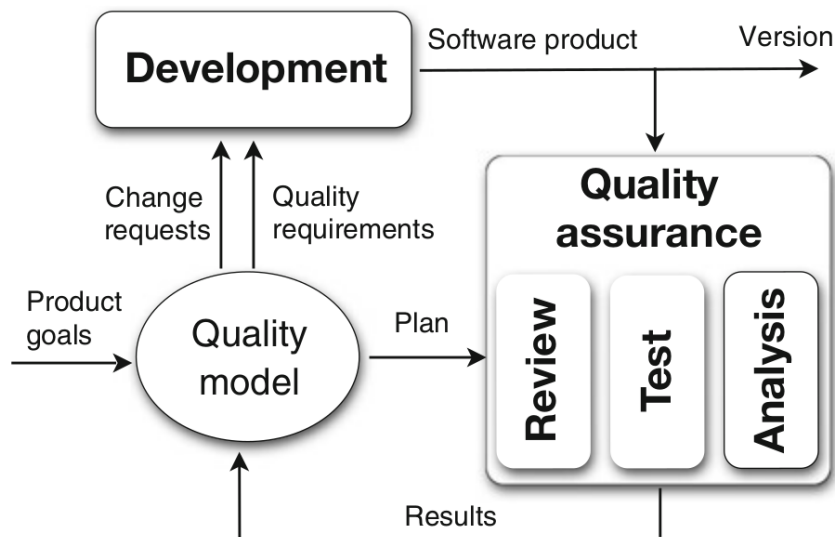
To hinder the previously alluded behaviour, continuous improvement in product

level is imperative. [Deissenboeck et al. 2008](#) proposed the loop shown in Figure 5 using the same analogy of the PDCA cycle (Figure 3) to describe the quality control process.

It begins by defining the product goals and from that, a quality model is used to specify the quality requirements for the project. Following those requirements, developers build a software product which is then passed to the SQA team, or developers themselves, which by means of reviews, tests and analyses, check if the specified quality requirements are met. Lastly, the results are introduced back into the quality model. Any deviation from the product goals is included in development as the cycle continues.

This project focused on code reviews by another developer every time new code was pushed to the repository. If both CI/CD pipelines (explained in Subsection 3.7.5.1) and peer review pass, the new code can be merged into the master code (main working version).

Figure 5 – Quality control loop.



Source: [Wagner 2005](#).

3.4 Software Testing

Software testing is a known strategy for risk management. With the objective of verifying and validating the raised functional requirements, such that code and software design requirements are met, it aids to ascertain whether all phases of the software development life cycle are fulfilled. Less complexity is needed if conducted continuously throughout the development process. The most common testing techniques are presented in the following items:

- **Black-box** (*Functional*): used to test the system's functionality by providing an

input and observing the *output*. In this type of testing, no information about the internal structure is given, as it is seen as a “black box”. A drawback is that internal errors are not perceived, even though the output might be consistent. A major advantage of black-box testing is to make sure that the system do what it is supposed to do, in other words, if the requirements are met.

- **White-box** (*Structural*): this testing technique is aimed at highlighting flawed logic paths in the internal structure of the system, focusing in the written code. On the other side, this test does not verify any specification or detects any data-sensitive errors.
- **Grey-box** (*Hybrid*): a combination of black-box and white-box testing, it provides both advantages of detecting internal errors as well as system specifications. It can also eliminate ambiguous tests, reducing effectively the number of tests to be implemented.
- **Regression**: it approaches by reapplying a set of test every time the code changes, in order to ensure that new modifications will not affect the former functionality. It continuously maintains and improves quality.
- **End-to-end**: used to verify the whole application as if it were in production state. The same environment settings should be utilised, such as databases and hardwares.

Lewis 2005 explains that the development life cycle has three test levels: *unit*, *integration* and *system* tests. At the **unit** testing phase, the system is decomposed into components, which are *classes* for object oriented languages and *modules* for other languages such as the one used in this project. These modules are the smallest building blocks that can be tested, they describe its functionality against the module specification.

In test-driven development, these unit tests are written first, then the module itself is implemented based on the test. Unit tests avoid large modifications that break the system, as it checks every change that could break an existing test. They are commonly developed as white-box tests when verifying the module’s consistency and black-box tests to encounter defects to a greater extent.

Code examinations are common in *unit* testing, a process known as **code review**. Two types are usually associated: code **walkthroughs** and inspections. In a *walkthrough*, the tester presents its code to a review team and lead an informal discussion where the main subject is finding faults in the code and not the coder itself. Code *inspections* are similar to walkthroughs, but are driven with more formalities and steps to be followed; both of them focus on the written code, not the programmer. Inspections are an efficient solution to spot early faults in the code development and are considered best practices.

With the *unit* tests sufficiently satisfied, **integration** testing with the rest of the system can be initialised. [Pfleeger e Atlee 2009](#) exposes several strategies to achieve integration, i.e., *bottom-up*, *top-down* and *sandwich* integration.

Bottom-up integration begins with tests from the lowermost component (general-purpose routines, e.g., input and output functions) and builds up to the top-level programs, until all components are included, as was approached in the development of the present project. An disadvantage is that the topmost components are usually more relevant and are the last to be implemented.

The *top-down* integration is basically the reverse of bottom-up, with the drawback of generating multiple mock data. Sandwich integration combines both bottom-up and top-down strategies with the middle layer as target. This strategy allows an early integration in the testing process.

Completing the integration phase, the **system** is practically finished and its tests can be implemented to check against specified functionality. Supposing that the customer makes the system test, it is then called as *acceptance* test.

3.5 Configuration Management

It is important to have all assets secured and available in a controlled and reliable repository. Code management offers tools and processes to manage source code and all the resulting artefacts that make up a system, called configuration items. Good source code management allows long-term developments along with quick emergency fixes [[Aiello e Sachs 2010](#)]. Non-effective source code management processes may result in major outages, unnecessary defects and wasted time doing the same work over again.

Configuration management is profitable for both individual and team projects, as one person may forget what changes have been made and several developers might work at the same time on a software system in the same place or across the world. Conforming to [Ian 2015](#), the configuration management of a software product involves the four closely related activities below:

1. **Version control:** involves keeping track of the multiple versions of system components and ensuring that changes made by different developers do not interfere with each other.
2. **System building:** process of assembling program components, data, and libraries, then compiling and linking these to create an executable system.
3. **Change management:** involves keeping track of requests for changes to delivered software from customers and developers, managing costs and impacts of making

these changes and deciding if and when the changes should be implemented.

4. **Release management:** involves preparing software for external release and keeping track of the system versions that have been released for customer use.

Common terminologies in code management can be seen in Table 1.

Table 1 – Code Management terminology.

Term	Description
Branching	The creation of a new codeline from a version in an existing codeline. The new codeline and the existing codeline may then develop independently.
Codeline	A set of versions of a software component and other configuration items on which that component depends.
Configuration (version) control	The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
Merging	The creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.
Repository	A shared database of versions of software components and meta-information about changes to these components.
Version	An instance of a configuration item that differs, in some way, from other instances of that item. Versions should always have a unique identifier.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.

3.6 Methodology

This section exposes the main methodologies used during project development in order to increase the organisational level and build a solid foundation for the application to evolve.

3.6.1 Agile software development

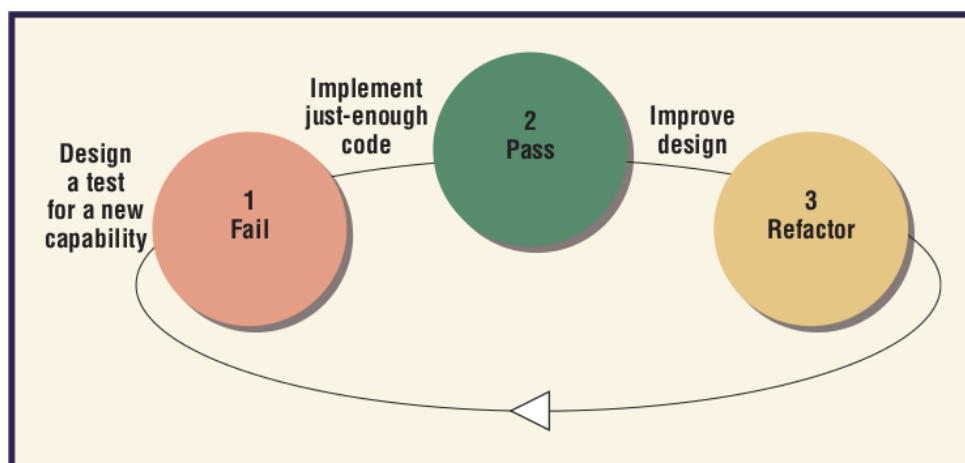
Design and implementation are the central activities in the software process when approaching the development with an agile intent. Other activities, such as requirements elicitation and testing, are incorporated into design and implementation. These agile methods allow the development team to focus on the software itself rather than on its design and documentation. They are best suited to application development where the system requirements usually change rapidly during the development process [Cha, Taylor e Kang 2019].

3.6.2 Test-driven development

Test-driven development is a discipline of design and programming where every new line of code is written in response to a test implemented just before coding. As the cycle shown in Figure 6 progresses, the program grows into form and the design evolves simultaneously.

The first step is to identify the increment of functionality that is required, which is usually small and implementable in a few lines of code. A test is written for this functionality and implemented as an automated test, meaning that the test can be executed and will report whether it has passed or failed. At the beginning of every cycle, the intention is for all tests to pass except the new one, which is “driving” the new code development. Then the functionality is implemented and the existing code is refactored to improve it.

Figure 6 – The test-driven-development step cycle: design a failing test, implement minimum code to pass the test and improve the design via refactoring.



Source: [Jeffries e Melnik 2007](#).

3.7 Technologies

This subsequent section addresses the technologies approached in the *Kartrak Laboratory* development.

3.7.1 Phoenix

Phoenix is a web framework written in Elixir with high developer productivity and high application performance, providing libraries and a standard way to build and deploy web applications into the *web*.

A strong functionality of Phoenix is Channels, a real-time layer that uses Web Sockets as a transport protocol, enabling a back and forth connection between *client*

and *server*. Vastly used in telecommunications, it leverages the Erlang Virtual Machine, allowing it to handle millions of concurrent connections [Phoenix 2016].

3.7.2 Elixir

Elixir is a modern, dynamic, functional, concurrent language that allows scalable and maintainable applications to be built [Elixir 2011]. Built on top of Erlang, it uses the Erlang VM (BEAM), known for running low-latency, distributed and fault-tolerant systems. Elixir's pragmatic syntax and built-in support for meta programming can highly increase productiveness.

Another advantage is the use of supervisors to manage execution faults. Said supervisors monitor other processes and always put the system in a working state in case an error is raised, thus creating an hierarchical process structure called supervision trees. They provide fault-tolerance and encapsulate the application's beginning and ending.

It also has an interactive mode named **IE_x**, a command-line interface where the developer can type any Elixir expression and get its result. It proved to be extremely useful in this project when debugging and testing implemented functions and expected behaviour after a request is made. Any data can be outputted and checked if its value is consistent.

3.7.2.1 Functional Programming

Functional programming is a paradigm that values immutability, first-class functions, referential transparency, and pure functions. In Elixir, all data is immutable, which means that their values are never modified, only transformed [Thomas 2018]. Any function that transforms data will return a new copy of it.

According to Halvorsen 2018, one of the most characteristic patterns of functional programming is composition. A complex code is divided into smaller and decoupled functions, then its full behaviour is recreated by chaining these functions together. That increases the code maintainability because smaller functions reduce cognitive load and are easier to work on.

3.7.3 PostgreSQL

PostgreSQL is a powerful, open source object-relational database management system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads [PostgreSQL 1996]. Custom data and index types are allowed to be defined, that is convenient as many custom states describing a specific attribute are indispensable.

A database is an organised, machine-readable collection of symbols, to be interpreted as a true account of some enterprise. A database is machine-updatable as well, and so must also be a collection of variables. A database is typically available to a community of users, with possibly varying requirements [Rothwell 1992]. It is where all persistent data pertinent to the project is stored.

3.7.4 GraphQL

GraphQL is a data query and manipulation language for APIs developed internally by Facebook in 2012 and open-sourced in 2015. It allows the data structure to be defined as the project needs, and the server returns only the data requested.

Each GraphQL module contains a declarative schema that defines the syntax for queries that the module supports, as well as the attributes that can be returned. It supports reading, writing (mutating) and subscribing to data changes (realtime updates) [GraphQL 2015].

While REST APIs require loading from multiple URLs, GraphQL APIs get all data the application needs in a single request, making apps using GraphQL fast even on slow mobile network connections.

3.7.4.1 Absinthe

Absinthe is the GraphQL toolkit for Elixir, an implementation of the GraphQL specification built to suit the language's capabilities and idiomatic style. Its functionality is understood as two broad areas: defining schemas and executing documents.

A schema defines the structure and relationships between data entities, as well as the available queries, mutations, and subscriptions. A GraphQL document can be any standard GraphQL query, mutation, or subscription; it can be analysed for its complexity and be rejected if it's unsafe/too expensive. Authentication and authorisation strategies can be integrated with an Absinthe context, which provides shared values to a given document execution, such as the current user of a given request [Absinthe 2017].

3.7.5 Git

Git is an open source version control system suited for any size of project. When multiple people are concurrently developing and modifying files from a project, Git is an excellent source code management tool that helps get track of what changes were made.

Nearly all operations are performed locally, giving Git a huge speed advantage on centralised systems that constantly have to communicate with a server somewhere. As it was built to work on the Linux kernel, it can effectively handle large repositories with high performance [Git 2005].

3.7.5.1 GitLab

GitLab is a fully integrated software development platform, not only for hosting code repositories with version control, but also for tracking bug reports and issues with the *Issue Tracker*. It has a tool for software development utilising continuous methodologies, such as: continuous integration (CI), delivery (CD) and deployment (CD) [GitLab 2014]. It can be defined a CI/CD pipeline for the pushed chunk of code to pass in order to build, test and validate the code changes before merging them into the main branch.

3.8 Quality Control of Concrete

There exists a vast range of tests regarding the quality of concrete. A few of them are:

- Tests on hardened concrete
 - Compressive strength (cylinder or cube);
 - Tensile strength;
 - Density;
 - Absorption;
 - Permeability;
 - Resistance.
- Tests on fresh concrete
 - Workability (e.g., **slump** test);
 - Air content;
 - Setting time;
 - Wet analysis.

In spite of existing several possible tests, in practice over 90% of all routine tests on concrete are concentrated on compression tests and slump tests [Day, Aldred e Hudson 2013]. These tests aim at establishing whether the concrete has attained a sufficient maturity (for stress testing or to be sent in a shipment), to ascertain that it is satisfactory for its purpose and to detect quality variations.

The primary objective of standard cured specimens is to assure that the concrete mixture delivered at the job site is able to meet the contract specifications. The potential strength and variability of the concrete can be established only by specimens made, cured, and tested under standard conditions, e.g., prolonged mixing when shipping concrete will

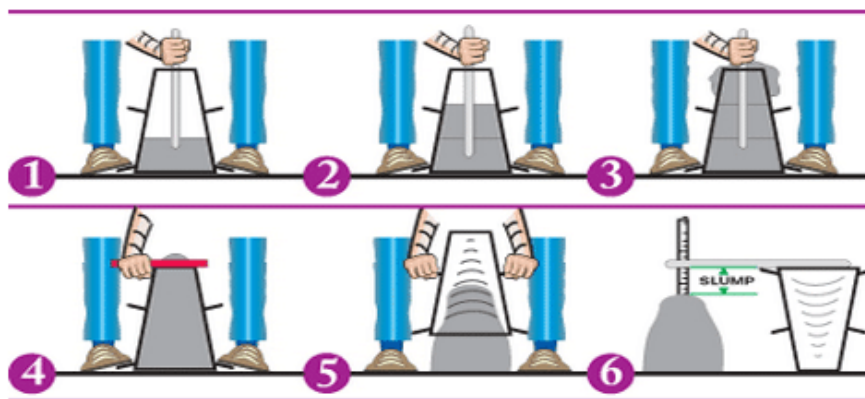
cause slump loss and result in lower workability [Committee 2011]. The two most common concrete tests are presented in the following subsections.

3.8.1 Slump Test

One of the main characteristics that influences in the workability of concrete is the *consistency*, construction site and shipping characteristics. Workability can be understood as a quality of concrete not an inherent characteristic. The slump test is a way of determining the workability (or consistency), which relates with the mixed concrete mobility and the cohesion between its components or, in other words, the water/cement ratio.

The test (Figure 7) consists of filling a conical mould with fresh concrete in 4 layers at the time, tamping each layer and slowly removing the mould. The test result, given in millimetres, is obtained by measuring the height difference between the mould and the specimen being tested.

Figure 7 – Slump test procedure.



Source: [Quora 2018].

3.8.2 Compressive Strength Test

After the process of moulding and curing of specimen samples, the batch is transported to a certified laboratory responsible for rupturing. The specimen goes through a capping procedure before being ruptured, as it must have a uniform surface when placed in the test machine.

The machine applies a constant increasing load on the specimen body, until it reaches a rupture point. The maximum load value is given in newtons (N) and used to calculate the compressive strength factor (f_{ck}) of the concrete, in megapascals (N/mm^2 or MPa). The characteristic compression resistance of the concrete is utilised in the structural calculation.

3.8.3 Standardisation Review

As commented before, a set of regulations is defined by ABNT as an effort to standardise procedures and impose a minimum degree of quality and security for whatever purpose the concrete shall be destined. This rule set was contemplated in the application's development so as to prevent future unnecessary modifications to assure the code is abiding by the standards. The main consulted regulations are listed below.

3.8.3.1 NBR 12655:2015

This Brazilian Standard covers the structural use of Portland cement concrete, a material composed by the homogeneous mixture of cement, fine and coarse aggregates, water and non-speciality grout material. Concrete can be mixed on site, ready-mixed or produced in precast plant. This Standard also establishes requirements for properties of fresh and hardened concrete and their verification; composition, preparation and control of concrete; and receipt and acceptance of the concrete.

3.8.3.2 NBR 5738:2015

This Standard prescribes the procedure for moulding and curing cylindrical and prismatic concrete test specimens. Some of them are: the height should have the doubled value of diameter; the mould must be made of steel or non-absorbent material; how many strokes are needed for each layer in the slump test, which depends on the specimen dimensions.

3.8.3.3 NBR 5739:2019

This Standard specifies the compression test method for cylindrical specimens of concrete moulded in accordance with ABNT NBR 5738. A test machine classification is exposed as well as how to calculate the compressive strength factor and what data should be in the result report, such as: specimen identification, moulding date, curing age, rupture test date, specimen dimensions, capping type, test machine class and compressive test result (f_{ck}). The characteristic compressive strength calculation is shown in the Equation 3.1 below:

$$f_{ck} = \frac{4 \cdot F}{\pi \cdot D^2}, \quad (3.1)$$

where f_{ck} is the compressive resistance, given in megapascals (MPa); F is the maximum load reached, expressed in newtons (N); and D is the specimen's diameter, given in millimetres (mm).

4 Project Approach

4.1 Current State

Currently, the Kartrak platform has no control over the specimens life cycle, whereabouts and test results; in other words, there is no quality control supervision over the early stages of the concrete mixture. Therefore, a web application was proposed to tackle this problem. The following advantages could be obtained:

- Ease in the access of specimen data as it is centralised in a single application;
- Assuring that all concrete shipments meet the regulation and are up-to-date with the current standardisations defined by ABNT;
- Increased organisation when scheduling ruptures and viewing test results;
- Data visualisation;
- More control over the specimen's current location and state;
- Access of a great number of packages in the web, which allows a faster addition of new functionalities and speeds up the application development;
- Data is stored in databases, assuring data consistency and standardisation;
- Overall enhancement in concrete quality control.

4.1.1 Kartrak's Web Platform

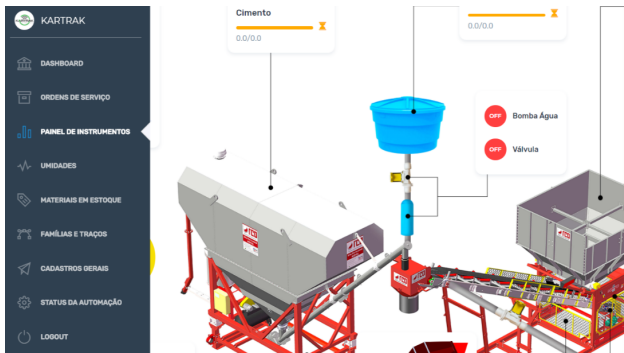
As described briefly in Chapter 2, Kartrak was developed with high portability, flexibility and easy maintainability in mind. This kind of architecture provides a scalable, flexible and comprehensive environment for the new application to be built.

It makes use of three main databases, one for the automation platform (*Cloud*), one for corporations, clients and user authentication (*ERP*) and one for receipts and invoices related data (*ERP-invoice*). *Kartrak Laboratory* will be inside the automation platform as a new feature and hence new tables were proposed to be included in the *Cloud* database.

The current web platform is implemented with *Elixir/Phoenix* and makes use of *Postgres* adapters for databases, so the same technologies were sought. A similar code design was adopted in order to maintain consistency between applications and increase maintainability and readability for other developers who might continue this work. Some screens of the platform are shown in Figure 8.

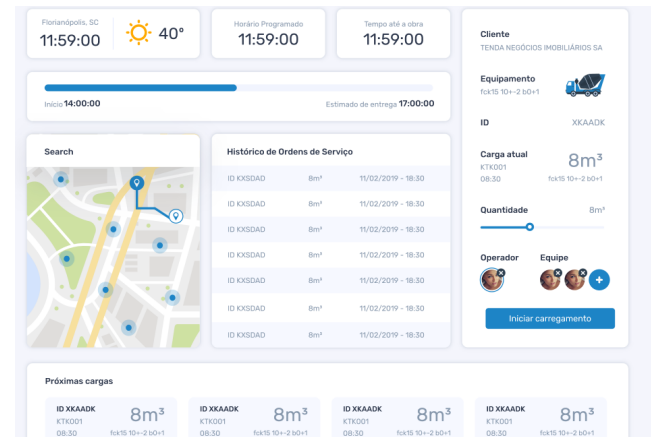
Figure 8 – Kartrak web platform screens.

(a) Kartrak Automation platform.



Source: [Kartrak 2015].

(b) Kartrak Enterprise Resource Planning platform.



Source: [Kartrak 2015].

Concrete is the main material for several structures in urbanised areas, with a great number of them being home to many people. A rigid quality supervision and control must be embraced and pursued; the probability of a disaster leading to many casualties due to poor quality concrete is huge, therefore high standards for quality level must be assured to cover all those risks.

4.2 Planned Solutions

This section contains the planned solutions to assess and solve the issues debriefed in the beginning of this chapter. The following list summarises the topics of this section as well as the steps taken to develop the project:

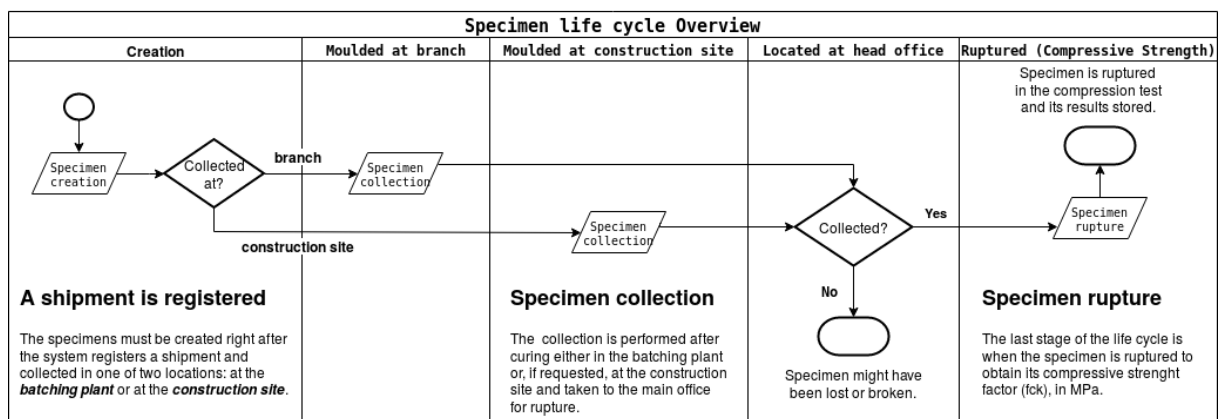
1. Software modelling
2. Quality Management
3. Kartrak *Laboratory* development plan
4. Methodology

Firstly, the software must be modelled raising its requirements and defining database model and technology stack. A continuous improvement of quality is then conceived, keeping in mind the specified requirements. With the foundation for a high quality software in a solid state, a development strategy is plotted to enhance the system's overall quality and abide by the specifications. Lastly, understanding *what* needs to be done, the methodologies approached to comprehend *how* it will be done is exposed. All modal screens shown in this chapter were made by the UI/UX team.

4.3 Software Modelling

The intent of this new feature is to supervise the whole range of locations and operations regarding the life cycle of a specimen. Based on Section 3.2, Figure 9 represents an abstraction of a possible specimen flow.

Figure 9 – A common location flow of the specimen, from creation to rupture.



Source: Author.

Supposing a company has many batching plant branches where the concrete is mixed and has its head office containing the test laboratory located elsewhere, the specimen must be collected after curing either in the branch or construction site and taken to the headquarters, where the compression test will rupture and end the specimen cycle.

The present *concrete_shipments* table has a **proof_body** attribute that flags *true* when specimen samples are required. Right after a shipment is authorised in the *Automation platform* and sent to deliver the concrete mixture, specimens are created automatically, given that **proof_body = true**, and the user is taken to the *Laboratory* feature, where the modal screen shown in Figure 10a pops up. The number of specimen rows listed will depend on default settings specified in the modal shown in Figure 10c.

After the period of curing, the specimen batch must be collected and shipped to the main office laboratory and, upon retrieving the specimen, the user registers a collection for that batch by using the modal contained in Figure 10b. To contemplate the case where a specimen is not collected due to any reason, an observation field is made available for justification. At this point, the batch of specimens has reached its final destiny and will be taken to the compression test machine. The result data is inputted into the rupture modal “Força (N)” field, presented in Figure 11.

Figure 10 – Kartrak *Laboratory* modal screens.

(b) Specimen collection modal.

(a) Specimen registration modal.

Source: Kartrak.

Source: Kartrak.

(c) Corporation default settings modal.

Source: Kartrak.

4.3.1 Requirements

Based on the above mentioned and reunions with the team, requirements for the application were specified. Considering the specimen cycle presented in Figure 9, the following requirements were raised:

Figure 11 – Rupture modal.

Informar ruptura

CLIENTE
ANTÔNIO RUBIK

CONTRATO
09/09/2019

ENDEREÇO DA OBRA
CAMPINAS - AV. JOÃO BATISTA MORATO DO CANTO 1161- OBRA VERSALHES GARDEN - MRV

OS
09/09/2019

N.NF
50454646464

DATA NF
09/09/2019

DATA DE RUPTURA
09/08/2020

ENSAIO REALIZADO POR
Marcela Baptista

CLASSE DA MÁQUINA DE ENSAIO
Classe 0.5

TIPO DE RUPTURA
Tipo A

TIPO DE CAPEAMENTO
Retificação

IDADE DO CORPO DE PROVA
28 dias

ALTURA
20 cm

DIÂMETRO
10 cm

FORÇA (N)
800

FCK REMESSA
25

FCK OBTIDO
7

Salvar

Source: Kartrak.

- CRUD zones for each stage of the specimen life cycle:
 - Creation;
 - Collection;
 - Rupture.
- Provide *default settings* CRUD operations for a company to auto-fill data in routine tasks;
- Design tests to cover most of the backend code;
- Create a shipment and collection overview page that displays specimen-related data;
- Display ruptures schedule;
- Generate moulding reports and compression test certificate.

After implementing CRUD zones for the specimen and its related operations, the *default settings* table for a corporation was planned. A corporation can have its own default diameter and height for moulding specimens and number of samples per age of specimen, which will be used in the automatic specimens batch creation.

Due to a tight deadline, most of the queries to meet the requirements of retrieving data to be displayed in overview pages of *collections*¹, *shipments*² and *ruptures*³ couldn't be implemented in this software release.

4.3.2 Database

In order to keep persistence of specimen data, four tables were added to the current *Cloud PostgreSQL* database. The specimen and rupture tables had its rows chosen based on ABNT standard NBR 5739:2018, as exposed in Section 3.8.3.3. The associations and cardinalities were defined:

- A **specimen** *has_one* **collection** and vice-versa;
- A **specimen** *has_one* **rupture** and vice-versa;
- A **specimen** *belongs_to* a **concrete shipment**;
- A **concrete shipment** *has_many* **specimens**;
- A **corporation** *has_many* **specimen settings** (number of samples for each age).

All tables will be added in migration files, except for the *corporations* table which already exists, in such case the table should only be *altered* by including the new default settings fields. A simplified view of the *Laboratory* database can be seen in Figure 12 of the next chapter.

4.3.3 Tech stack

To enforce acceptability, the same technologies from the *Automation* platform will be used. The communication between backend and a future frontend will be developed and tested through a GraphQL API. The technologies pursued were:

- **Elixir & Phoenix** will be the framework from the server side of the application.
- **PostgreSQL** as the main relational database management system.

¹ See Figure 14 in Annex A.

² See Figure 15 in Annex A.

³ See Figure 16 in Annex A.

- **Absinthe** will serve as a handler for GraphQL requests and responses.
- **Credo**⁴, a static code analysis tool for the Elixir language with a focus on code consistency and teaching was used for linting. It exposes refactoring opportunities in the code, too complex code fragments, warn about common mistakes, show inconsistencies in scheme nomenclature and help to enforce a desired coding style. It will be used to increase code readability and maintainability.
- **ExCoveralls**⁵ is an Elixir library for reporting test coverage statistics as console output using Erlang's *cover* to generate coverage information.
- **ExMachina**⁶ simplifies the generation of test data. When testing, many attributes must be specified to fulfil validations, even if there is no relation with the test in hand. It offers a solution to set up various models and their associations without having to rewrite the creation logic.
- **GitLab** was used to manage the project and for code configuration and version control.

4.4 Quality Management

This section encloses the quality planning to be followed throughout the development. Firstly, in conformance to the loop shown in Figure 5, the product goals are hereby defined:

- Implement mostly all backend functionality for the *Laboratory* application;
- Integrate frontend and backend of *Laboratory* through GraphQL API requests;
- Test at least 80% of the code;
- Test as many use cases as possible.

The quality model proposed has three different orientations: Product Operation, Product Revision and Product Transition; as conceptualised in the [McCall, Richards e Walters 1977](#) report. These characteristics identify quality factors as follows:

- **Product Operation** influences the extent to which the specifications are fulfilled.
 - **Correctness**: if the application actually does what it is supposed to do;

⁴ <https://github.com/rrrene/credo>

⁵ <https://github.com/parrotty/excoveralls>

⁶ https://github.com/thoughtbot/ex_machina

- **Reliability:** guarantee that data requested is consistent and uniform for all outputs;
- **Integrity:** only authorised users can access the app.
- **Product Revision** enhances the ability to modify the software in future due to user requirements.
 - **Maintainability:** keeping coding best practices a priority;
 - **Flexibility:** easily changeable, not much effort must be made to change a section of the code;
 - **Testability:** composition of modules that are easy to test.
- **Product Transition** enables the software to adapt in different environments.
 - **Portability:** if the application can run in other hardwares;
 - **Reusability:** reusing code excerpts from the *Automation* software;
 - **Interoperability:** how well all components work together.

With the quality model defined, a quality plan consisting of reviews and tests is elaborated to assess which part of the application will be inspected and tested. The reviews will be done on the folder/file structure, to assess maintainability; all API schemas must be inspected to attain a higher level of maintainability, reusability and correctness.

Automated tests will be developed due to the tight deadline of the project. Unit tests for all API resolvers were developed so as to test as many situations as possible. Manual tests will be executed to assure connections between the GraphQL API and the backend functionalities are stable and have no data loss.

Lastly, end-to-end tests are designed to ascertain that all modules are transforming and returning the correct data as well as the entire flow runs fluently. The specimen flow to be tested will be to *create* a specimen batch, *updating* it with correct data, collect and then rupture the specimen.

4.5 Kartrak *Laboratory* development plan

The raised requirements in the beginning of this chapter will be met by building the foundation blocks of the application: CRUD zones, which were created for each phase of the specimen life cycle. Once the fundamental functionality of the system is concluded, the queries to retrieve general information necessary to display data on reports are implemented.

4.5.1 Create, Read, Update and Delete zones

4.5.1.1 Default Settings

Keeping in mind that a concrete batching plant can send multiple shipments in a single day, routine operations might share data patterns such as specimen *diameter* and *height* — both measured in millimetres — which comes from the moulding equipment, is usually the same. A corporation may also follow a sampling protocol stating that, for example, two samples of three distinct curing ages (e.g., 3, 7 and 14 days) must be created for every shipment, therefore a table containing all that repetitive data is stored in the database so as to auto-fill those default values when requested.

4.5.1.2 Specimens

Sampling procedures, performed either by the shipment driver or an assistant (attribute `moulder_type`), consist of moulding a number of specimens per age (specimen batch) instead of a single specimen, therefore group operations must be executed so as to keep track of the whole batch location. The parameter holding that information is `situation`, which can have one of the following values and descriptions:

- `:pendente`: a specimen is created without a code;
- `:na_filial`: a specimen is moulded in the batching plant site;
- `:na_obra`: a specimen is moulded in the construction site;
- `:na_matriz`: a specimen is collected and received in the head office;
- `:observation`: a specimen is lost or broken (unknown location) and an `observation` text is inputted when collected;
- `:concluido`: a specimen is ruptured and reaches its end of existence.

4.5.1.3 Collection

The collection modules will have the same implementation plan of the specimens, with the difference of needing a function to update the specimen situation. The function must inspect the attribute `observation` and, if a `nil` value is provided, the situation changes to `:observation`; else it receives the value `:na_matriz`.

4.5.1.4 Ruptures

The rupture components will also share many resemblances with the other modules; they need to update the specimen `situation` after the compression test is executed, diverging only in the status being modified to `:concluido`.

A clause to check whether a specimen has already been collected must be implemented to avoid inconsistency. As explained in Section 3.8, a function to calculate the compressive strength factor f_{ck} must be designed by providing a force value, in newtons.

4.5.2 Queries

The required return data from queries are listed for each overview page below.

Collections:

- Receipt number;
- Receipt data;
- Client name;
- Construction site address;
- Specimen situation.
- **Filtering Options:**
 - Corporation;
 - Time interval;
 - Specimen situation.

Shipments:

- Start date;
- Concrete formula;
- Client name;
- Driver name;
- Shipment load volume;
- Specimen situation.
- **Filtering Options:**
 - Specimen situation.

Rupture schedule:

- Specimen code;
- Concrete formula;
- Branch location;
- Specimen age;
- Specimen moulding date;
- Days left to rupture;
- Specimen rupture date;
- Specimen situation.
- **Filtering Options:**
 - Today;
 - Next 7, 15 and 30 days;

4.6 Methodology

As the project focus was in the software itself and only a short amount of time (from a beginner developer perspective) was available, agile software development was adopted. To make sure that the modules implemented had a secure ground of testing, test-driven development was the chosen cycle of production.

Firstly, a macro understanding of the targeted problem is acquired, then the main problem will be divided into smaller issues and each one individually assessed. Creating tests prior to the actual implementation of the component aids in avoiding performance holes in the overall functionality. The smaller modules were first designed and the system was built on top of them, this type of approach corresponds to a *bottom-up* strategy, as explained in Section 3.4.

5 Implementation and Testing

This chapter contains a description of the development needed to successfully solve the problems explained in the beginning of this chapter.

5.1 Development environment

The author's notebook running Ubuntu 16.04 LTS (Xenial Xerus) as the operating system was chosen as the environment for the application to be built. Below is a list of the installed softwares necessary to build a default application:

- *Visual Studio Code* was adopted as the source code editor, for its smart syntax highlighting and autocompletion features as well as built-in support for JavaScript and extensions for Elixir and Git;
- *Elixir*, the functional language optimised on top of Erlang.
- *Hex*, a package manager necessary to get a Phoenix app running by installing dependencies needed during the development.
- *Phoenix* as the web framework for the backend to be built;
- *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 4* for managing the PostgreSQL database;

Phoenix works within the ordinary structure of a regular Elixir application. One can install Phoenix and Hex package manager by running:

(1) `$ mix local.hex -force`, (2) `$ mix local.rebar -force` and (3) `$ mix archive.install phoenixframework_link1`. All necessary dependencies are fetched and installed in this step. Now, a new project named “*Kartrak Laboratory*” could be bootstrapped using the command `$ mix phx.new kartrak-laboratory` and phoenix would generate the directory structure and all the necessary files for the application scaffold.

5.2 Database tables and model creation

Since *Kartrak Cloud Automation* application already has its database and tables created, that step was skipped and the *git* repository containing the source code was

¹ https://github.com/phoenixframework/archives/raw/master/phx_new.ez

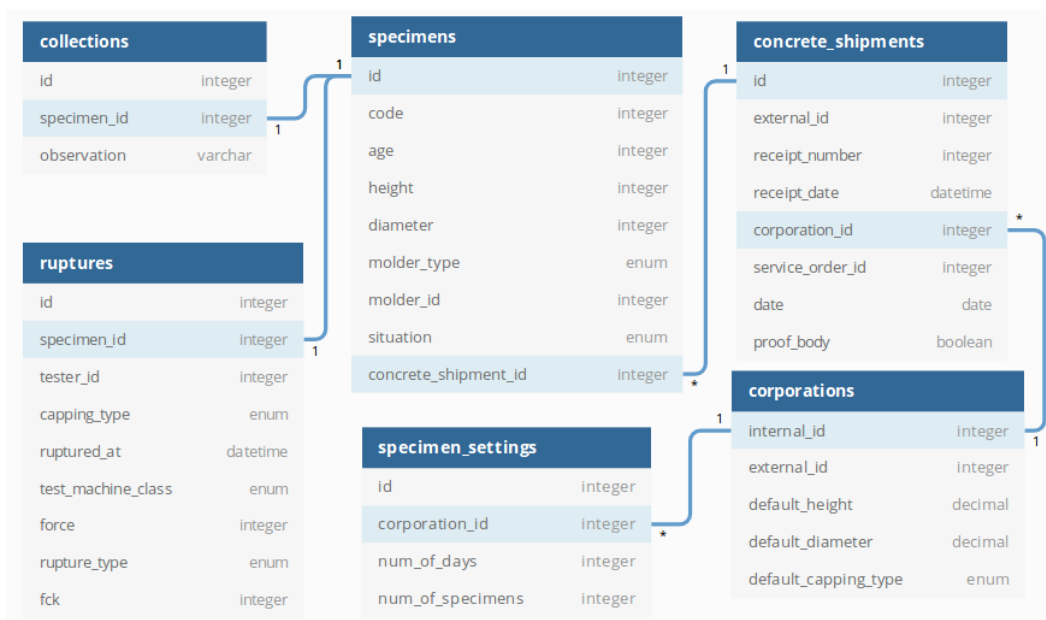
cloned running the command `$ git clone git@gitlab.com:repositoryName.git` and all tables were created using `$ mix ecto.setup`, based on *Automation's* migration files. Migrations are used to modify the database schema over time.

As discussed in Chapter 4, five tables were created:

- specimens
- collections
- ruptures
- corporations
- specimen_settings

In addition to the tables cited above, the simplified database shown in Figure 12 also has a *concrete_shipments* table included, which comes from the *Kartrak Cloud Automation* database. Its existence is to contemplate the association between *specimens*(S) and *concrete_shipments* (CS) during explanation, which is that a *specimen belongs__to* a *concrete shipment* and a *concrete shipment has__many* *specimens*.

Figure 12 – Simplified *Kartrak Laboratory* database diagram and its cardinalities.



Source: Author.

To avoid repetitiveness, only the *specimen* and *collection* models were explained. The codes exposed in 5.1 & 5.2 are the migration files, whereas codes 5.8 & 5.10 are the context and schema files, respectively, for the specimen model and codes 5.9 & 5.11 refers to the collection model.

5.2.1 Migrations

Listing 5.1 – Specimen migration file.

```

1 defmodule KartrakAutomation.Repo.Migrations.CreateSpecimens do
2   use Ecto.Migration
3
4   def change do
5     SpecimenSituationEnum.create_type()
6     SpecimenMolderType.create_type()
7
8     create table(:specimens) do
9       add :code, :integer
10      add :age, :integer, null: false
11      add :height, :decimal, null: false
12      add :diameter, :decimal, null: false
13      add :molder_type, SpecimenMolderType.type()
14      add :molder_id, :integer
15      add :situation, SpecimenSituationEnum.type(), default: "pendente"
16      add :concrete_shipment_id, references(:concrete_shipments, on_delete: :nothing),
17        null: false
18
19      timestamps()
20    end
21
22    create(index(:specimens, [:concrete_shipment_id]))
23  end

```

Listing 5.2 – Collection migration file.

```

1 defmodule KartrakAutomation.Repo.Migrations.CreateCollections do
2   use Ecto.Migration
3
4   def change do
5     create table(:collections) do
6       add :observation, :string
7       add :specimen_id, references(:specimens, on_delete: :nothing), null: false
8
9       timestamps()
10    end
11
12    create(unique_index(:collections, [:specimen_id]))
13  end
14 end

```

In order to create a module in Elixir, the *defmodule* macro must be used. The *def* macro is used to define functions in that module. Both files use the *Ecto.Migration* behaviour, which is a module that provides many helpers for migrating the database.

Inside the *change* function, the table is created and its attributes and types are defined. When an attribute consists of the `enum` type in *Elixir*, it's defined as a list of custom atoms, as shown in 5.3. Atoms are a data type that are greatly used in pattern-matching [Laurent e Eisenberg 2016] and are often used to express the state of an operation, by using values such as `:ok` and `:error`.

The `null: false` parameter is set to all obligatory attributes, preventing it from being added with a *null* field in the database. To reference another table attribute, a *foreign_key* is declared with the *references(attrs)* parameter, where *attrs* contains the external table name and the behaviour to adopt when the referenced object is deleted. A *unique_index* makes the attribute exclusive to the parent table.

Listing 5.3 – *Kartrak Laboratory enum types.*

```

1  import EctoEnum
2
3  defenum SpecimenSituationEnum, :specimen_situation_enum, [
4    :pendente,
5    :na_filial,
6    :na_obra,
7    :na_matriz,
8    :observation,
9    :concluido
10 ]
11
12 defenum SpecimenMolderType, :specimen_molder_type, [:motorista, :assistente]
13
14 defenum SpecimenCappingType, :capping_type, [:retificacao, :capeamento]
15
16 defenum SpecimenRuptureType, :rupture_type, [
17   :type_A,
18   :type_B,
19   :type_C,
20   :type_D,
21   :type_E,
22   :type_F,
23   :type_G
24 ]
25
26 # Machine classes represent the maximum admissible values for
27 # each relative error inherent to the measurement system.
28 defenum TestMachineClass, :machine_class, [:class_05, :class_1, :class_2, :class_3]

```

5.2.2 Context & Schema

Contexts are dedicated modules that expose and group related functionality, often encapsulating patterns such as data access and validation. The intent is to build an API that handles fetching, creating, updating, and deleting each model. Considering the specimen model as an example, the `Specimens` context module will be the public API for all specimen functionality in the system.

Web applications need some form of data validation and persistence, Ecto handles this issue with its out of the box support for PostgreSQL. `Ecto.Repo` is the foundation necessary to work with databases in a Phoenix application, it defines a repository. A repository maps to an underlying data store, controlled by the adapter. This project uses a Postgres adapter that stores data into a PostgreSQL database.

Following the **Don't Repeat Yourself** (DRY) principle, which states that duplication

in logic should be eliminated via abstraction, the *Crudry* library was used. *Crudry*, developed by an ECA student, is an elixir library for DRYing CRUD of Phoenix Contexts and Absinthe Resolvers while also providing a simple middleware for translating changeset errors into readable messages, as described in its repository².

Aliases are used to shorten a module call, as an example, in the specimen context (Code 5.8), instead of writing the entire path “`KartrakLaboratory.Specimens.Specimen`”, one can use `alias` and simply call “`Specimen`”, enhancing the code readability.

Using `Context.generate_functions(Specimen)` will generate CRUD functions for that schema (5.10), some of those functions are described below:

Listing 5.4 – List specimens.

```

1  # Fetch all specimen entries in the database
2  def list_specimens() do
3    Repo.all(Specimen)
4  end

```

Listing 5.5 – Get specimen by ID.

```

1  # Fetch a specimen by passing its ID
2  def get_specimen(id) do
3    Repo.get(Specimen, id)
4  end

```

Listing 5.6 – Specimen creation.

```

1  # Recieves a map with the necessary attributes and creates a specimen.
2  def create_specimen(attrs) do
3    %Specimen{}
4    |> Specimen.changeset(attrs)
5    |> Repo.insert()
6  end

```

Listing 5.7 – Specimen update.

```

1  # Updates an existing specimen by passing the Specimen struct.
2  def update_specimen(%Specimen{} = specimen, attrs) do
3    specimen
4    |> Specimen.changeset(attrs)
5    |> Repo.update()
6  end

```

One way to pass arguments to a function is through the pipe operator “`|>`”. In Code 5.7, for example, `Repo.update/1` receives a *changeset* as parameter, which comes through the pipe transformation above from function `Specimen.changeset/2` that had the *specimen* struct piped into it.

It’s clear that when mutating a model, the model’s struct must be piped through the schema’s *changeset* first and just then the database is modified through `Repo` functions. *Changesets* define the pipeline transformations a given data needs to undergo before it can be used by the application. These transformations might include type-casting, user input validation, and filtering out wrong parameters. The changeset is defined in the schema module (5.10 and 5.11).

² <https://github.com/gabrielpra1/crudry>

Both validations and constraints are turned into errors in case something goes awry. The difference is that most validations can be executed without interacting with the database and, therefore, are always executed before attempting to insert or update the database entry. Since constraints rely on database and are always safe, validations are checked before constraints. In case the changeset is not valid, an error will be raised and the constraints are not even checked.

Ecto schemas are responsible for mapping Elixir values to external data sources, as well as mapping external data back into Elixir data-structures. By default, a schema will automatically generate a primary key named *id* of type *:integer*. The *field* macro defines a field in the schema. Relationships to other schemas can also be defined. In the specimen schema (5.10), for example, *has_one* associates one *collection* with the *specimen* schema, whereas *belongs_to* creates a *foreign_key* to associate the *specimen* with the *concrete_shipment* schema. Schemas are regular structs in the form presented below using *IEx*:

```

1  iex> collection_example =
2      %Collection{
3          type: :na_obra,
4          molder_id: 1,
5          specimen_id: 1,
6          collected_at: "2019-09-18T11:27:24Z"
7      }

```

Inside a module, private functions can be defined with *defp* and can only be invoked locally. The final transformation pipe in the *specimen* changeset is a private function dubbed *null_code?*, it has an arity³ of 2 (*defp/2*) and checks if a specimen has the attribute *code*. If the given *Specimen* returns a code with *nil* value, it modifies the *situation* attribute to “*pendente*” and updates the changeset. In any other case it returns the changeset in pristine conditions.

It’s also observable that the function was defined two times; that is done in order to prevent any changes to an invalid changeset. It’s achieved by pattern-matching the *key: value* pair *valid?: false* with the changeset map. If a match happens, the unmodified changeset is returned.

³ Number of arguments that a function can handle.

Listing 5.8 – Specimen context file.

```
1 defmodule KartrakLaboratory.Specimens do
2   @moduledoc """
3     The Specimens context.
4     """
5
6   import Ecto.Query, warn: false
7   require Crudry.Context
8
9   alias Crudry.Context
10  alias Dataloader.Ecto, as: DataloaderEcto
11  alias KartrakAutomation.Repo
12  alias KartrakLaboratory.Specimens.Specimen
13
14  Context.generate_functions(Specimen)
15
16  def data do
17    DataloaderEcto.new(Repo, query: &query/2)
18  end
19
20  def query(queryable, _args) do
21    order_by(queryable, :id)
22  end
23 end
```

Listing 5.9 – Collection context file.

```
1 defmodule KartrakLaboratory.Collections do
2   @moduledoc """
3     The Collections context.
4     """
5
6   import Ecto.Query, warn: false
7   require Crudry.Context
8
9   alias Crudry.Context
10  alias Dataloader.Ecto, as: DataloaderEcto
11  alias KartrakAutomation.Repo
12  alias KartrakLaboratory.Collections.Collection
13
14  Context.generate_functions(Collection)
15
16  def data do
17    DataloaderEcto.new(Repo, query: &query/2)
18  end
19
20  def query(queryable, _args) do
21    order_by(queryable, :id)
22  end
23 end
```

Listing 5.10 – Specimen schema file.

```

1 defmodule KartrakLaboratory.Specimens.Specimen do
2   @moduledoc """
3     (Simplified) Specimen schema
4     """
5
6   use Ecto.Schema
7   import Ecto.Changeset
8
9   alias KartrakAutomation.Shipments.ConcreteShipment
10  alias KartrakLaboratory.{
11    Collections.Collection,
12    Ruptures.Rupture
13  }
14
15  schema "specimens" do
16    field :code, :integer
17    field :age, :integer
18    field :situation, SpecimenSituationEnum
19    field :molding_date, :utc_datetime
20
21    belongs_to :concrete_shipment, ConcreteShipment
22    has_one :collection, Collection, references: :id, foreign_key: :specimen_id
23    has_one :rupture, Rupture, references: :id, foreign_key: :specimen_id
24    timestamps()
25  end
26
27  @doc false
28  def changeset(specimen, attrs) do
29    specimen
30    |> cast(attrs, [
31      :code,
32      :age,
33      :situation,
34      :molding_date,
35      :concrete_shipment_id
36    ])
37    |> validate_required([
38      :age,
39      :height,
40      :diameter,
41      :concrete_shipment_id
42    ])
43    |> foreign_key_constraint(:concrete_shipment_id)
44    |> null_code?(specimen)
45  end
46
47  defp null_code?(%{valid?: false} = changeset, _), do: changeset
48
49  defp null_code?(changeset, specimen) do
50    case specimen.code do
51      nil -> put_change(changeset, :situation, :pendente)
52      _else -> changeset
53    end
54  end
55 end

```

Listing 5.11 – Collection schema file.

```

1 defmodule KartrakLaboratory.Collections.Collection do
2   @moduledoc """
3     Defines the Collection schema.
4     """
5   use Ecto.Schema
6   import Ecto.Changeset
7
8   alias KartrakLaboratory.Specimen
9
10  schema "collections" do
11    field :type, CollectionType
12    field :collected_at, :utc_datetime
13    field :molder_id, :integer
14    field :observation, :string
15
16    belongs_to :specimen, Specimen, references: :id, foreign_key: :specimen_id
17
18    timestamps()
19  end
20
21  @doc false
22  def changeset(collection, attrs) do
23    collection
24    |> cast(attrs, [:type, :observation, :collected_at, :molder_id, :specimen_id])
25    |> validate_required([
26      :type,
27      :collected_at,
28      :molder_id,
29      :specimen_id
30    ])
31    |> foreign_key_constraint(:specimen_id)
32    |> unique_constraint(:specimen_id)
33  end
34 end

```

5.3 GraphQL API

GraphQL APIs are organised in terms of types and fields, which runs in a single endpoint. This endpoint will be used to view and alter data stored in the database through queries and mutations.

The *router* implementation in 5.12 shows how incoming requests are parsed and dispatched to the correct action, passing parameters as needed. After a request goes through the pipelines, route paths or *URLs* are generated to access resources. With the endpoint set up, all types, fields and operation, i.e., **queries** and **mutations**, related to a model must be defined and imported in the *schema* file.

In order to execute some queries, the user must have a valid token in the header to be authenticated. After a token has been successfully decoded into a user name, the authenticated user name is added to the *Absinthe context*, which allows resolvers to perform

operations with this information, such as verifying data access permission. *Rajska*⁴, created by an ECA alumnus, is an elixir authorisation library for Absinthe that provides many middlewares to handle user permissions.

Once the user establishes a connection, the Absinthe library handles the oncoming GraphQL queries and mutations requests by redirecting them to the specified resolvers. Non-authenticated users are allowed to perform login mutations only. The user authentication system was already implemented by another developer.

Listing 5.12 – Simplified GraphiQL endpoint implementation.

```

1 defmodule KartrakAutomationWeb.Router do
2   use KartrakAutomationWeb, :router
3
4   pipeline :api do
5     plug :accepts, ["json"]
6   end
7
8   pipeline :graphql do
9     plug KartrakAutomationWeb.AbsintheContext
10  end
11
12  scope "/" do
13    pipe_through [:api, :graphql]
14
15    forward "/graphiql", Absinthe.Plug.GraphiQL,
16      schema: KartrakAutomationWeb.Schema,
17      interface: :playground,
18      context: %{pubsub: KartrakAutomationWeb.Endpoint},
19      socket: KartrakAutomationWeb.UserSocket
20  end
21 end

```

5.3.1 Queries & Mutations

The GraphQL API is how data will be exposed to the web, therefore the model's GraphQL schema must be defined in the web context of the application. In order to better explain the steps taken to write the web schema code, the implementation was divided into three steps: write the GraphQL schema (with Absinthe) “skeleton”; write a *query* to fetch all **Specimens** and a *mutation* to create a **Specimen**; implement the resolver function to actually access the database and test the functionality in GraphiQL interface. The complete *CRUD* zones will be shown in 5.4.

⁴ <https://github.com/rschef/rajska>

Listing 5.13 – Specimen GraphQL schema: scaffold.

```

1  defmodule KartrakAutomationWeb.Schema do
2    @moduledoc """
3      Defines the Absinthe graphql schema.
4      """
5    use Absinthe.Schema
6    alias KartrakAutomationWeb.SpecimenResolver
7
8    # Object types definition area -----
9    # -----
10
11   query do
12     # query entry point
13   end
14
15   mutation do
16     # mutation entry point
17   end
18 end

```

The `Absinthe.Schema` module is used to provide some macros for schema building. Absinthe Schemas are also type checked at compile time, which means that if a type that doesn't exist is referred to, Absinthe will catch the error rapidly. With the empty root query and mutation objects set, the input/output objects types are written in the indicated area (Code: 5.13) and the abilities to `get` all *specimens* and `create` one are implemented as follows.

Listing 5.14 – (Simplified) Specimen GraphQL schema: objects definition.

```

1  # Object types -----
2  enum :specimen_situation_enum do
3    value :pendente
4    value :na_filial
5    value :na_obra
6    value :concluido
7  end
8
9  @desc "a specimen"
10 object :specimen do
11   field :id, :integer
12   field :code, :integer
13   field :age, :integer
14   field :situation, :specimen_situation_enum
15   field :concrete_shipment_id, :integer
16 end
17
18 @desc "Params to create a specimen"
19 input_object :specimen_params do
20   field :code, :integer
21   field :age, non_null(:integer)
22   field :situation, :specimen_situation_enum
23   field :concrete_shipment_id, non_null(:integer)
24 end
25 # -----

```

If any attribute has a custom `enum` type, its values must be defined. The `@desc` attribute will be shown as an object description in the GraphQL interface. When *querying*, the `:specimen` object fields will be available, while *mutating*, the `:specimen_params` input_object fields will be available. A `non_null` parameter will abort the query if the specified field is left blank.

Listing 5.15 – Query and mutation definition.

```

1  query do
2    @desc "Get all specimens"
3    field :all_specimens, list_of(:specimen) do
4      resolve &SpecimenResolver.list_specimens/2
5    end
6  end
7
8  mutation do
9    @desc "register a new specimen"
10   field :create_specimen, type: :specimen do
11     arg :params, non_null(:specimen_params)
12     resolve &SpecimenResolver.create_specimen/2
13   end
14 end
15 end

```

Resolvers are functions mapped to GraphQL fields, with their actual behavior. A field is specified for a resolver by using the `resolve` macro and passing it a function. The field `:all_specimens` expects the function `SpecimenResolver.list_specimens/2` to return a list of *specimens*. When mutating, the `:create_specimen` field will receive an argument of type `:specimen` and expect the `SpecimenResolver.create_specimen/2` to return a new specimen with the given attributes passed as argument.

“`&SpecimenResolver.list_specimens/2`” is a reference to the 2 arity function `list_specimens` found in the `KartrakAutomationWeb.SpecimenResolver` module, presented below.

Listing 5.16 – Resolver implementation.

```

1  defmodule KartrakAutomationWeb.SpecimenResolver do
2    @moduledoc """
3      Defines the Specimen resolver functions.
4      """
5    alias KartrakLaboratory.Specimens
6    alias KartrakLaboratory.Specimens.Specimen
7
8    def list_specimens(_args, _info) do
9      {:ok, Specimens.list_specimens()}
10   end
11
12   def create_specimen(%{params: params}, _info) do
13     Specimens.create_specimen(params)
14   end
15 end

```

A resolver module must handle all function calls from the web-context schema. In the resolver above, the functions defined in 5.4 and 5.6 are called to fetch all specimens and create one, respectively.

One might question the reason to return the tuple `{:ok, Specimens.list_specimens()}` instead of just the second term; that is because the function `list_specimens/0` returns only the `Specimen` struct and GraphQL expects a tuple with `{:ok, data}` or `{:error, error}`.

To support that, the function `create_specimen/1` returns a `{:ok, specimen}` tuple, when succeeds, so itself is sufficient as the return data. The server can be started by running `$ iex -S mix phx.server` and accessed at `localhost:4000/graphiql` in the browser.

Listing 5.17 – All specimens query.

```

1  query allSpecimens {
2    allSpecimens {
3      id
4      code
5      age
6      situation
7      concreteShipmentId
8    }
9  }

```

Listing 5.18 – Create specimen mutation.

```

1  mutation createSpecimen {
2    createSpecimen(params: {
3      code: 1444,
4      age: 1,
5      situation: NA_FILIAL,
6      concreteShipmentId: 1
7    }) {
8      id
9      code
10     age
11     concreteShipmentId
12   }
13 }

```

Listing 5.19 – GraphQL query interpretation
of 5.17.

```

1  [debug] QUERY OK source="users" db=0.4ms
2  SELECT
3    u0."id" ,
4    u0."name" ,
5    u0."email" ,
6    u0."role" ,
7    u0."encrypted_password" ,
8    u0."created_at" ,
9    u0."updated_at" ,
10   u0."corporation_id"
11 FROM "users" AS u0 WHERE (u0."id"=$1) [3]
12
13 [debug] QUERY OK source="specimens" db=1.1
14   ms queue=1.1ms
15 SELECT
16   s0."id" ,
17   s0."code" ,
18   s0."age" ,
19   s0."height" ,
20   s0."diameter" ,
21   s0."concrete_shipment_id" ,
22   s0."inserted_at" ,
23   s0."updated_at"
FROM "specimens" AS s0

```

Listing 5.20 – Response data from query
5.17.

```

1  {
2    "data": {
3      "allSpecimens": [
4        {
5          "situation": "NA_FILIAL" ,
6          "id": 1,
7          "concreteShipmentId": 1,
8          "code": 144101,
9          "age": 7
10         },
11         {
12           "situation": "NA_FILIAL" ,
13           "id": 2,
14           "concreteShipmentId": 1,
15           "code": 144102,
16           "age": 14
17         },
18       ]
19     }
20   }

```

In 5.19 the actual query used to access the database appears in the VS Code terminal⁵, which can be used for debugging. Firstly, the query is authenticated for the user with `id = 3` (`$1` is a reference to the first argument passed in the login function, which is not presented here); secondly, after the authentication succeeds, the attributes from `specimens` table are retrieved. The elapsed time of the database access is also shown.

It can be concluded that GraphQL has a clear separation of structure and behaviour. The structure of a GraphQL server is represented by its schema, an abstract description of the server's capabilities and *resolver* functions, which are key components to determine the server's behaviour.

In order to increase code readability, all types were defined in separate files and imported into a single file. In this way, all laboratory web-context schema types can be loaded from a single line of code. That is, Codes 5.14 and 5.15 would be written in a separated file and imported into a file arbitrarily named `laboratory_types.ex` with the function `import_types(Specimens)`, minding the *aliases*. Then, `laboratory_types.ex` would be loaded via `import_types/1` in the `schema.ex` file.

⁵ An interface in which text based commands can be typed and executed.

5.4 CRUD Zones

This section contains some of the fundamental functionalities of each model and explanations regarding specific sections, instead of the entire code. Reminding briefly of the specimen life cycle explained in Section 3.8:

1. When a concrete shipment is registered **and** has the attribute `proof_body = true`, a specimen batch based on the corporation's default settings is created; if a specimen is registered with a **null** code, its `situation` is set to *pendente*;
2. After ageing, specimens are collected and taken to a laboratory for compression tests;
3. The specimen is ruptured to obtain its compressive strength (f_{ck}), measured as $f_{ck} = F/A$ in megapascal [MPa], where **F** is the maximum load applied to the specimen in newtons [N] and **A** the cross sectional area of the specimen (in mm^2).

Based on the thoughts above and the screens shown in Section 4, the details of each model implemented will be presented and discussed below.

5.4.1 Corporation and Specimen default settings

Analogously to Code 5.15, a query to retrieve the corporation default settings via `id` was implemented. A mutation to update existing corporations from the ERP database was created in its schema file and the resolver function generated via *Crudry*.

Listing 5.21 – Corporation default settings update mutation.

```

1  # Mutations
2  object :corporation_defaults_mutations do
3    @desc "update default settings for a corporation"
4    field :update_corporation_defaults, type: :corporation_defaults do
5      arg :id, non_null(:integer)
6      arg :params, non_null(:corporation_defaults_params)
7      middleware Rajska.QueryAuthorization, [permit: :user, scope: false]
8      resolve &CorporationResolver.update_corporation/2
9    end
10  end

```

As for the `specimen settings`, a slightly different approach was needed. According to what was explained in the previous chapter, a corporation can have *many* specimen settings, therefore a list must be manipulated and several database operations shall be executed.

Repo transactions are the countermeasure for this kind of problem. In Elixir, the function `Repo.transaction/2` works by rollbacking all modifications made in the database

in case one of them fails. Both resolver functions of creation and update below are wrapped inside a transaction, ensuring that the database remains untouched if any error occurs.

Listing 5.22 – Specimen settings creation.

```

1  def create_specimen_settings(%{params: params}, _info) do
2    creation =
3      Repo.transaction(fn ->
4        {
5          :ok,
6          with %{default_diameter: _default_diameter} <-
7            Corporations.get_corporation(corporation_id) do
8              for specimen_settings_params <- params do
9                case Corporations.create_specimen_settings(
10                 Map.put(specimen_settings_params, :corporation_id, corporation_id)
11                 ) do
12                   {:ok, specimen_settings} -> specimen_settings
13                   {:error, changeset} -> Repo.rollback(changeset)
14                 end
15               end
16             else
17               nil->Repo.rollback({:error, "no corporation found with id [#{corporation_id}]"})
18             end
19           }
20         end)
21
22     case creation do
23       {:ok, creation} -> creation
24       {:error, _} -> creation
25     end
26   end

```

The GraphQL schema calls the `create_specimen_settings` function and passes a `map` with the *specimen settings* parameters `list`, that `map` is then pattern-matched to a variable named `params`. Since the resolver function expects a `{:ok, data}` resolution, that structure was adopted.

Inside the transaction, before a specimen setting is created for each list iteration, a `with` clause verifies if the corporation to which the specimen setting is being set exists; in case of a failure, the operation rolls-back returning either an error message or the `changeset`, depending on the error location. When the transaction finishes, it is then pattern-matched for errors and returned to the schema. The `with` statement is a condition checker, the execution continues as long as *left* \leftarrow *right* matches.

In the code below, it will abort the execution if the function `get_specimen_settings/1` returns a `SpecimenSettings` struct with a field `id` inside it (the compiler ignores any value with an underscore before its name), if there are no specimens with the given `id`, it executes the `case` block rolling back the transaction with an error message.

Listing 5.23 – Specimen settings update transaction.

```

1  Repo.transaction(fn ->
2    {
3      :ok,
4      for specimen_settings_params <- params do
5        with specimen_settings =
6          %SpecimenSettings{id: _specimen_id} <- Corporations.get_specimen_settings(id)
7          do
8            case Corporations.update_specimen_settings(specimen_settings,
9              specimen_settings_params) do
10             {:ok, specimen_settings} -> specimen_settings
11             {:error, changeset} -> Repo.rollback(changeset)
12           end
13         else
14           nil -> Repo.rollback("no specimen_settings found with id [#{id}]")
15         end
16       end
17     }
18   end)

```

5.4.2 Specimens

Everytime a **shipment** is created with the attribute **proof_body** as *true*, a specimen batch related to that shipment must be created. All specimens must have an identification code so it can be traced. Since the specimens might be created without a code, their **situation** attribute must be automatically set to “*pendente*”. An **update_specimen_batch/2** function was implemented so those fields can be filled.

Two alternatives for the specimen creation were proposed: specifying all attributes of the specimens in the batch, or an automatic creation with the only parameters passed being the **corporation_id** and **concrete_shipment_id**, therefore all specimens attributes would be retrieved from the corporation default settings.

For the first alternative, a list with the specimen parameters is given and thus a simple iteration through that list in an analogous way done in Code 5.23 is enough. The second option was implemented by means of recursive calling since immutability is a pillar of functional programming.

Firstly, the corporation and its default settings are loaded and checked for nullity; once passed that condition, the **create_specimens/3** function (Code 5.25) is called to create the default number of samples for each specimen age; it has three call matches where the first one will inspect if the corporation actually has default settings and roll back the transaction if it doesn't. The recursion resides in calling the function inside itself and continue executing until the **when** clause is flagged.

Listing 5.24 – Specimen batch creation based on the default settings of a corporation.

```

1  Repo.transaction(fn ->
2    {
3      :ok,
4      # corporation default dimensions
5      with %{
6        default_diameter: default_diameter,
7        default_height: default_height
8      } <- Corporations.get_corporation(corporation_id) do
9        # specimen default settings
10       for s <- Corporations.get_specimen_default_settings(corporation_id) do
11         case create_specimens(
12           s.num_of_specimens,
13           %{
14             age: s.num_of_days,
15             diameter: default_diameter,
16             height: default_height,
17             concrete_shipment_id: concrete_shipment_id
18           },
19           default_diameter
20         ) do
21           {:ok, specimen} -> specimen
22           {:error, error} -> Repo.rollback(error)
23         end
24       end
25     else
26       nil -> %{error: "no corporation found with id [#{corporation_id}"]}
27     end
28   }
29 end)

```

A (seemingly) common approach in Elixir to call the same function with different arguments can be observed in Code 5.25, where the two first function definitions make the use of Guards, which start with the *when* keyword.

Guards are a way to augment pattern matching with more complex checks, such as type checking the `default_diameter` with `is_nil/1`, to verify if the Corporation actually has default settings setted, and using recursive loop constructs to create the default quantity of specimens.

Listing 5.25 – Recursive specimen creation.

```

1  defp create_specimens(_, _, default_diameter) when is_nil(default_diameter) do
2    {:error, "Corporation doesn't have default settings"}
3  end
4
5  defp create_specimens(quantity, params, _) when quantity <= 1 do
6    {:ok, %{id: __specimen_code}} = Specimens.create_specimen(params)
7  end
8
9  defp create_specimens(quantity, params, _) do
10   {:ok, %{diameter: diameter}} = Specimens.create_specimen(params)
11   create_specimens(quantity - 1, params, diameter)
12 end

```

To update a batch of specimens, the implemented function expects two lists: the `id`'s of all specimens in the batch and their update attributes, i.e., `code`, `situation`, `molder_type` and `molder_id`, which are the ones missing from the first option of creation.

From an object-oriented view, iterating two lists simultaneously could be achieved by referring both lists with the same mutating loop index, yet it cannot be done in Elixir due to immutability. The “elixir way” of solving that is shown in Code 5.26.

The `Enum.zip/2` function receives two lists (`params` and `id`) and zips the first element of each list into a tuple, which goes on until one of the lists reaches its end and a list is returned with all combined tuples.

In this case, both lists will always have the same length. That list of tuples is piped into the `Enum.map/2` higher-order function, where each `{params, id}` tuple in the list is run through a function that holds the update operation, then the updated specimens are returned in a list.

Listing 5.26 – Function to update a specimen batch.

```

1  Repo.transaction(fn ->
2    {
3      :ok,
4      params
5    } |> Enum.zip(id)
6    |> Enum.map(fn {params, id} ->
7      with specimen = %Specimen{id: _specimen_id} <- Specimens.get_specimen(id) do
8        case Specimens.update_specimen(specimen, params) do
9          {:ok, updated_specimen} -> updated_specimen
10         {:error, changeset} -> Repo.rollback(changeset)
11       end
12     else
13       nil -> Repo.rollback({:error, "no specimen found with id [#{id}]"})
14     end
15   end)
16 }
17 end)

```

5.4.3 Collections

When a specimen is collected, the flow can follow two distinct paths: its `situation` is updated to `:observation` in case anything happens to specimen, otherwise it changes to `:na_matrix`. Both `create` and `update` functions must go through the `update_specimen_situation/2` to ascertain that the `situation` attribute is always up-to-date with the actual specimen location.

The pipeline section where the `nil_to_error/3`⁶ function is called transforms a `nil` value in an `{:error, message}` tuple if an error is raised during the update operation. When `update_specimen_situation/2` is called from the update function, it passes a

⁶ Generated by the `Crudry` library macro `Resolver.generate_functions/3`.

tuple `{:ok, schema}` and thus a second definition to update the specimen situation was implemented (not shown here) to pattern-match that tuple and return it in the end of execution.

Listing 5.27 – Function to register a collection batch.

```

1 Repo.transaction(fn ->
2   {
3     :ok,
4     for collection_params <- params do
5       case Collections.create_collection(collection_params) do
6         {:ok, collection} ->
7           update_specimen_situation(collection, collection_params.specimen_id)
8         {:error, error} ->
9           Repo.rollback(error)
10      end
11    end
12  }
13 end)

```

Listing 5.28 – Function to update a collection.

```

1 def update_collection(%{id: id, params: params, _info} do
2   id
3   |> Collections.get_collection()
4   |> nil_to_error("collection", fn record ->
5     Collections.update_collection(record, Map.delete(params, :specimen_id))
6   end)
7   |> update_specimen_situation(params.specimen_id)
8 end)

```

Listing 5.29 – Collection resolver private function to update the specimen situation.

```

1 defp update_specimen_situation(schema, specimen_id) do
2   specimen = %Specimen{} = Specimens.get_specimen(specimen_id)
3   case is_nil(schema.observation) do
4     true ->
5       {:ok, %Specimen{}} =
6         Specimens.update_specimen(specimen, %{situation: :na_matriz})
7     false ->
8       {:ok, %Specimen{}} =
9         Specimens.update_specimen(specimen, %{situation: :observation})
10    end
11  schema
12 end

```

5.4.4 Ruptures

The `update_rupture/2` function is analogous to the one shown in the previous subsection and also uses the `update_specimen_situation/2`, only it modifies the `situation` with `:concluido`, for that reason it will not be shown.

A compression test machine outputs the maximum load applied (in newtons) to

the specimen until it ruptures, and that value persists in the database through the `force` attribute. The maximum load will be used in the function `fck_calculation/2` to obtain the f_{ck} value in megapapascals.

The values used for default settings are defined with the *decimal* type and the f_{ck} is obtained by using *Decimal* library functions, which performs arbitrary precision decimal arithmetic⁷.

Listing 5.30 – Function to register a rupture.

```

1  def create_rupture(%{params: params}, _info) do
2    case collected?(params.specimen_id) do
3      %{specimen_id: specimen_id} ->
4        fck = fck_calculation(params.force, Specimens.get_specimen(specimen_id))
5        case Ruptures.create_rupture(Map.put(params, :fck, fck)) do
6          {:ok, rupture} -> {:ok, update_specimen_situation(rupture, specimen_id)}
7          {:error, error} -> {:error, error}
8        end
9
10       {:error, error} -> {:error, error}
11       nil -> {:error, "specimen with id [#{params.specimen_id}] not collected yet"}
12     end
13   end

```

Listing 5.31 – Collection verifier function.

```

1  defp collected?(specimen_id) do
2    case Specimens.get_specimen(specimen_id) do
3      nil -> {:error, "specimen with id [#{specimen_id}] doesn't exist"}
4      %Specimen{id: id} -> Repo.get_by(Collection, specimen_id: id)
5    end
6  end

```

Listing 5.32 – f_{ck} calculation function.

```

1  def fck_calculation(force, %{diameter: diameter}) do
2    # fck = 4*F/(pi*D^2) =
3    D.round(
4      D.div(D.mult(4, force), D.mult(D.from_float(:math.pi()), D.mult(diameter, diameter))),
5      1
6    )
7  end

```

5.5 Testing

5.5.1 Unit Tests

A test for each API resolver mentioned in the past section was designed to assess the CRUD queries and mutations that are passed to them. The tests intended to reach most of the clauses inside a resolver function, in order to increase overall test coverage.

⁷ Arbitrary Precision: the precision of any calculation is limited by the largest number that can be stored in one of the processor's registers.

All tests will follow both examples below, differing only in specific logic assertion. Codes 5.33 and 5.34 will suffice for explanation purposes. A rupture can be registered only if the specified specimen actually exists and was previously collected, generating an error message informing what went wrong when it does. The first test evaluates the case of a non-existent specimen being ruptured.

The second test simulates a non-collected specimen being ruptured. After asserting that it does not succeed, a collection is inserted in the same test using the factory library *ExMachina* and the specimen is ruptured. Two cases were covered in this testing of application-level behaviour.

Listing 5.33 – Asserting that creating a rupture with invalid data generates an error.

```
1 test "create a rupture with non-existent specimen" do
2   variables =
3     %{
4       params:
5         %{
6           capping_type: :retificacao ,
7           force: 786000,
8           ruptured_at: "2019-09-18T11:27:24Z",
9           specimen_id: 8001,
10          test_machine_class: :class_1 ,
11          tester_id: 12
12        }
13      }
14   result = RuptureResolver.create_rupture(variables , %{})
15
16   assert result == {:error , "specimen with id [8001] doesn't exist"}
17 end
```

Listing 5.34 – Rupture creation unit test to assert that valid data creates a rupture.

```

1  test "create a rupture with non-collected specimen, then collects and ruptures it" do
2    {:ok, shipment} = create_concrete_shipment()
3
4    %Specimen{id: specimen_id} = insert(:specimen, concrete_shipment_id: shipment.id)
5
6    variables =
7      %{
8        params:
9          %{
10           capping_type: :retificacao,
11           force: 786000,
12           rupture_type: :type_A,
13           ruptured_at: "2019-09-18T11:27:24Z",
14           specimen_id: specimen_id,
15           test_machine_class: :class_1,
16           tester_id: 12
17         }
18       }
19    result = RuptureResolver.create_rupture(variables, %{})
20
21    assert result == {:error, "specimen with id [#{specimen_id}] not collected yet"}
22
23    # collection occurs
24    insert(:collection, specimen_id: specimen_id)
25    {:ok, result} = RuptureResolver.create_rupture(variables, %{})
26
27    assert result.capping_type == :retificacao
28    assert result.force == Decimal.new(786000)
29    assert result.rupture_type == :type_A
30    assert result.specimen_id == specimen_id
31    assert result.test_machine_class == :class_1
32    assert result.tester_id == 12
33  end

```

5.5.2 Integration End-to-end Tests

The frontend will be based on GraphQL queries that interact with the application, therefore were performed integration tests in the GraphQL API interface. Essentially there are three main ways to perform queries and mutations using Absinthe: Triggering a request to the GraphQL endpoint; Calling `Absinthe.run/3` to execute the query/mutation directly; Unit testing resolver modules. Each one has its optimal suitability, considering that the objective is to make sure that different flows of the application run smoothly, end-to-end resolver tests were run to secure that.

To simulate all specimen life cycle stages, the flow presented in Figure 9 was implemented with an automated test that asserts the main data after mostly all resolver functions. The *ERP* database is not available for testing and thus the automated end-to-end resolver test was designed. Code 5.35 shows a possible implementation of the connection between GraphQL and backend for a simple query to obtain the ages from all specimens in the database.

To set up a new connection with the GraphQL endpoint when each test is run, the `ConnCase` module was used. To assess the authentication layer that handles the access points of queries and mutations, a valid token was cached in the connection. A post request containing the query and variables is sent to the API and its response checked for `status code 200 (OK)`, which indicates that the request has been processed successfully on server and it has a *response body* with correct data. Some tests were implemented based in the examples from [Williams e Wilson 2018](#).

Listing 5.35 – Possible testi to assess the integration between modules.

```

1 describe "list specimens" do
2   test "list all specimens", context do
3     {:ok, shipment} = create_concrete_shipment()
4     %Specimen{id: specimen_id, age: age, code: code} =
5       insert(:specimen, concrete_shipment_id: shipment.id)
6
7     response =
8       context.conn |> post("/graphql", query: query, variables: %{})
9
10    assert json_response(response, 200) == %{
11      "data" => %{"allSpecimens" => [
12        %{
13          "id" => specimen_id,
14          "age" => age,
15          "code" => code,
16          "concreteShipmentId" => shipment.id
17        }
18      ]
19    }
20  }
21 end
22 end

```

5.5.3 Code Coverage Statistics

Conforming to what was mentioned in Section 3.4, *code coverage* is a type of white-box testing, for it is a measure of how efficient is the application. It outputs a quantitative measure of code coverage, reflecting the application's quality. Several test cases should be implemented so as to simulate a greater number of system's states, which helps to understand which sections of the tested modules are not being executed.

Although it has many advantages, it measures the percentage of lines that are executed, if any specification is not yet implemented, the coverage might still be 100%. It also does not aid in determining whether all possible values of a module are tested. White-box techniques analyses the structure, not specified requirements. The *ExMachina* library was chosen to obtain the aforementioned metrics.

6 Development Analysis and Results

With the objective of implementing a working backend functionality achieved, the developed application allows specimens to be registered, updated and collected in batches, and have its compressive strength factor calculated based on an inputted force from the test machine.

Analysing from the quality perspective, the specified objectives were pursued following a methodological approach and achieved with the desired level of confidence. Multiple unit tests were designed to avoid flawed logic paths in the program's core and, in addition to that, an end-to-end test was implemented to integrate the backend modules and assure a smooth usability of the application. The following section comprises the statistical results from ExCoveralls test library.

6.1 Test Coverage

The backend components tests were evaluated via the ExCoveralls library `$ mix coveralls` command. When executed, it outputs via *terminal* the code coverage results shown in 6.1.

Listing 6.1 – Test coverage result.

```

1 .....
2
3 Finished in 1.2 seconds
4 21 tests , 0 failures
5
6 Randomized with seed 947683
7
8 COV    FILE                                     LINES RELEVANT  MISSED
9 89.5% (...)/resolvers/laboratory/collection_resolver.ex      66      19      2
10 80.0% (...)/resolvers/laboratory/corporation_resolver.ex     70      20      4
11 71.4% (...)/resolvers/laboratory/rupture_resolver.ex         72      14      4
12 90.3% (...)/resolvers/laboratory/specimen_resolver.ex       120     31      3
13 100.0% lib/kartrak_laboratory/collections/collection.ex      26       2      0
14 100.0% lib/kartrak_laboratory/corporations/specimen_settings.ex 24       2      0
15 100.0% lib/kartrak_laboratory/kartrak_laboratory_enums.ex   28       5      0
16 100.0% lib/kartrak_laboratory/ruptures/rupture.ex           49       2      0
17 100.0% lib/kartrak_laboratory/specimens/specimen.ex         64       6      0
18 [TOTAL] 87.1%
19

```

The 21 implemented automated tests succeeded the assertions, increasing the system's level of confidence. With more than 87.1% of the code covered, calculated by

means of the Equation 6.1, the specifications are being met.

$$COV(TOTAL) = 100 \cdot \left(1 - \frac{\sum_{i=1}^n ML_i}{\sum_{i=1}^n RL_i}\right) = 100 \cdot \left(1 - \frac{13}{101}\right) = 87.13\%, \quad (6.1)$$

where COV is the *code coverage*; ML_i is the *i-th missed line*; and RL_i is the *i-th executed relevant line*. A line is executed if it contains an *Erlang* expression such as a matching or a function call. A blank line or a line containing a comment, function head or pattern in a case statement is not executable [Erlang 1986].

For a more data-oriented coverage approach, where the input and output parameters were emphasised, a spectrum of possible values were tested. Figure 13 presents all queries and mutations included in the end-to-end test. After the automated specimen life cycle flow test succeeded, a few more cases were evaluated and the total COV increased to 92.0%.

Since there was no *ERP* database in the testing environment, automatic integration tests could't be performed. Manual testing was executed directly in the GraphQL API, where the HTTP *request header* contains a string "authorization" and an encoded token – in the form `Bearer token` – to authenticate requests.

Figure 13 – End-to-end queries, from the GraphiQL interface, used for the integration testing.

```

19 # Corporation Defaults -----
20 ▶ mutation mUpdateCorporationDefaults { ⌵ }
33 # Specimen Settings
34 ▶ mutation mRegisterSpecimenSettings { ⌵ }
60 # Specimens -----
61 ▶ query qGetAllSpecimens { ⌵ }
79 ▶ mutation mRegisterSpecimensBatchCorpDefaults { ⌵ }
90 ▶ mutation mUpdateSpecimensBatch { ⌵ }
121 # Collections -----
122 ▶ query qGetAllCollections { ⌵ }
129 ▶ mutation mRegisterCollectionBatch { ⌵ }
149 # Ruptures -----
150 ▶ query qGetAllRuptures { ⌵ }
163 ▶ mutation mRegisterRupture { ⌵ }
183 ▶ mutation mUpdateRupture { ⌵ }
204
205 HTTP HEADERS (1)
206

```

Source: Author.

7 Considerations and Perspectives

With all backend functionalities regarding the specimen supervision throughout its life cycle implemented, this new feature can provide clear and useful specimen management. Every unit testing made sure that each phase of the specimen operations are correctly executed and generate consistent data.

During the development were applied software quality assurance techniques and a quality control plan was plotted, where product goals and requirements of the application were exposed. Along with that, the ABNT defined Standards were always kept in mind to avoid rework. A *Kartrak Laboratory* development plan was discussed and followed in order to clarify the application's specifications.

Coding best practices and conventions were pursued so as to ensure maintainability and readability for the developers who'll continue this project. The designed tests were able to assure a great code coverage and contemplate as many use cases as possible. A few achievements obtained in this project are:

- Overall quality of the application secured by applying SQC techniques;
- Maintainability, usability and portability were ensured by utilising modern software technologies and techniques;
- Specimen supervision operations made available;
- Solid ground for the application provided by testing routines.

The code coverage of the backend modules is 92.0%, which is higher than the product goal defined in Chapter 4. The end-to-end test going through all the specimen life cycle stages accounted for many use cases of the application.

A tight deadline and a non-existent previous experience in web-development and functional programming made this project a personal challenge for the author. Agile and test-driven-development methodologies were useful tools to prevent the developer becoming adrift while the application progressed.

As the application was built with flexibility in mind, many adjustments and new functionalities can be introduced in the *Laboratory* application, such as:

- Implement queries to provide overview pages for data visualisation;
- Include constraints and warnings to aid the user in abiding by the strict Standards;
- Develop a frontend to integrate the implemented GraphQL queries and mutations with the rest of the automation platform;

- Create the *ERP* database in the testing environment, so as to test the connection between front and backend;

Bibliography

- ABSINTHE. *The GraphQL toolkit for Elixir*. 2017. <<https://absinthe-graphql.org/>>. Accessed: 2019-11-11. Cited on page 39.
- AIELLO, R.; SACHS, L. *Configuration Management Best Practices: Practical Methods that Work in the Real World*. 1. ed. [S.l.]: Addison-Wesley Professional, 2010. ISBN 9780321685865,0321685865. Cited on page 35.
- CHA, S.; TAYLOR, R. N.; KANG, K. *Handbook of Software Engineering*. [S.l.]: Springer, 2019. ISBN 3030002616. Cited on page 36.
- COMMITTEE, A. *ACI 363.2R-11 - Guide to Quality Control and Assurance of High-Strength Concrete*. [S.l.]: American Concrete Institute (ACI), 2011. ISBN 9780870317033, 0870317032. Cited on page 41.
- DAY, K. W.; ALDRED, J.; HUDSON, B. *Concrete Mix Design, Quality Control and Specification, Fourth Edition*. 4. ed. [S.l.]: CRC Press, 2013. ISBN 0415504996,9780415504997. Cited on page 40.
- DEISSENBOECK, F. et al. Tool support for continuous quality control. *IEEE Software*, Institute of Electrical and Electronics Engineers (IEEE), v. 25, n. 5, p. 60–67, set. 2008. Disponível em: <<https://doi.org/10.1109/ms.2008.129>>. Cited on page 33.
- ELIXIR. *Functional language*. 2011. <<https://elixir-lang.org/>>. Accessed: 2019-09-18. Cited on page 38.
- ERLANG. *Cover module*. 1986. <<http://erlang.org/doc/man/cover.html>>. Accessed: 2019-11-21. Cited on page 78.
- GIT. *Version control system*. 2005. <<https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>>. Accessed: 2019-11-12. Cited on page 39.
- GITLAB. *Code collaboration software*. 2014. <<https://about.gitlab.com/>>. Accessed: 2019-11-12. Cited on page 40.
- GRAPHQL. *Query and manipulation language*. 2015. <<https://graphql.github.io/>>. Accessed: 2019-08-23. Cited on page 39.
- HALVORSEN, L. *Functional Web Development with Elixir, OTP, and Phoenix: Rethink the Modern Web App*. [S.l.]: Pragmatic Bookshelf, 2018. ISBN 1680502433. Cited on page 38.
- IAN, S. *Software Engineering, Global Edition*. [S.l.]: PEARSON EDUCACION, 2015. ISBN 9781292096131. Cited 4 times on pages 27, 28, 29, and 35.
- IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. pub-IEEE-STD:adr:pub-IEEE-STD, 1990. 84 p. Cited on page 27.

- ISO 9001:2015. *Quality management systems — Requirements*. pub-ISO-std:adr: ISO, 2015. 29 p. Cited on page 31.
- ISO/IEC 25010:2011. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. pub-ISO-std:adr: ISO, 2011. 34 p. Cited on page 29.
- JEFFRIES, R.; MELNIK, G. Guest editors' introduction: Tdd—the art of fearless programming. *Software, IEEE*, v. 24, p. 24 – 30, 06 2007. Cited on page 37.
- KARTRAK. *Kartrak Automation Concrete*. 2015. <<https://www.kartrak.app/business>>. Accessed: 2019-11-24. Cited on page 25.
- KARTRAK. *Kartrak Automation platform*. 2015. <<https://www.kartrak.app/automation>>. Accessed: 2019-11-17. Cited on page 44.
- KARTRAK. *Kartrak Enterprise Resource Planning platform*. 2015. <<https://www.kartrak.app/erp>>. Accessed: 2019-11-17. Cited on page 44.
- LAURENT, S. S.; EISENBERG, J. D. *Introducing Elixir Getting Started in Functional Programming*. 2nd edition. ed. [S.l.]: O'Reilly, 2016. Cited on page 55.
- LEWIS, W. E. *Software Testing and Continuous Quality Improvement*. 2nd ed. ed. [S.l.]: Auerbach Publications, 2005. ISBN 9780849325243,0-8493-2524-2. Cited 2 times on pages 32 and 34.
- MCCALL, J. A.; RICHARDS, P. K.; WALTERS, G. F. *Factors in software quality assurance*. <<https://pdfs.semanticscholar.org/82a9/18fd83f1c0addb890ef313ff892807a10a11.pdf>>, 1977. Cited on page 49.
- PFLEEGER, S. L.; ATLEE, J. M. *Software Engineering: Theory and Practice*. 4. ed. [S.l.]: Prentice Hall, 2009. ISBN 0136061699,9780136061694. Cited on page 35.
- PHOENIX. *A productive web framework that does not compromise speed or maintainability*. 2016. <<https://phoenixframework.org/>>. Accessed: 2019-09-18. Cited on page 38.
- POSTGRESQL. *Database management system*. 1996. <<https://www.postgresql.org/about/>>. Accessed: 2019-11-02. Cited on page 38.
- QUORA. *What is the ideal value of slump?*. 2018. <<https://www.quora.com/What-is-the-ideal-value-of-slush>>. Accessed: 2019-11-23. Cited on page 41.
- ROTHWELL, D. M. *An Introduction to Relational Database Theory*. First edition, first printing. [S.l.]: McGraw-Hill Companies, 1992. ISBN 0077074823. Cited on page 39.
- RUMBAUGH, J. *The Unified Modeling Language Reference Manual (2nd Edition) (The Addison-Wesley Object Technology Series)*. [S.l.]: Addison-Wesley Professional, 2004. ISBN 0321245628. Cited 2 times on pages 28 and 29.
- SCHULMEYER, G. G. *Handbook of Software Quality Assurance, Fourth Edition*. [S.l.]: Artech House, 2007. ISBN 1596931868. Cited on page 29.
- THOMAS, D. *Programming Elixir: Functional, Concurrent, Pragmatic, Fun*. 1. ed. [S.l.]: Pragmatic Bookshelf, 2018. (The Pragmatic Programmers). ISBN 9781680502992. Cited on page 38.

WAGNER, S. *Engineering and Managing Software Requirements*. [S.l.]: Springer, 2005. ISBN 3540250433. Cited 3 times on pages [28](#), [32](#), and [33](#).

WILLIAMS, B.; WILSON, B. *Craft GraphQL APIs in Elixir with Absinthe: Flexible, Robust Services for Queries, Mutations, and Subscriptions*. 1. ed. [S.l.]: Pragmatic Bookshelf, 2018. ISBN 1680502557,9781680502558. Cited on page [76](#).


Annex

ANNEX A – Overview pages


Figure 14 – Screen holding general information about shipments and collections.

Coleta de Corpos de Prova


Certificado de Resistência Relatório de Moldagem Informar coleta




Total
20



Na obra
02



Coletado
18



Rompido
00

Empresa: 1 - KARTRAK MIL GRAU Período: 22/02/2019 - 26/02/2019 Situação: COLETADO

N. NF	DATA NF	CLIENTE	ENDEREÇO DA OBRA	SITUAÇÃO
5062	22/02/2019	ANTÔNIO RUBIK	CAMPINAS - AV. JOÃO BATISTA MORATO DO CANTO 1161- OBRA VERSALHES GARDEN - MRV ENGENHARIA	Concluído
5027	23/05/2019	FERNANDO DIAFREDO	CAMPINAS - AV. JOÃO BATISTA MORATO DO CANTO 1161- OBRA VERSALHES GARDEN - MRV ENGENHARIA	Na obra
5023	02/08/2019	TUNICO JUNIOR	RUA KARTRAK MIL GRAU, 01	Pendente
5062	22/02/2019	ANTÔNIO RUBIK	RUA SANTA LUZIA, 100	Coletado
5062	22/02/2019	ANTÔNIO RUBIK	CAMPINAS - AV. JOÃO BATISTA MORATO DO CANTO 1161- OBRA VERSALHES GARDEN - MRV ENGENHARIA	Coletado

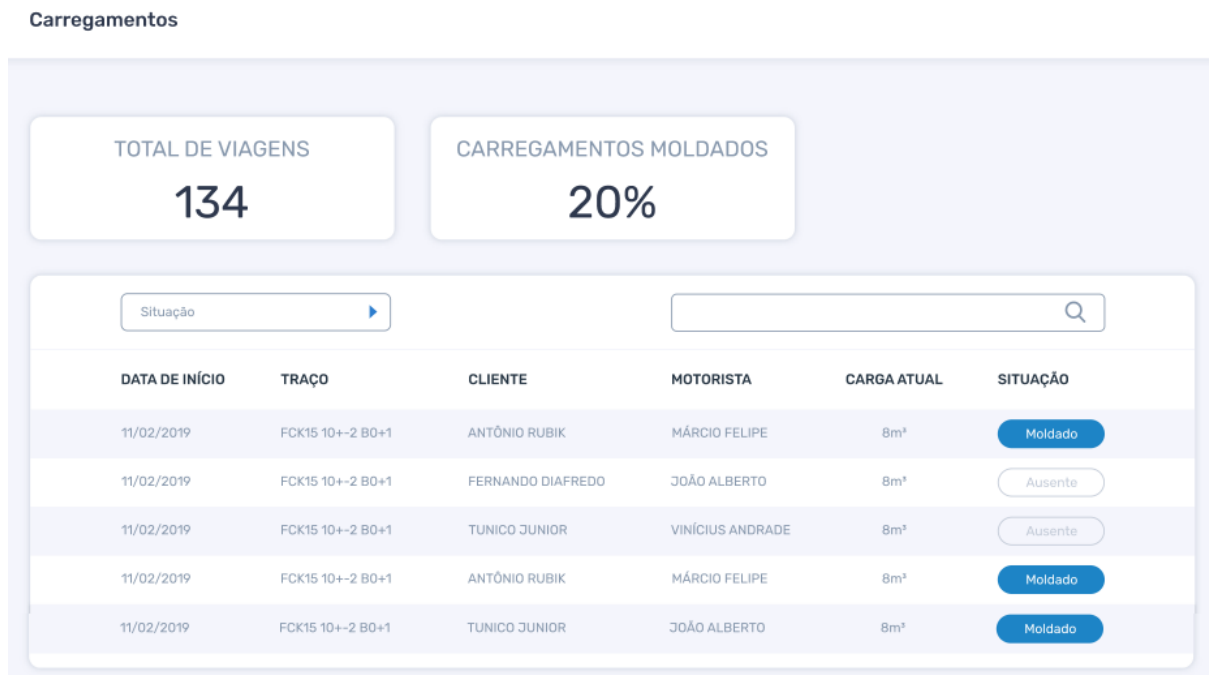
CÓDIGO CP	5062	5062	5062	5062
PRAZO DE RUPTURA	07	07	28	28
STATUS	Rompido	Rompido	Não rompido	Não rompido

→

5062	22/02/2019	ANTÔNIO RUBIK	RUA SANTA LUZIA, 100	Na filial
5023	02/08/2019	TUNICO JUNIOR	RUA KARTRAK MIL GRAU, 01	Concluído
5062	22/02/2019	ANTÔNIO RUBIK	RUA SANTA LUZIA, 100	Coletado

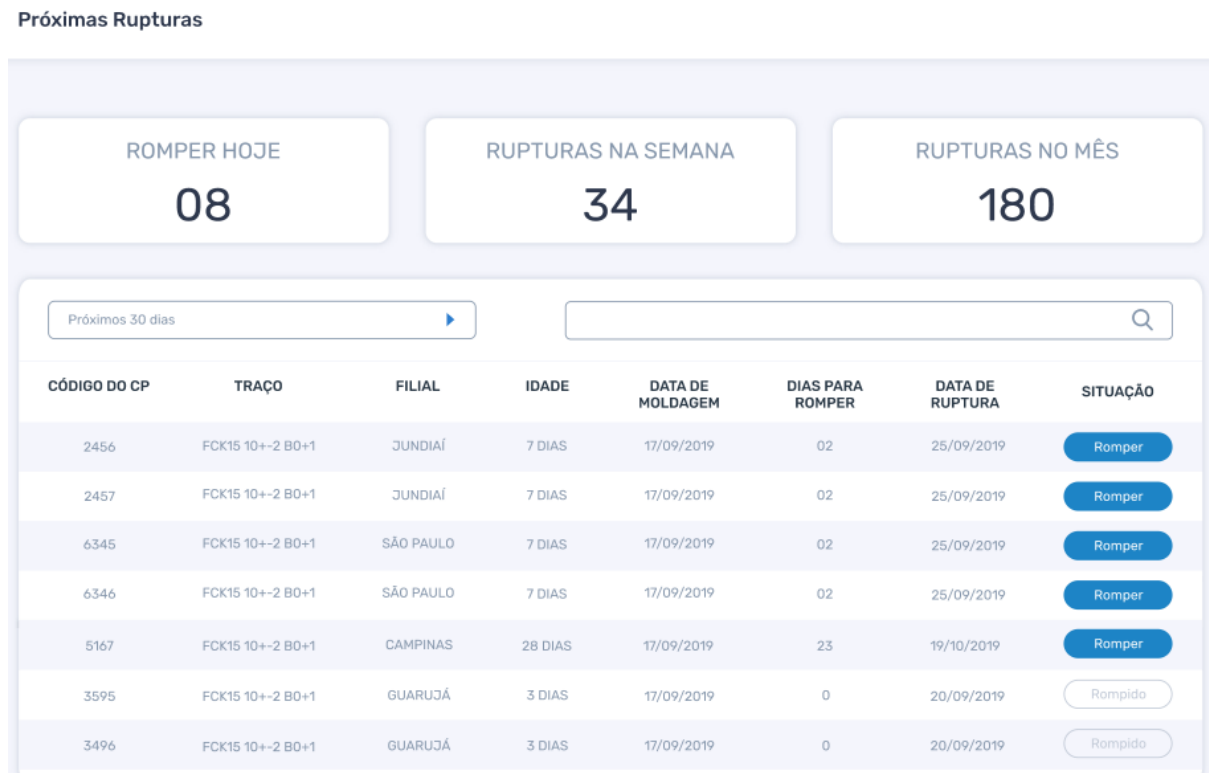
Source: Kartrak.

Figure 15 – Screen containing information about the total percentage of shipments with specimens moulded.



Source: Kartrak.

Figure 16 – Screen with information about ruptures scheduling.



Source: Kartrak.