INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Distributed Job Scheduler

JORGE MIGUEL DE SOUSA CARVALHO
Junho de 2022

POLITÉCNICO
DO PORTO

Distributed Job Scheduler

Jorge Miguel de Sousa Carvalho

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Informatics Engineering, Specialisation Area of Software Engineering

Supervisor: Goreti Marreiros

Porto, June 30, 2022

Dedicatory

To my parents and grandparents, who always supported and encouraged me throughout my academic and personal life. Without them nothing of this would be achieved.

Abstract

Since Cron was released for Unix operating systems in 1975, it became a useful tool for making developers and system administrators' life easier by programming tasks to be launched autonomously.

Although Cron is a simple and powerful tool, it has some problems associated with it, such as lack of visibility, and complexity, because scheduling tasks using crontab's notation can sometimes be difficult.

As times wore on, new approaches of job scheduling systems emerged, most of them providing a user friendly interface to manage jobs/tasks scheduling and reports or statistics about job's execution.

Every day Jumia dispatches millions of marketing campaigns which include emails, newsletters, push notifications, SMS, and other types of channels to engage its customers to visit the e-commerce online store and other Jumia applications.

In Jumia Marketing and Digital Services team's systems a job scheduler is also used, it's called Eye Of Sauron (EOS). EOS is very useful, however it wasn't designed very well when it begun and, nowadays, it's considered a problem for Jumia's business because it's not reliable.

It's Eye Of Sauron's duty to trigger the dispatch process for all the marketing campaigns for Jumia's users, so it needs to be well designed and provide trust to Jumia's business stakeholders.

With this project the problems from the original service were addressed. A new distributed job scheduler named Eye of Sauron v2 was designed and developed. It is composed by several components that are capable of being scaled horizontally and/or vertically. The new system also uses a message broker for asynchronous communication and a relational database as storage solution.

The new job scheduler was considered successful because it was evaluated with a quality percentage of eighty-seven points using a Quantitative Evaluation Framework (QEF) model that considers numerous aspects not only related with functionality, but also with user interface and experience.

Keywords: Job Scheduler, Distributed Systems, Microservices, Databases, Reliability, Fault Tolerance

Resumo

Desde que o Cron foi lançado para sistemas operativos Unix em 1975, este tornou-se uma ferramenta muito útil para facilitar a vida de programadores e administradores de sistemas ao possibilitar o agendamento de tarefas a serem lançadas de forma autónoma.

Embora o Cron seja uma ferramenta simples e poderosa, ele possui alguns problemas associados, como a falta de visibilidade e complexidade, pois o agendamento de tarefas usando a notação do crontab às vezes pode ser difícil.

Com o passar do tempo, surgiram novas abordagens de sistemas de agendamento de tarefas, a maioria delas fornecendo uma interface amigável para promover a manutenção do agendamento de tarefas e relatórios ou estatísticas sobre a execução dessas tarefas.

Todos os dias a Jumia envia milhões de campanhas de *marketing* que incluem e-mails, *newsletters*, notificações *push*, SMS e outros tipos de canais para aliciar os seus clientes a visitar a loja *online* de comércio eletrónico e outras aplicações da Jumia.

Nos sistemas da equipa Jumia *Marketing and Digital Services* também é usado um agendador de tarefas, chamado Eye Of Sauron (EOS), ou "Olho de Sauron". Este sistema é muito útil, porém não foi adequadamente projetado, o que fez com que hoje em dia seja considerado considerado um problema para o negócio da Jumia por não ser confiável.

É dever do Eye Of Sauron chamar o processo de envio de todas as campanhas de *marketing* para os utilizadores da Jumia, por isso precisa fornecer confiança aos executivos da Jumia.

Com este projeto os problemas do serviço original foram solucionados. Um novo agendador de tarefas distribuído chamado Eye of Sauron v2 foi projetado e desenvolvido. É composto por vários componentes que podem ser escláveis horizontalmente e/ou verticalmente. O novo sistema também utiliza um *message broker* para comunicação assíncrona e uma base de dados relacional como solução de armazenamento.

O novo agendador de tarefas foi considerado bem sucedido porque foi avaliado com uma percentagem de qualidade de oitenta e sete pontos usando um modelo *Quantitative Evaluation Framework (QEF)*. Este modelo considera inúmeros aspectos, não apenas relacionados à funcionalidade, mas também com a interface e a experiência do utilizador.

Palavras-chave: Sistema de agendamento de tarefas, Sistemas distribuídos, Micro-serviços, Bases de dados, Confiabilidade, Tolerância a falhas

Acknowledgement

I thank all of those who supported me throughout my academic journey and also the ones that contributed to this dissertation project's development, namely João Granja (Jumia's supervisor), Hugo Martins (Jumia's co-supervisor) and Goreti Marreiros (ISEP supervisor) for all the unconditional availability and help.

Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms

API	Application Programming Interface.
CI/CD	Continuous Integration and Continuous Delivery.
CORS	Cross-Origin Resource Sharing.
CRM	Customer Relationship Management.
CRUD	Create, Read, Update and Delete.
DTO	Data Transfer Object.
EOS	Eye Of Sauron.
FFE	Fuzzy Front End.
FURPS+	Functionality, Usability, Reliability, Performance, Supportability.
HoQ	House of Quality.
HTTP	Hypertext Transfer Protocol.
JSON	JavaScript Object Notation.
JVM	Java Virtual Machine.
MA	Monolithic Architecture.
MDS	Marketing and Digital Services.
MSA	Microservices Architecture.
NCD	New Concept Development.
NPD	New Product Development.
ORM	Object–Relational Mapping.
QA	Quality Assurance.
QEF	Quantitative Evaluation Framework.
QFD	Quality Function Deployment.
RAM	Random-Access Memory.
REST	Representational State Transfer.
SOA	Service-Oriented Architecture.
SQL	Structured Query Language.
SSD	System Sequence Diagram.

SSL	Secure Sockets Layer.
UI/UX	User Interface/User Experience.
UML	Unified Modeling Language.
URI	Uniform Resource Identifier.
URL	Uniform Resource Locator.
UUID	Universal Unique Identifier.

Chapter 1

Introduction

This chapter describes the aspects that led to the development of this dissertation.

It starts with a presentation of the company to understand the business of Jumia, and also gives context about the current solution that is used to schedule jobs to run at a certain date and time.

Finally, the problem, objectives and contribution are described and, in the end, the structure of the document is presented.

1.1 Company Presentation

Jumia is a leading e-commerce platform in Africa that was founded in 2012 in Nigeria. Soon started to expand the business to other locations and, by the year 2022, operates in more than ten African countries including Nigeria, Morocco, Egypt, Kenya, South Africa, among others.

Jumia's marketplace helps millions of consumers to find the best deals and also improves the connection and growth of, not only small sellers but also big and well-known brands.

The company's role is to provide ways for making people's lives easier by helping them shop and pay (with the app JumiaPay) for millions of products at the best prices.

In 2016 Jumia became the first African unicorn being evaluated over one billion USD and, in April 2019, went public on the New York Stock Exchange. Every year Jumia is increasing the number of sales and revenue but, to achieve that, it has to keep and maintain a strong relationship with its customers.

1.2 Context

One of the ways that Jumia uses to engage its customers is through the delivery of millions of marketing communications every day, such as emails, newsletters, push notifications, and SMS, to several segments of users configured by an internal service for each operational country. The content of each one of these communications is put together using an internal platform developed and maintained by the Marketing and Digital Services (MDS) team.

In the MDS platform, Jumia's Customer Relationship Management (CRM) teams can create marketing campaigns to be triggered at a certain scheduled date and time. To do that, MDS team created a campaigns microservice named Butterfly that calls another microservice, a

job scheduler called Eye Of Sauron (EOS), using as reference the service Chronos from Airbnb (AirbnbEng 2013).

A job is the main entity of the domain of Eye Of Sauron. It can be defined as a unit of work to be done. In EOS' logic, a job can execute Hypertext Transfer Protocol (HTTP) or Bash commands. One of the use cases is to do HTTP requests and call an endpoint from the campaigns service to trigger the dispatch of a marketing campaign.

EOS uses a Representational State Transfer (REST) API written in Go that enables HTTP consumers to do Create, Read, Update and Delete (CRUD) operations for the job entity.

Figure 1.1 is a diagram that represents the current implementation of some microservices that are part of the MDS platform. Andromeda is the frontend microservice that consumes an interface from Butterfly for creating and managing marketing campaigns. Andromeda also consumes an interface from Eye of Sauron to fetch the list of available jobs. Butterfly is the campaigns service, as mentioned before, and it consumes an interface from Eye Of Sauron over HTTP to create the jobs to later trigger the dispatch process for the marketing campaigns in Butterfly. Eye Of Sauron consumes an interface from Redis, a NoSQL database, to store its jobs.

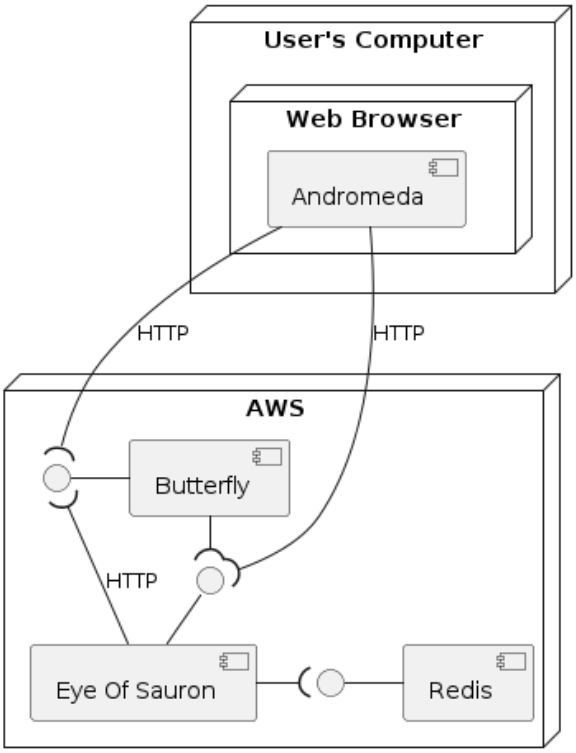

Figure 1.1: Current implementation.

1.3 Problem

When Eye Of Sauron starts up it loads every job from Redis into memory and only then allows to perform CRUD operations. This is considered one of its biggest problems because the number of jobs created grows every day and this service can't scale since, as mentioned,

it manages jobs in memory. To maintain the jobs from being lost, there is a process that frequently persists this data into a file on disk.

Another problem that happens sometimes is related to the service shutting down itself because it runs out of Random-Access Memory (RAM) resources. This is a big issue because there's only a single instance of Eye Of Sauron running in production, as the data integrity isn't guaranteed if more were deployed and both used the same Redis cache.

The service has a huge lack of reliability because when it has a problem and goes down the jobs that were scheduled for that time aren't executed.

Some processes, like the campaigns dispatch, are dependent on Eye Of Sauron and, if the service isn't up and running, the marketing campaigns aren't triggered and Jumia's consumers don't receive the communications that were programmed, which has a big impact on the company's business.

Lack of visibility is another issue. If the service goes down usually isn't easy to know the entire reason why it happened and when it happened without investigating the logs. The users responsible for managing the campaigns are not aware of the problems as the service doesn't report the jobs that were completed successfully or not.

1.4 Objectives

Having in mind all the problems described in the previous section, the main goal of this project is to do a proof of concept and develop a new version of Eye Of Sauron to solve the issues mentioned.

This project wants to achieve a distributed job scheduler capable of launching scheduled jobs across multiple servers and try to accomplish some objectives:

- **Flexibility** - a job can be either periodic or ad hoc (when needed);
- **Consistency** - one graphical user interface used to add and configure jobs on EOS and a shared storage between multiple instances of the service for those jobs;
- **Visibility and Control** - a single point of control to manage jobs and statistics of those;
- **Reliability** - all the jobs scheduled should be executed at the date and time programmed.

1.5 Contribution

As mentioned in the section above, this project's final goal is to obtain a second version of the existing microservice but this time using a distributed architecture.

It's expected that with this development the existing problems should be fixed, which will allow MDS and CRM teams to have more confidence in the process of delivering marketing communications to the customers and reducing the number of errors. Even if some occur, it will be easier to understand why and when it happened.

A job scheduler is a powerful tool since it offers a lot of possibilities not only in e-commerce but also for other kind of businesses as it helps in the automation of processes.

Having such a tool it's good for Jumia's business and helps the company to position itself among its competitors in the e-commerce world. The value analysis of the solution proposed with this project will be detailed in the next chapter.

1.6 Document Structure

The remainder part of this document is structured as follows:

- **Value Analysis** - describes and evaluates the opportunities and value of the solution proposed with this project;
- **State of the Art** - presents the review of the current state of the literature regarding the technologies, programming languages, tools, and other job schedulers that already exist and mentions, from those tools, which are the ones that are used for the implementation;
- **Analysis and Design** - presents the project's functional and non-functional requirements, as well as the domain model proposal and two design alternatives, its evaluation and which one was chosen to implement the service;
- **Implementation** - details the implementation process, mentioning how the project was setup, describing the implementation of the required features, and how the different types of jobs are executed and stored;
- **Evaluation** - describes the hypothesis, the indicators and sources of information, the methodology to evaluate the solution proposed with this project, and what is this proof of concept's final quality percentage;
- **Conclusions** - presents this project's achieved objectives and what are its limitations. It also mentions some aspects that can be addressed in the future to improve the work that was done.

Chapter 2

Value Analysis

Value creation and innovation are some of the most important aspects when evaluating if a business is successful or not.

The decision-making process before investing in a new product can be very difficult for most companies. To help in this process a value analysis can be done.

Value analysis examines the advantages and costs to reduce the expenses without diminishing the quality of the final product. In other words, its goal is to achieve the maximum possible value in a sustainable way.

This chapter describes the value analysis of this project to assess its potential value taking into consideration the context of the problem to be solved.

The concepts that are associated with this topic are also described including the innovation process, the New Concept Development (NCD) and the value that it is generated with the solution proposed with this project.

2.1 Innovation Process

Innovation can be considered as the introduction of something new, for instance, goods, products, or methods, which its end result aims to evolve and keep a business strategy adapted to customers' needs since they are always changing.

According to Koen et al. (2002), the innovation process can be divided into three areas: Fuzzy Front End (FFE), the New Product Development (NPD) and commercialization as can be observed in the Figure 2.1.

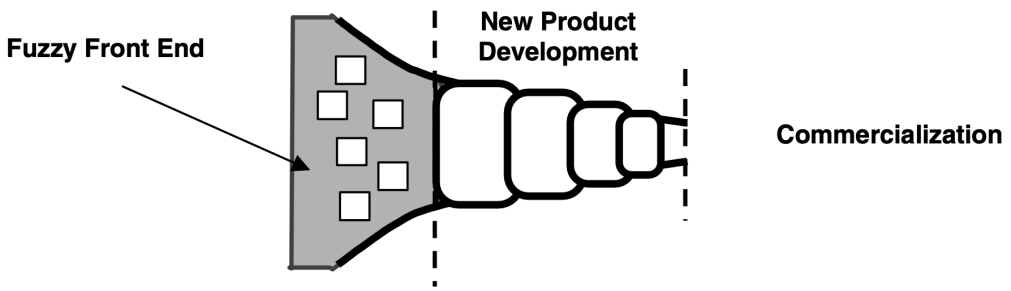

Figure 2.1: Innovation Process Parts (Koen et al. 2002).

Table 2.1: Differences between FFE and NPD. Adapted from (Koen et al. 2002).

Aspects	**Fuzzy Front End (FFE)**	**New Product Development (NPD)**
Nature of Work	Experimental, often chaotic. "Eureka" moments. Can schedule work - but not invention.	Disciplined and goal-oriented with a project plan.
Commercialization Date	Unpredictable or uncertain.	High degree of certainty.
Funding	Variable - in the beginning phases many projects may be "bootlegged," while others will need funding to proceed.	Budgeted.
Revenue Expectations	Often uncertain, with a great deal of speculation.	Predictable, with increasing certainty, analysis, and documentation as the product release date gets closer.
Activity	Individuals and team conducting research to minimize risk and optimize potential.	Multifunction product and/or process development team.
Measures of Progress	Strengthened concepts.	Milestone achievement.

The FFE is commonly seen as "one of the greatest opportunities for improvement of the overall innovation process" since the NPD part of the process already was improved according to Koen et al. (2002).

Koen et al. (2002) also mentions that "many of the practices that aid the NPD portion do not apply to the FFE" because some aspects, such as the nature of work, commercialization date, funding level, revenue expectations, activities, and measures of progress are very different, as shown in the Table 2.1.

Due to the lack of common terms and definitions for key elements of the FFE across companies, Koen et al. (2002) developed the New Concept Development (NCD) model which tries to provide a common terminology for these key elements.

2.2 New Concept Development

To better understand the New Concept Development (NCD) model, Koen et al. (2002) provides the definition of the following words:

- **Opportunity** - "A business or technology gap, that a company or individual realizes, that exists between the current situation and an envisioned future in order to capture competitive advantage, respond to a threat, solve a problem, or ameliorate a difficulty."
- **Idea** - "The most embryonic form of a new product or service. It often consists of a high-level view of the solution envisioned for the problem identified by the company."

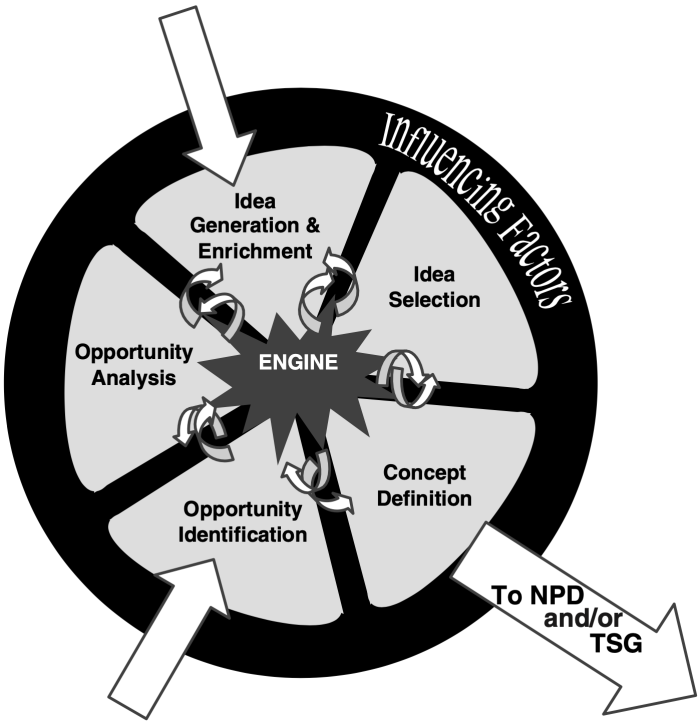

Figure 2.2: New Concept Development (NCD) (Koen et al. 2002).

- **Concept** - "Has a well-defined form, including both a written and visual description, that includes its primary features and customer benefits combined with a broad understanding of the technology needed."

The NCD is a relationship model, not a linear process, as shown by the Figure 2.2, that is composed of three main topics (Koen et al. 2002):

- **Engine or Bull's-eye portion** - This is the center part of the model. Is related with the "(...) leadership, culture, and business strategy of the organization that drives the five key elements that are controllable by the corporation."
- **Inner spoke area** - This part of the NCD model is related with the five controllable elements of the FFE - opportunity identification, opportunity analysis, idea generation and enrichment, idea selection and concept definition.
- **Influencing factors** - These are uncontrollable factors by the organization that are related to the outside world, such as "(...) distribution channels, law, government policy, customers, competitors, and political and economic climate", which affect the innovation process through to commercialization.

The NCD model has a circular shape as can be seen in Figure 2.2. This is meant to "(...) suggest that ideas and concepts are expected to iterate across the five elements" (Koen et al. 2002).

In the same Figure, the arrows pointing inside represent where it all starts - opportunity identification or idea generation - and the exiting arrow shows that concepts leave the NCD model and enter the New Product Development (NPD).

2.2.1 Opportunity Identification

As analyzed in the NCD model, one of the five key elements is opportunity identification. This is a starting point to build one or more concepts that, eventually, can lead to some kind of innovation.

Opportunity identification has, at its essence, the objective of identifying what the organization wants to pursue. It's commonly driven by business goals that could lead to a completely new direction or an upgrade for an existing product. "Overall opportunity identification defines the market or technology arena the company may want to participate in" (Koen et al. 2002).

Taking into consideration the scope of this project and Jumia's objectives, the opportunities identified are:

- Need of a new solution for automation of the marketing campaigns dispatch and other system tasks at a certain timestamp;
- Need of a solution with a good design that allows the evolution of the system in a sustainable and scalable way;
- Need of a solution that is flexible and reliable, avoids errors and promotes visibility and control of the management of the jobs scheduled or executed and the statistics of those.

The opportunities listed above were identified because the current job scheduler, named Eye Of Sauron (EOS), it's starting to manifest some problems and is being the root cause of some issues in Jumia's Marketing and Digital Services team systems.

Having a proper working distributed job scheduler to automate the trigger of marketing campaigns is the key to continue engaging Jumia's customers with discounts and products focused on the clients' needs.

The implementation and adoption of a new job scheduler it's also the key to reducing the amount of time spent by the developers to investigate the cause of errors during execution and fixing or triggering the campaigns manually.

2.2.2 Opportunity Analysis

In the opportunity analysis, are assessed the opportunities identified and if they are worth pursuing.

Before the specific business and technological opportunities establishing, additional information is required. This information is often obtained, for example, with market assessments and scientific experiments.

The effort expended by the organization to try to obtain this information should always be considered and depends on the value obtained when reducing the uncertainties about the "(...) attractiveness of the opportunity" (Koen et al. 2002).

On the project's scope, this is an opportunity that is worth pursuing because its end goal is to replace the existing product with one that allows reducing the time and effort spent fixing recurring issues that happen constantly with the increasing number of marketing campaigns that need to be triggered every day.

Implementing a new job scheduler also allows introducing a new design, such as turning it distributed, with the possibility of having more than one instance running at the same time, something that isn't possible with Eye Of Sauron currently.

Although this is a cost for Jumia that could be avoided initially with a better design, it's a must and the investment now it's allowing the product team and the business stakeholders to evolve the existing features and also create new ones.

2.2.3 Idea Generation and Enrichment

Idea generation and enrichment is the element in the NCD model that deals with the first ideas on how to solve the problem identified by the opportunity.

It's expected that these ideas may have multiple iterations and even new ideas can occur during the process of discussion and brainstorming.

Taking into consideration the project's scope, the following ideas emerged to fulfill the main objective of the work:

- Re-design EOS (the existing micro-service);
- Implementation of a distributed job scheduler from the scratch;
- Use of an open-source implementation that contains all or part of the architecture needed.

2.3 Value of the Solution

This section provides the definitions of some fundamental terms such as value, customer value, and perceived value. It's also in this section where the value of the solution proposed with this project is analyzed.

2.3.1 Value

"The value concept is one of marketing theory's basic elements" (Graf and Maas 2008).

The term value can have different meanings, it depends whether it's considering the consumers or the producer's point of view.

On the producer's side, value is commonly related with its customers and if they are "loyal" to the company in terms of the business perspective, basically if they buy the company's products or services.

From the consumer's point of view, value is often related to the quality of the products or services bought from a company, and the overall satisfaction and relationship with the producer.

When thinking on how to improve the value of a product, some aspects should be considered (Rich and Holweg 2000):

- **Use value** - Concerns about how useful/functional the product is;
- **Esteem value** - Comes from the ownership of the product and the characteristics that are attributed to it by the consumer. These characteristics are things more related to

aesthetics and the subjective value given by the customer to the product or service, not directly linked with its utility.

Another aspect that should be considered is Market value. This is an important element that can be translated as the sum of the use value and the esteem value. Its final result represents what the market/consumers would pay for the product or service.

On this project's scope, the value is related to the final contribution, description, and development of a distributed job scheduler that is supposed to fix the major part of the problems that happen with the current Eye Of Sauron (EOS) micro-service, as described in the previous chapter.

2.3.2 Customer Value

"Identifying and creating customer value (...) is regarded as an essential prerequisite for future company success" and "long-term company survival" (Graf and Maas 2008).

Customer value can be viewed as the general understanding of what the customers really want and value in a product or service, and what can be made by the producer to meet consumers' needs to achieve a competitive advantage over other companies (Graf and Maas 2008).

Graf and Maas (2008) also mentions that the main goal of customer value research is "to describe, analyze, and make empirically measurable the value that companies create for customers and to link these insights to further marketing constructs".

McFarlane (2013) summarizes customer value in the following questions:

- "What do customers really want and how do we meet their demands?"
- "What do customers really value?"

Customer value can be delivered through four components (McFarlane 2013):

- **Service** - Intangible value offered to the consumers;
- **Quality** - The consumers' perception of how well the products/services suit their expectations;
- **Image** - The consumers' perception of the organization that they interact with;
- **Price** - The price that the company wants for their products or services and that the consumers are willing to pay.

In the end, for a company to deliver a superior customer value to its consumers the previous components "must be treated with equal attention and importance" (McFarlane 2013).

On the project's scope, Jumia creates customer value with the marketing emails and push notifications that are sent for each user based on their preferences. These are triggered with the job scheduler in predefined date and time by Jumia's CRM teams, so it's very important to have a reliable solution that is capable of delivering these communications and be able to grow and scale when needed since the number of orders continues to increase each year.

2.3.3 Perceived Value

Ravald and Grönroos (1996) mentions the work of Zeithaml, from 1988, which defines customer perceived value as "(...) the consumer's overall assessment of the utility of a product based on a perception of what is received and what is given" and that "(...) perceived value is subjective and individual, and therefore varies among consumers".

Ravald and Grönroos (1996) also reviewed the work of Monroe, from 1991, that defines "customer-perceived value as the ratio between perceived benefits and perceived sacrifice".

Ulaga and Chacour (2001) considers that "customer-perceived value is a trade-off between benefits and sacrifices perceived by the customer in a supplier's offering".

According to Graf and Maas (2008), perceived customer value "is conceptualized as a trade-off between benefits and sacrifices with a focus on the concrete performance characteristics of the products/services".

Kotler and Keller (n.d.) refer to customer perceived value as "the difference between the prospective customer's evaluation of all the benefits and all the costs of an offering and the perceived alternatives".

As can be observed, as the times wore on, the overall definition of customer perceived value remains and different authors define it almost the same way. The only thing missing is to understand what the authors consider by benefits and sacrifices.

In the words of Ravald and Grönroos (1996), benefits and sacrifices are:

- **Benefits** - "(...) combination of physical attributes, service attributes and technical support available in relation to the particular use of the product";
- **Sacrifices** - "(...) all the costs the buyer faces when making a purchase: purchase price, acquisition costs, transportation, installation, order handling, repairs and maintenance, risk of failure or poor performance".

On this project's scope, the perceived value can be defined as the ratio between the benefits (job scheduler, with a distributed architecture, capable of triggering marketing campaigns and other system's work, more reliable and flexible) and the sacrifices (costs associated with the implementation and maintainability of the proposed solution, such as developer salaries and software/hardware needed).

2.3.4 In-depth Analysis of the Benefits and Sacrifices

According to Woodall (2003), a longitudinal perspective, that considers some aspects of customer value, which changes over time and, therefore, is cumulative, can be made.

This perspective can be divided into four temporal phases/positions, as can be seen in the Figure 2.3, and have the following definitions (Woodall 2003):

- **Ex Ante** - This phase corresponds to a pre-purchase position and implies that customers have some preconditions/ideas about the product/service (value) that they expect to acquire;
- **Transaction** - This phase is related to the customer value that is experienced at the point of sale when buying a product/service;

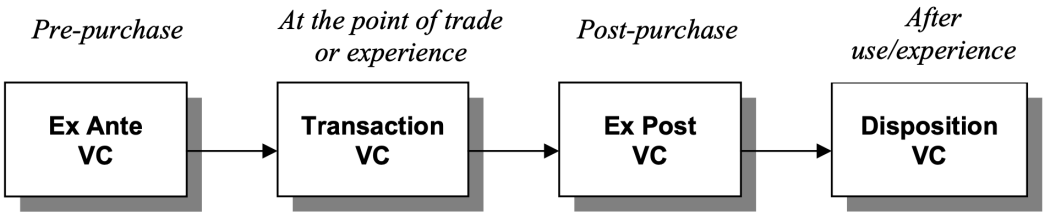

Figure 2.3: Longitudinal Perspective on Customer Value (Woodall 2003).

- **Ex Post** - This phase corresponds to a post-purchase position or moment when the use value of the product/service is experienced;
- **Disposition** - This is the last phase and is related to the value of the product/service after it has been withdrawn from the market.

Table 2.2 presents the benefits and sacrifices associated with each phase of the longitudinal perspective as defined by Woodall (2003).

Table 2.2: Benefits and sacrifices in each phase of the longitudinal perspective.

Phase	Benefits	Sacrifices
Ex Ante	It's expected a solution, with distributed architecture, that allows the triggering of marketing campaigns and other system work.	Money investment and time from the developers to analyze, design, and implement the new system.
Transaction	At the moment where the system is deployed into production and the old one is deprecated, it is expected that the new one should be more reliable, flexible, that promotes visibility and allows the control of jobs scheduled and to be scheduled more easily, and that is also open to changes in the future.	Costs associated with the infrastructure and time for the developers to migrate the old jobs to the new system.
Ex Post	Trust in the execution and triggering of the jobs, fewer errors, and visibility. The time that is saved finding and fixing errors and triggering campaigns manually is used for other tasks.	Costs with infrastructure and software maintainability.
Disposition	The system can be re-designed without much effort to accommodate new features and, if a new one needs to be implemented, this one can serve as a starting point.	Costs for upgrading the system or re-design it. If this one is not capable of satisfying the new needs, it's the cost of implementing a new service from the ground, basically, the same sacrifices as in Ex Ante phase.

As can be understood by Table 2.2, there are benefits to obtain and sacrifices to make in each phase. Most of them are related to the development of the new system, the infrastructure, and the time needed to implement it.

In an early phase, even before the development starts, this system can be considered as an investment since it is expected that allows saving time finding and fixing jobs manually by the developers. It's a game-changer compared with the current Eye Of Sauron (EOS) system running in production.

2.4 Value Proposition

The value proposition can be defined as the benefits that are offered by a company to its customers. Osterwalder and Pigneur (2003) describe value proposition as "(...) the definition of how items of value, such as products and services as well as complementary value-added services, are packaged and offered to fulfill customer needs".

To better assess the value proposition of a company to its customer segment, a graphical tool named Value Proposition Canvas, which was proposed by Osterwalder, Pigneur, et al. (2015), can be used.

This tool breaks the value proposition down into a value map that describes the features of a specific value proposition (products and services, pain relievers, and gain creators), and a customer profile which details a specific customer segment (customer jobs, pains, and gains) (Osterwalder, Pigneur, et al. 2015):

- **Products and Services** - List of all items of value that the company has to offer to its customers;
- **Pain Relievers** - Description of how the products and services alleviate customer pains;
- **Gain Creators** - Description of how the products/services create customer gains;
- **Customer Jobs** - Description of what the customers are trying to achieve or get done in their work/lives;
- **Pains** - Bad outcomes, risks, and obstacles related to customer jobs;
- **Gains** - Description of the outcomes/benefits that the customers want to achieve.

It's said that a "fit" is reached if the value map meets the customer profile. In other words, a "fit" is obtained when the products/services produce pain relievers and gain creators that match one or more of the jobs, pains, and gain of the customers (Osterwalder, Pigneur, et al. 2015).

This project's value proposition canvas is presented in the Figure 2.4. The value map portion of the model is related to the new distributed job scheduler system suggested in this dissertation. On the other part, the customer segment is Jumia itself and its business stakeholders.

The value proposition of this project is a job scheduler, with a distributed architecture, that has a lot of improvements when compared with the current system used (Eye Of Sauron).

The solution relieves the pain caused by the lack of reliability, visibility, and centralized management that currently exists, through the implementation of a new product that is capable of scaling, that doesn't have problems with shared storage for the jobs if more

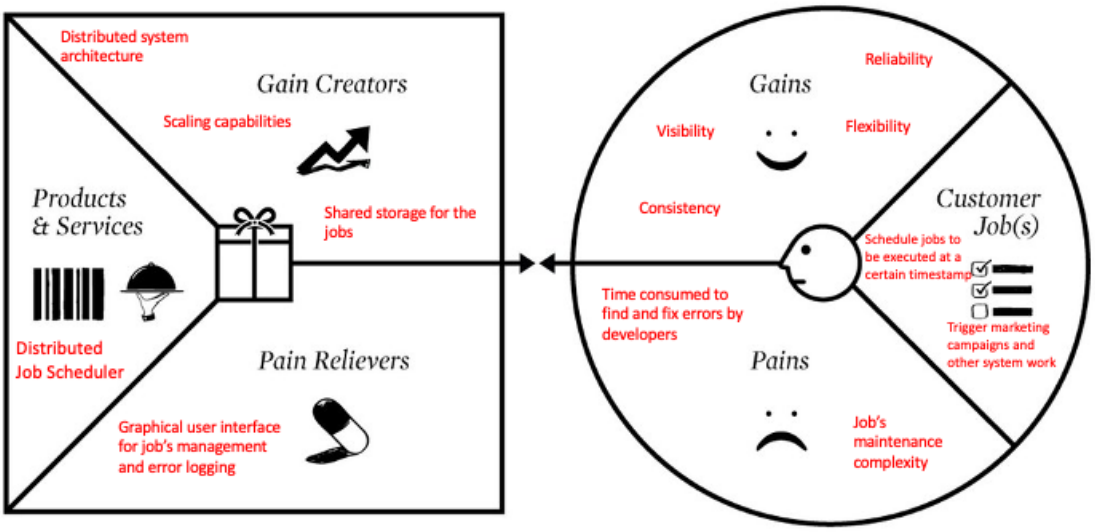

Figure 2.4: Project's Value Proposition Canvas. Adapted from (Osterwalder, Pigneur, et al. 2015).

than one instance of the service is running, and that allows a better comprehension and management of its jobs with a graphical user interface.

2.5 Quality Function Deployment

According to Chan and Wu (2002), Quality Function Deployment (QFD) is "(...) an overall concept that provides a means of translating customer requirements into the appropriate technical requirements".

QFD was developed in Japan in the late 1960s and early 1970s, and it started to be used quickly, during the 1980s, in the United States of America. It was initially proposed to allow the development of products with higher quality since this methodology collects and analyzes the "voice of the customer" and focuses on the customer's needs (Chan and Wu 2002).

This methodology allows the detection of problems at the very early stages of the product development and determines product design specifications ("hows") based on customer needs ("whats") and competitive analysis ("whys") (Chan and Wu 2002).

Based on this project's objectives, described in the first chapter, the customer needs are as follows:

- Flexibility;
- Consistency;
- Visibility and Control;
- Less Errors;
- Reliability of the overall system.

To fulfill the customer's needs, the following technical requirements were defined:

- User interface;
- Storage of the jobs (database);
- Distributed architecture;
- Use of ISO 8601 Notation (for creating the jobs' scheduling);
- REST API for job's CRUD operations;
- API documentation;
- Unit tests;
- Code quality;
- Job's execution statistics.

2.5.1 House of Quality

House of Quality (HoQ) is a tool that is used in the first phase (Product Definition) of the QFD methodology, and "(...) demonstrates the relationship between the customer wants or "Whats" and the design parameters or "Hows"". It allows to "capture a large amount of information in one place" using a matrix format (International 2017).

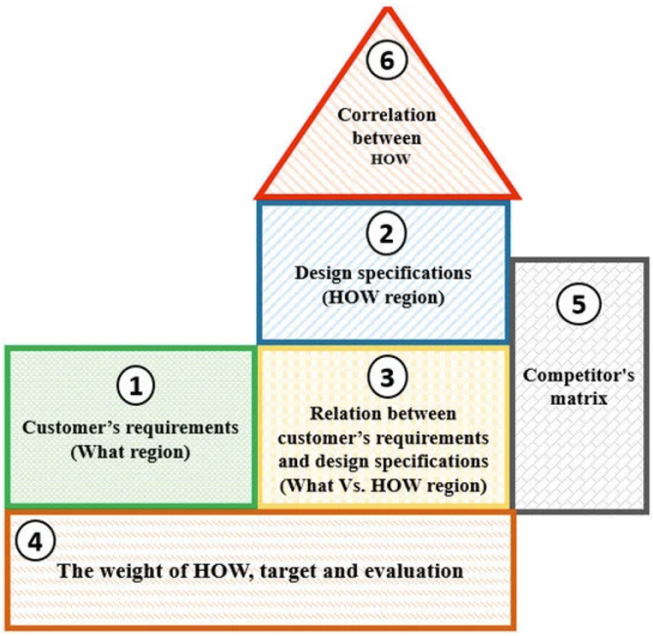

Figure 2.5: House of Quality structure. Adapted from (Abdel-Basset et al. 2019).

According to Abdel-Basset et al. (2019), the main components of the HoQ, and visible in the Figure 2.5 are:

- **Area (1)** - Place where the customer's requirements ("whats") are listed, and also where an importance is given to each one of them according to the priorities of those customers;

- **Area (2)** - Region where the technical requirements (design specifications - "hows") are placed;
- **Area (3)** - Area where is defined the relationship between the customer and technical requirements ("what" vs. "how");
- **Area (4)** - Zone where it's done the sum, row by row, of each priority of the customer requirements multiplied by the relationship value according to the legend of the model;
- **Area (5)** - Area where a comparison of the product and competitors is made, and the evaluation of how much they satisfy the customer's needs;
- **Area (6)** - Region of the model where it's done a comparison of each technical requirement and how much their importance may affect each other.

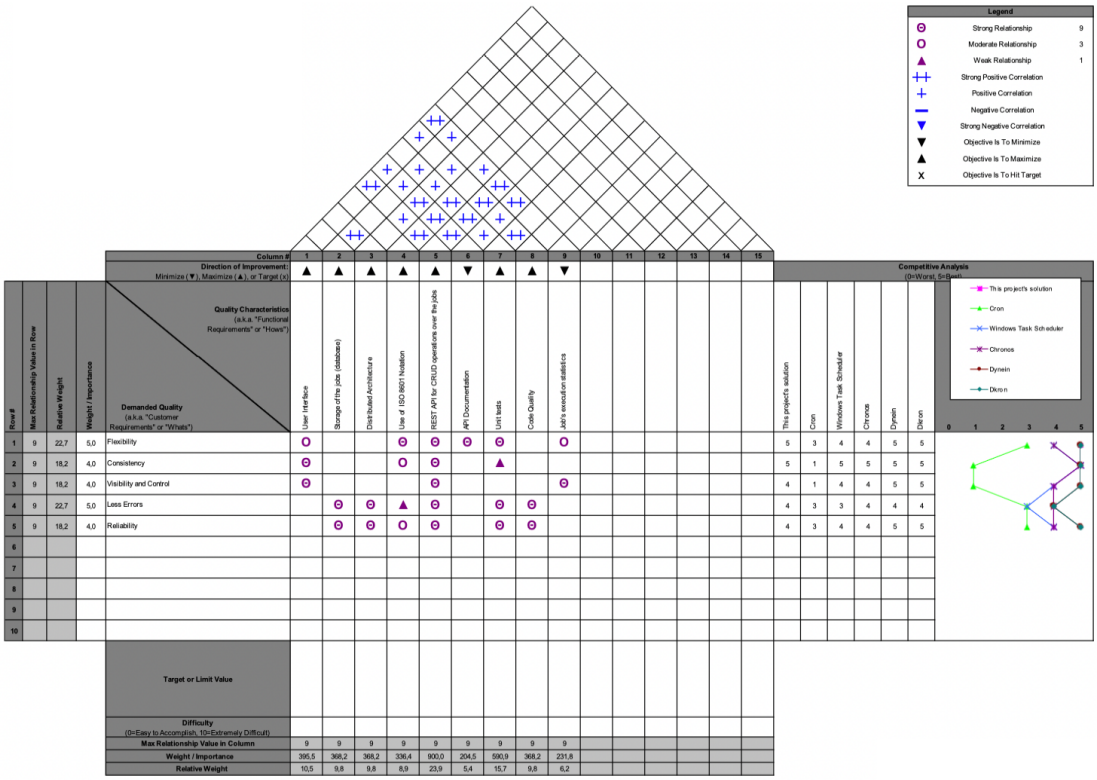

Figure 2.6: Project's House of Quality.

Figure 2.6 illustrates the QFD's House of Quality under this project's scope and using the customer and technical requirements mentioned before, alongside the legend for the symbols used.

The empty cells in areas 3 and 6 mean that there isn't a correlation between "whats" vs. "hows", and between technical requirements, respectively. The competitors (area 5) contains the analysis made to other solutions that are similar to this project's system.

It can be seen that the most important priorities for the customers are the flexibility and the reduction of errors, compared with the current system Eye Of Sauron. The rest of the customer's requirements are equally important between each other, but not as the first two.

The Figure 2.6 also allows understanding the assessment, in percentage, of each technical requirement that needs to be considered during the analyses, design, and implementation of this project's solution. According to the results, the priorities, and the ones that contribute the most to meet the customer's needs, are the REST API for job's CRUD operations, followed by the unit tests and the user interface implementation.

Chapter 3

State of the Art

Before starting the analysis, design and implementation of any project is important to understand the existing work on the topic.

This chapter describes some of the existing solutions about this dissertation project's thematic - a job scheduling system. There are also concepts about monolithic and distributed architectures, the presentation of some programming languages and storage solutions, and the comparison between them.

3.1 Related Work

This section presents a few examples of systems that already exist for jobs' scheduling.

3.1.1 Cron and Crontab

Cron is an utility initially released in 1975 for Unix systems (Kazanov 2020). It was "designed to periodically launch arbitrary jobs at user-defined times or intervals" (Davidovič and Guliani 2015).

The Cron package is composed of a daemon, called crond, that "(...) loads the list of Cron jobs to be run and keeps them sorted based on their next execution time" and "(...) then waits until the first item is scheduled to be executed", "(...) launches the given job or jobs, calculates the next time to launch them, and waits until the next scheduled execution time" (Davidovič and Guliani 2015).

Crontab is an utility that is also delivered in the Cron package, and allows to edit tables of jobs with a specific time format expression as described in Kerrisk (2021) and Kazanov (2020).

Cron has some problems regarding its reliability. Is executed in a single machine and, if that machine is not running, Cron doesn't run and the jobs aren't launched.

Another important aspect about Cron is that it doesn't persist the information about the jobs launched between crond restarts and system reboots, only the crontab configuration file is stored.

Some Cron jobs are idempotent, which means that in case of an error, restart or reboot of the system it is safe to launch the jobs scheduled multiple times. Other Cron jobs cannot be launched twice because they can cause a negative effect, such as sending an email to a large number of users.

Although Cron has some limitations and well-known problems, it is one of the most used utilities by developers, system administrators and other IT personnel due to its simplicity and availability in current Linux distributions.

3.1.2 Windows Task Scheduler

According to Academic (2021), Task Scheduler (formerly Scheduled Tasks) is a job scheduler in Microsoft Windows that was introduced in 1995 in the Microsoft Plus! for Windows 95.

This service allows to schedule any program to run, in a computer running the operating system Microsoft Windows, at a time defined by the user or when a specific event occurs (White and Satran 2019).

The tasks (work to be executed) have some action types as follows (White, Satran, and Batchelor 2021):

- ComHandler Action - "This action fires a COM handler";
- Exec Action - "This action executes a command-line operation such as starting Notepad";
- E-mail Action - "This action sends an email when a task is triggered";
- Show Message Action - "This action shows a message box with a specified message and title".

Windows Task Scheduler is considered a useful and powerful tool, however it has some problems regarding system freezes if the machine is not capable to deal with a lot of tasks triggered at a same time due to lack of available resources.

3.1.3 Chronos

Chronos is a distributed scheduler that runs on top of Apache Mesos and can be used for job orchestration.

This job scheduler allows the execution of sh scripts, or simple commands, and is also able to schedule jobs that run inside Docker containers.

When comparing Chronos with Cron from Unix systems, Chronos has a number of advantages such as the ability of scheduling jobs following ISO8601 with repeating intervals of time, which allows more flexibility in job creation, and also supports the schedule of jobs to be triggered by the completion of others jobs or a job chain (Leibert et al. 2017).

Chronos has some advanced features (Leibert et al. 2017):

- "Write job metrics to Cassandra for further analysis, validation, and party favours";
- "Send notifications to various endpoints such as email, Slack, and others";
- "Export metrics to graphite and elsewhere".

Even though Chronos has a number of capabilities and advanced features when compared with traditional job schedulers, it also has some disadvantages because cannot (Leibert et al. 2017):

- "Magically solve all distributed computing problems";
- "Guarantee precise scheduling";

- "Guarantee clock synchronization";
- "Guarantee that jobs actually run".

3.1.4 Dynein

Dynein is a job scheduling system built by Airbnb in 2019 to power a lot of use cases from delivering in-app messages to dynamic pricing. It was designed with a high capability of scalability on mind, and has become a very important component in Airbnb's business (Fang 2019).

According to Fang (2019), Dynein had to provide some abilities:

- Reliability - Data shouldn't be lose if the system fails or restarts, and it should execute every single job at least once;
- Scalability - Seen as a long-term investment, the system should be able to horizontally scale to support future needs;
- Isolation - A single application's queue with high demand should not affect job processing in other services;
- Time Accuracy - All of the jobs should be scheduled with a maximum deviation lower than 10 seconds from the intended time;
- Efficient Queuing - The job queue should support success or fail acknowledgment for the messages, and the ones that fail should not affect the others, dead letter queues to move failed messages after all retries, and separate worker pools for each consumer;
- Usability - The job scheduling mechanism, that passes through a queue, should be transparent to the developers;
- Unscheduling - Every job should be possible to be un-scheduled at any time.

Dynein divides jobs into two types (Fang 2019):

- Immediate jobs - Jobs that are scheduled to run within 15 minutes. A message is written directly in its final destiny - the third party service queue outside Dynein;
- Delayed jobs - Jobs that are executed to deliver "the right message to the right service queue at the right time". When a job create request reaches Dynein, a message is emitted to an "inbound queue" (write buffer) and then is processed by Dynein which picks the job from the buffer and stores a trigger for the job into the scheduler when has resources available to do it.

Dynein is a job scheduling service, but the actual scheduler is just a component of the service that can be replaced by another. In fact, according to Fang (2019), it was the case for Dynein. In the beginning Quartz, another open-source job scheduler, was used but it wasn't good enough for Airbnb's needs, so they implemented a small and simpler scheduler to replace Quartz. Figure 3.1 represents a diagram of Dynein's architecture.

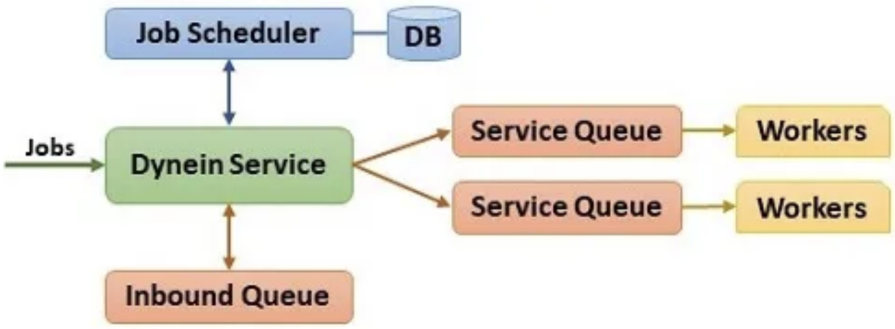

Figure 3.1: Diagram of Dynein's architecture (Stenberg 2019).

3.1.5 Dkron

According to Dkron.io (2015b), Dkron is a distributed job scheduler system that is written in Go and uses Serf, a tool for providing scalability, fault tolerance and reliability, that runs on all current platforms, such as Linux, Mac OS and Windows.

This job scheduler is designed to execute commands in given intervals of time, like Cron from Unix systems, however this is a modern solution, one that prevents problems such as the machine goes down and the scheduled jobs are picked to run in a different instance of the service.

Dkron has some advantages (Dkron.io 2015b):

- Easy to use due to the graphical user interface;
- Reliable since is fault tolerant;
- Has high capability of scaling;
- Can handle with high volumes of scheduled jobs;
- System easy to configure and maintain (Dkron.io 2015a);
- Is a lightweight system.

3.1.6 Comparison Between Solutions

Table 3.1 presents the a comparison between the job schedulers analyzed.

It is possible to understand that Eye Of Sauron, Jumia's existing job scheduler, has some problems when compared with other solutions.

From this project's objectives, and taking into consideration the solutions proposed in this analysis, the systems that are most similar to the intended goals are Dynein and Dkron, so these two will be used as a guide to implement a new job scheduler for Jumia.

Implementing Jumia's own system will allow to add new features in the future if needed, so none of the existing will be used.

Table 3.1: Comparison Between Job Schedulers.

Aspects	Cron	Windows Task Scheduler	Chronos	Dynein	Eye Of Sauron	Dkron
Reliability	No	±(system freezes sometimes)	±(job's execution not guaranteed)	Yes	No	Yes
Persistence of info about job's executions	No	Yes	Yes	Yes	±(stats)	Yes
Simplicity of usage	±(command-line)	Yes	Yes	Yes	Yes	Yes
Extra features available	No	Yes	Yes	Yes	No	Yes
Distributed System	No	No	Yes	Yes	No	Yes
Scalability	No	No	Yes	Yes	No	Yes

3.2 Monolithic and Microservices Architectures

This section presents two architectural styles, monolithic and microservices, and describes some advantages and disadvantages of both of them.

3.2.1 Monolithic Architecture

Monolithic Architecture (MA) is considered the traditional approach to software development. It was used in the past by many companies, like Amazon and eBay, but some companies maintain this type of systems to this day (De Lauretis 2019).

A MA is usually composed of a number of business functionalities encapsulated into a single application, so its modules cannot be executed independently.

This type of architecture is characterized by its high coupling, and all the logic to handle a request runs in a single process (Ponce, Márquez, and Astudillo 2019).

According to Gos and Zabierowski (2020), the systems that follow this traditional approach are "deployed as one single standalone program".

De Lauretis (2019) and Ponce, Márquez, and Astudillo (2019) mention that MA, if not complicated, or with just a small amount of functions, can have some advantages:

- Easiness of development;
- Easiness of testing;
- Easiness of deployment;
- Capability to horizontally scale by running many instances behind a load-balancer.

However, and by the words of Ponce, Márquez, and Astudillo (2019), "once the application becomes large and the team grows in size, this architecture has some drawbacks that become increasingly significant".

Some of the disadvantages pointed out by De Lauretis (2019) and Ponce, Márquez, and Astudillo (2019) are:

- Difficult to understand;
- Difficult to modify;
- Development speed slows down;
- Difficulty to follow Continuous Integration and Continuous Delivery (CI/CD) approaches;
- A small change, that can be just a few lines of source code, needs the entire system to be rebuilt and deployed, which requires more time to introduce new features;
- Scaling becomes difficult "(a monolithic architecture is that it can only scale horizontally)";
- Long-term commitment to the same technological stack (e.g., programming language, database solution, frameworks).

3.2.2 Microservices Architecture

Microservices Architecture (MSA) is a technique for developing software. This technique inherited some of the concepts and principles from the Service-Oriented Architecture (SOA) style, and allows to structure a service-based system as a "collection of very small loosely coupled software services" (De Lauretis 2019).

Although Monolithic Architecture was the most used type of architecture, nowadays many companies started to follow a Microservices Architecture, and a transition from MA to MSA is a topic with a lot of recent studies, such as the ones from De Lauretis (2019) and Ponce, Márquez, and Astudillo (2019).

Figure 3.2 represents an approach to evolve from a MA to a MSA.

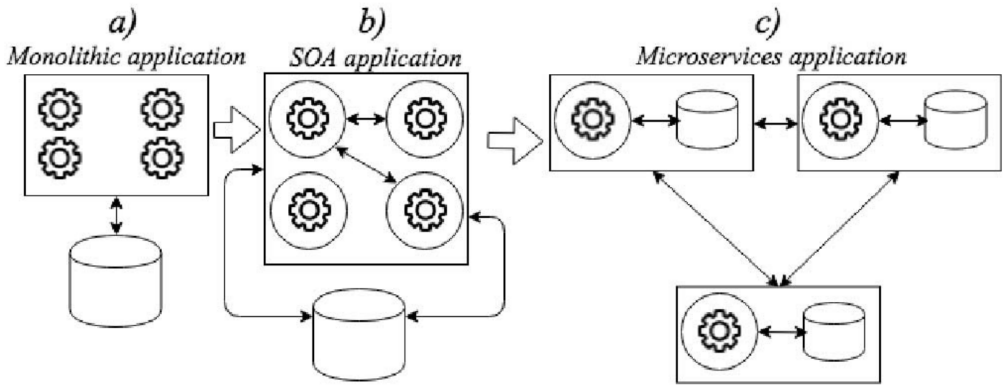

Figure 3.2: Evolution from a Monolithic system to a Microservices Architecture application (Ponce, Márquez, and Astudillo 2019).

Taking an in-depth analysis into the previous Figure, a) is the starting point, a Monolithic Architecture application, which contains a set of business features (represented by the gear symbol) and a single database (illustrated by the cylinder).

If an analysis is performed, using a "separation of concerns mindset", it is possible to start decoupling the monolith into a collection of large services, maintaining the same database structure as seen in b). This is the intermediate step, where the system is considered a traditional SOA application.

If the process of decoupling goes further, usually it's possible to split even more the large services, considering the bonded contexts (purpose) of the features, into small, autonomous, and lightweight services, now considered microservices. Each one of them can have its own database, and they could be connected between them by HTTP requests or with messages produced and consumed in message brokers, for example.

Overall, the Microservices Architecture has a lot of advantages that are worth mentioning (De Lauretis 2019) and (Ponce, Márquez, and Astudillo 2019):

- Flexibility to introduce new features;
- Modularity - It's easier to understand the business logic of the service and change them if needed;
- Maintainability - Breaking the system into small services helps the developers deploying bug fixes more frequently and the testing process is easier;
- Scalability - Microservices can be deployed on different machines, each one with different performances and resources available;
- Fault tolerance - Being small independent services, if they need to be replaced by a better implementation it's easier to do that, and the costs associated are smaller compared with tossing out an entire monolith;
- Reliability - It is a lot better to manage small pieces of a system and fix them if needed than having to deal with an entire, high coupled, system.

On the other hand, microservices architecture also has some disadvantages, such as being more complex because it is a distributed system with more points of failure, there is a chance of observing some errors in the communication between services, and investigating errors is harder if the flow of a certain process involves more than one microservice.

In Jumia Marketing and Digital Services team's systems, a Microservices Architecture is used, and a lot of the benefits previously mentioned are experienced in a daily basis, more than the disadvantages, so the solution proposed with this project will follow the same architectural style.

3.3 Technologies Review

To achieve this dissertation project's main goal, the development of a distributed job scheduler system, it is important to assess some of the technologies that are currently available which allow a proper architecture implementation.

The following subsections provide some information about different programming languages, storage solutions and other relevant tools and frameworks.

3.3.1 Programming Languages

Programming languages are essential to the production of source code that will then be converted to machine level instructions.

There is a large number of programming languages available nowadays, some easier to use and understand than others, but in the end, not all of them are designed to be suitable for backend development because some consume fewer resources and are faster, so these are preferable.

Figure 3.3 shows a ranking of the 15 most used programming languages (for all kinds of usages, not only backend) in February of 2022 and the comparison with the previous year. This data was obtained from TIOBE (2022) which is a company specialized in assessing and tracking the quality of software.

This subsection describes a selection of some of the most currently used programming languages for backend development. The languages considered for this project are C, Java, JavaScript, and Go.

Feb 2022	Feb 2021	Change	Programming Language	Ratings	Change
1	3	▲	Python	15.33%	+4.47%
2	1	▼	C	14.08%	-2.26%
3	2	▼	Java	12.13%	+0.84%
4	4		C++	8.01%	+1.13%
5	5		C#	5.37%	+0.93%
6	6		Visual Basic	5.23%	+0.90%
7	7		JavaScript	1.83%	-0.45%
8	8		PHP	1.79%	+0.04%
9	10	▲	Assembly language	1.60%	-0.06%
10	9	▼	SQL	1.55%	-0.18%
11	13	▲	Go	1.23%	-0.05%
12	15	▲	Swift	1.18%	+0.04%
13	11	▼	R	1.11%	-0.45%
14	16	▲	MATLAB	1.03%	-0.03%
15	17	▲	Delphi/Object Pascal	0.90%	-0.12%

Figure 3.3: Programming languages' usage comparison for February 2022 (TIOBE 2022).

C

C is a programming language that is being used since the 1970s, and programmers still learn it today (Mohn 2022).

It was initially developed by Dennis Ritchie, a computer scientist that worked at Bell Laboratories in the 1970s, and it became an important programming language during the 1980s.

In the twenty-first century, C remains a popular programming language, and even parts of some operating systems, such as Microsoft Windows and Linux, were written using C (Mohn 2022).

Although C is a high-level programming language, it was designed to be simple. "Because the language is so simple, it is very flexible and allows programmers to develop many types of programs and commands" (Mohn 2022).

Some of C's advantages are as follows (Mohn 2022):

- Usability - Some operating systems and web-based programs use the language;
- Portability - It can be used on different machines without having to be recompiled if the other machines use a similar processor architecture;
- Simple and Flexible - Gives programmers a lot of options and functions when using it.

C also has some disadvantages (Mohn 2022):

- Security - Some functions make it possible for malware to penetrate programs written in C;
- Errors - It's easy to introduce bugs into a project that uses C.

Java

Java is a class-based and object-oriented programming language developed by Sun Microsystems in 1995, which was sold to Oracle in 2010 (Gos and Zabierowski 2020).

Java follows the philosophy "write once, run anywhere" because Java's compiler doesn't compile source code into machine instructions but for bytecodes, which are idenpendent from the hardware and the operating system.

After compiling the bytecodes, those files can be interpreted by any machine running an instance of Java Virtual Machine (JVM) which supports the language C++, since Java was developed to follow the same linguistic structure and syntax of C++, the dominant programming language on the market during mid 1990s (Lockhart 2019).

In the twenty-first century, Java is used in numerous formats. It is the primary language used to write programs for the Android mobile operating system, and it still plays and important role when choosing the languages for developing server-side applications.

JavaScript

JavaScript was developed by Brendan Eich in the mid 1990s and is a scripting language, which means that it is interpreted during execution time and it isn't pre-compiled.

Since late 1990s, JavaScript is one of the most popular programming languages to develop applications for the web, and allows a number of features such as asynchronous communications and productivity applications (Chmiel 2019).

According to Chmiel (2019), "(...) while JavaScript takes some of its naming conventions from Java, the languages have little else to do with one another. Unlike Java, which is intended for professional computer programmers, JavaScript was aimed at web designers and other non-programmers".

JavaScript became so popular that powers most of the frameworks for frontend development currently available, like Angular, Vue.js and React.

Nowadays, JavaScript it's also a well used programming language to implement backend applications using Node.js, "an asynchronous event-driven JavaScript runtime (...) designed to build scalable network applications" (Node.js 2014).

Go

Go is a programming language initially developed by Google and released as an open source project in November 2009.

According to Meyerson (2014), Go was created because of the "dissatisfaction with the development environments and languages" that Google was using during late 2000s. The author mentions that none of the tools "(...) had really been designed well; C++ and Java were more than a decade old by that stage" and those environments weren't designed to work at Google's business scale.

Go it's an objected-oriented language that has a different approach when compared with more traditional programming languages, such as Java or C++. For instance, Go doesn't have classes, and the notions of inheritance, and other relevant concepts related with this types of programming languages are a bit different.

Some of Go's advantages mentioned by Meyerson (2014) are:

- Go's objects have less responsibilities and tend to be smaller than objects in other languages;
- Code written in Go tends to be more simpler and reliable;
- Concurrency is one of Go's biggest features;
- Go's garbage collector tends to perform really well, although is not advanced as the ones in other programming languages.

Go also has some disadvantages, such as the dependencies injection and management being more complex than in other languages, and error handling is more unconventional if compared with traditional programming languages, because it doesn't have exceptions and a lot of functions return errors as values.

Programming Language Chosen for Development

Although using Java for developing the system was feasible, this language has some performance issues due to its age and large libraries. Developing in Java also is more time consuming since it's a very verbose programming language, so Java was not considered.

Using C wasn't also considered because it's a verbose language, and the existing libraries are limited.

JavaScript with Node.js was a good possibility, but since in Jumia Marketing and Digital Services team Go is used to implement almost all the microservices, including the original job scheduler Eye Of Sauron, Go was chosen to implement the backend of the solution proposed with this project to avoid increasing the existing tech stack and because it's fast.

3.3.2 Storage Solutions

Most backend systems work with some form of data that is processed and eventually needs to be persisted for future access. This is were databases are useful, as they allow applications to store data and perform operations over it.

There are two big types of databases (Hammink 2018):

- **Relational** - This type of databases emerged during the 1970s, and it allows to store data that can be displayed as tables with rows and columns (schema). The schema of the tables represents fixed attributes and each one of them has a data type. Relational databases use Structured Query Language (SQL) statements to execute some operations over the tables of a database, for example, select, insert, update, and drop. Each table has its own primary key which is used to identify a specific element (row). Data integrity is one of the major concerns in relational databases and, for that, they use constraints to ensure the data stays reliable and accurate.
- **NoSQL** - This type emerged as an alternative to relational databases because the development of web applications increased over time and data became more complex. NoSQL databases don't follow a rigid schema as relational databases, so they allow to store unstructured data. There are some types of NoSQL databases:
    - Key-Value - Simple database systems that only store key-value pairs and allow basic functionality for retrieving data. They are well suited for scenarios where the data isn't very complex and the speed is important;
    - Wide Column - Allow to store data in column families or tables, and each row can be thought as a key-value store. They are designed with the goal of scaling;
    - Document - Store data in the form of JavaScript Object Notation (JSON) documents. Although they are similar to the previous types, NoSQL document databases persist data in a file whose name is the key, and the contents of the file are the values;
    - Graph - Represent data as a network of related nodes or objects to facilitate data visualization and analysis;
    - Search Engines - Are similar to NoSQL document databases but store data without using a fixed schema JSON document and make the unstructured data easily accessible via text-based search strings.

Next are presented some examples of Relational and NoSQL databases.

PostgreSQL

PostgreSQL is an open source relational database that started to be developed in 1986 at the University of California in Berkeley, USA (Postgresql 2022).

It supports a vast number of data types, including JSONB since version 9.4, a data type that accepts unstructured data to be stored, so PostgreSQL is a powerful database management system very used among the software development industry.

PostgreSQL runs on all major operating systems and "earned a strong reputation for its proven architecture, reliability, data integrity, robust feature set, extensibility, and the dedication of the open source community" (Postgresql 2022).

MySQL

MySQL is an open source relational database management system that was initially released in May 1995. It was property of a Swedish company called MySQL AB, but was bought by Sun Microsystems, now Oracle Corporation.

Nowadays, MySQL is used by big companies such as Facebook, Twitter, YouTube, and many more (Mysql 2022).

Redis

According to Redis.io (2022), Redis is a NoSQL, open source, in-memory data structure store that can be used as a database, cache, and message broker.

It provides some features, like storing key-value pairs, among others, and works with data in-memory to achieve the best possible performance but, if needed, data can be persisted periodically to the system's disk.

In Jumia Marketing and Digital Services team's systems, Redis is used as queue for some crucial operations, and it's also the main storage of the original Eye Of Sauron job scheduler.

MongoDB

MongoDB is a NoSQL document database that stores records into JSON documents. It's very strait-forward and easy to use because the "(...) values of fields may include other documents, arrays, and arrays of documents" (Mongodb.com 2020).

There are some advantages of using JSON documents (Mongodb.com 2020):

- Correspond to native data types in many programming languages;
- "Embedded documents and arrays reduce need for expensive joins";
- "Dynamic schema supports fluent polymorphism".

Mongodb.com (2020) also mentions some of MongoDB's advantages:

- High performance data persistence;
- High availability due to data redundancy and automatic failover;
- Horizontal scalability using sharding, a method for distributing data across multiple machines;

- Support for multiple storage engines.

Storage Solution Chosen

One of the biggest issues with the original Eye of Sauron job scheduler was the usage of a NoSQL storage solution, so this category was not considered for the new system, so the go to option is to use a relational database.

Although the usage of MySQL was possible, PostgreSQL is already used in Jumia Marketing and Digital Services team's systems as main database of almost all the existing microservices, so this was the option chosen for storage system of the solution proposed with this dissertation project to avoid increasing the current tech stack.

3.3.3 Message Brokers

When a Microservices Architecture (MSA) is used and two or more services need to communicate between each other there are two ways of doing it, basically utilizing synchronous or asynchronous calls.

In synchronous calls, for example, using a HTTP request, the caller needs to wait for a response before continuing to operate. On the other hand, in asynchronous calls, the caller publishes a message and doesn't wait for a response to continue its work.

Although asynchronous communications require more components to be added to the tech stack, they are very useful when using a MSA, because they have better scaling capabilities, allow to decouple the services, and introduce more flexibility to the overall system.

In Microservices Architecture it's common to use a message broker, which is a system that ensures asynchronous communications between services. Next are presented some popular message brokers (Arguç 2021).

Apache Kafka

Kafka is a message broker created in 2011 by LinkedIn. It is a distributed system "(...) optimized for ingesting and processing streaming data in real-time" (AWS 2022).

This message broker allows to publish and subscribe to streams of records, in which the order of publishing is maintained, or to work as a queue. This is a useful tool to move data from one system to another. Kafka also persists all the messages, so no data is lost even if the system is restarted.

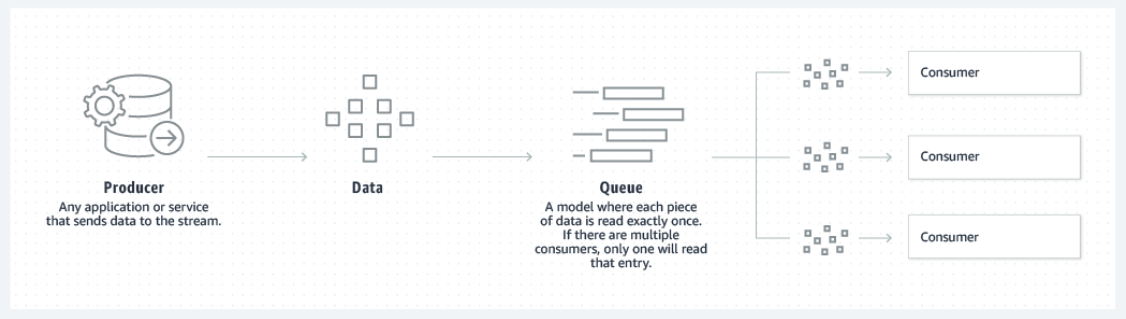

Figure 3.4: Queuing message model (AWS 2022).

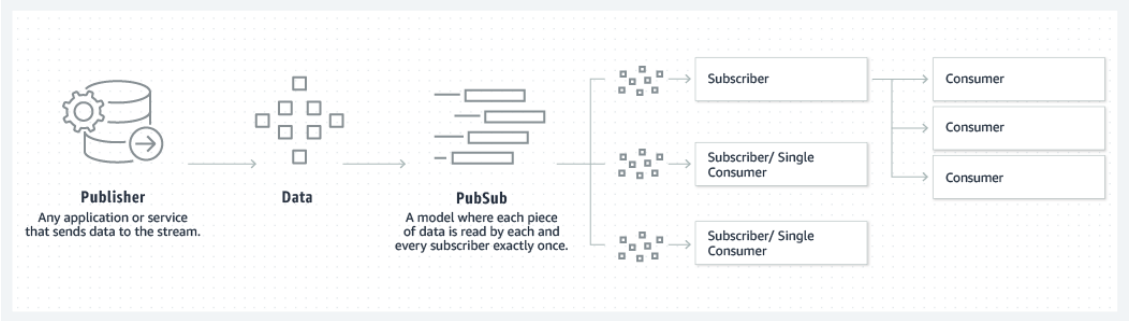

Figure 3.5: Publish-Subscribe message model (AWS 2022).

Figure 3.4 represents a queuing message model and Figure 3.5 a publish-subscribe model using Kafka. The main difference between both is that, in the first, a message is only read by one consumer and, in the second, each consumer reads the same message exactly once.

RabbitMQ

RabbitMQ was released in 2007 and is an open-source message broker that allows applications to connect to each other using asynchronous messages.

This message broker has some advantages such as being reliable, allowing flexible routing, clustering, and tracing (Rabbitmq.com 2022).

According to Arguç (2021), RabbitMQ has some performance issues when in persistent mode.

Redis

Although being used as an in-memory database, Redis also allows to act as a message broker.

Redis is different from other message brokers because it has no persistency, only allows to dump the data into disk, which is what the original Eye Of Sauron does.

Considering the previous point, Redis is "(...) perfect for real-time data processing" that doesn't need to be tracked or maintained (Arguç 2021).

Message Brokers per Use Case

Arguç (2021) mentions the best message broker depending of the use case:

- Kafka - large amounts of data that needs to be stored;
- RabbitMQ - older, but mature, option with lots of features and capabilities that support complex routing;
- Redis - perfect for uses where data persistency is not required, because is very fast.

Message Broker Chosen

Both Redis and Apache Kafka are used in some of the existing services in Jumia MDS. RabbitMQ was never used so, to avoid increasing the current tech stack, and because it's an older technology, this option was not considered.

Redis was already used in the original Eye of Sauron microservice for storage purposes, but this is one of its biggest issues. Arguç (2021) also mentions that Redis is perfect for use cases where data persistency is not required, so this is not the perfect case because it has to work as a message broker. If some kind of problem happens and the service goes down without persisting into disk its contents, they will be lost, so the option chosen for the new Eye of Sauron system is Apache Kafka.

3.3.4 Frontend Frameworks

Nowadays, to provide the best possible experience in user interfaces, where the users see and fell seamless and dynamic interactions, there are available a lot of frontend frameworks that allow to build these kind of projects.

According to Tripathi (2021), in the year 2021 the most popular frontend frameworks are Vue, React and Angular. Figure 3.6 presents the estimated usage numbers.

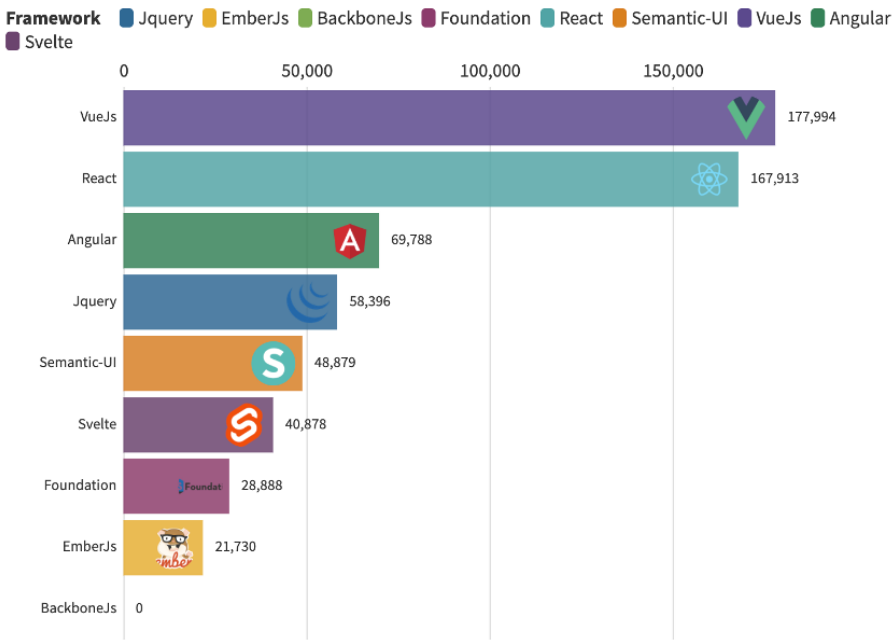

Figure 3.6: Polularity of Frontend Frameworks in 2021 (Tripathi 2021).

Vue.js

According to Tripathi (2021), Vue is the most popular frontend framework in 2021 and is "(...) simple and straightforward".

Vue is a flexible tool, which allows to be used for a lot of purposes. It has some advantages in Tripathi (2021) opinion:

- Extensive and comprehensive documentation;

- Simple syntax;
- Simple to learn for programmers with a JavaScript background;
- Support for TypeScript (JavaScript but with data types).

Tripathi (2021) also mentions some disadvantages:

- Inconsistency of components;
- Small community.

React

In Tripathi (2021) opinion, React is one of the easiest frontend frameworks to learn. It was created by Facebook to address code maintainability problems caused by new requirements introduced in their business.

React's advatages are (Tripathi 2021):

- Allows to create components without classes;
- Reusable components;
- Consistent and seamless performance.

On other perspective, React's disadvatages are (Tripathi 2021):

- Complixities of JSX (React's similar, but not equal, to "HTML") are hardto learn in the begining;
- Documentation lacks some points.

Angular

Angular was created by Google in 2016 and it has a lot of built-in features for frontend solutions.

Angular's advantages according to Tripathi (2021) are:

- Less code to do "big things";
- Usage of dependency injection;
- Reusable components;
- Built-in two-way data binding;
- Vast community;
- Decoupling of components.

The disadvantages are (Tripathi 2021):

- In large scale applications the code structure and size are complex to manage;
- Complex applications sometimes don't perform well.

Frontend Framework Chosen

Vue.js is the chosen option because the existent frontend microservice, called Andromeda, was developed using this framework. Since the users will use the same system to create and manage jobs, it's better to create a new module inside of the existent service and not create another one exclusively for this purpose, so all the other frameworks were not considered just because of this aspect.

Chapter 4

Analysis and Design

This chapter's goal is to present and describe this dissertation project's analysis of functional and non-functional requirements, and to provide two different design approaches to solve this project's problem and objectives. In the end the advantages and disadvantages of both are described and one of them is chosen to be used in the implementation of this project's solution.

4.1 Analysis

System requirements describe not only system's functionalities and goals, but also constraints based on the needs of the clients or project stakeholders. They are divided into two groups: functional and non-functional requirements.

This section presents and describes this project's system requirements. Functional requirements are documented through use cases, and the non-functional follow FURPS+ specification.

4.1.1 Functional Requirements

Functional requirements describe what are the system's functionalities and its behavior. They can be represented trough a use case diagram and system sequence diagrams using Unified Modeling Language (UML) notation. Figure 4.1 presents this project's use case diagram.

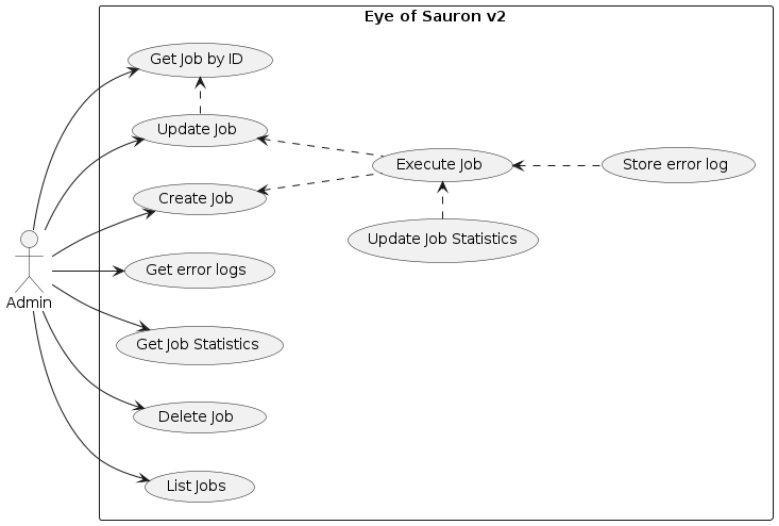

Figure 4.1: Project's Use Case Diagram.

Next is provided a description for each one of the use cases identified.

List Jobs

The system should make available an endpoint on its API to get a paginated list of all jobs with the possibility of applying some filters, for example, next run at, and last run at.

The user interface should have a table with the records fetched from the API with a default descending sort by the job's next run at timestamp, so it will present in the beginning of the table the jobs that will be executed later.

Figure 4.2 is a System Sequence Diagram (SSD) that shows the flow to display to the users the jobs list in the user interface. When they click in the Jobs menu button, the first thing that they should see is the jobs list with descending next run at.

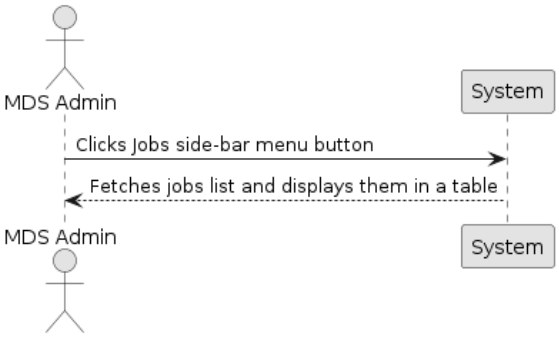

Figure 4.2: List Jobs System Sequence Diagram.

Get Job by ID

The system should provide an endpoint to get the job by ID with its data. In the user interface, when the user clicks in the button to view/edit a job, the system should display a form with the job's available data.

Figure 4.3 is the SSD for this process.

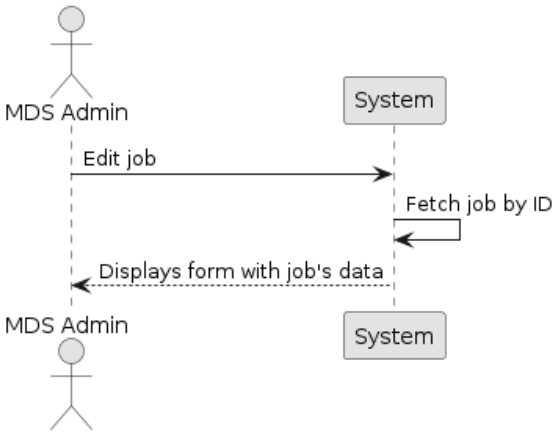

Figure 4.3: Get Job System Sequence Diagram.

Create Job

The system must provide an endpoint to create a job with one of two possible schedule types: immediate or delayed. An user interface should also be created with a form for the users to fill in the required data and call the API to persist it. If it is an immediate job, it should run right after being stored in the database. If it's a delayed job, the users should have the capability to defined a schedule with start date and time following ISO8601 notation, and number of repetitions with an interval of time between them. Figure 4.4 is a SSD that represents the flow of creating a job in the system by the users.

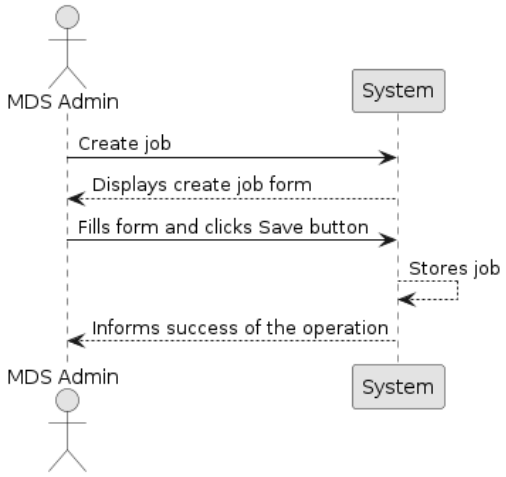

Figure 4.4: Create Job System Sequence Diagram.

Update Job

The system should allow to update an existing job. When selecting the option to edit a job in the user interface, the system should display to the user a form with the existing data. Then, the user should be able to change any value and, when clicking save, the system should store the new data into the database. For this, the system's API must have an endpoint that receives the new job payload and persists it into the database. Figure 4.5 is a SSD that shows the interactions between the user and the system when updating an existent job.

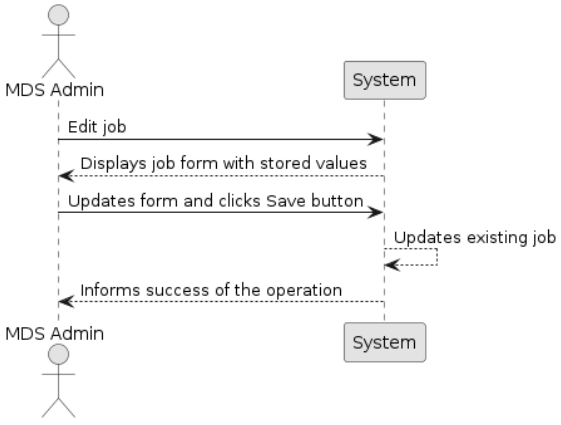

Figure 4.5: Update Job System Sequence Diagram.

Delete Job

The system should allow to delete a job from the database with a call to an endpoint from the system's API. This option must also be available in the jobs list on the user interface.

Figure 4.6 is a SSD that illustrates the process between the system and a user when deleting a job.

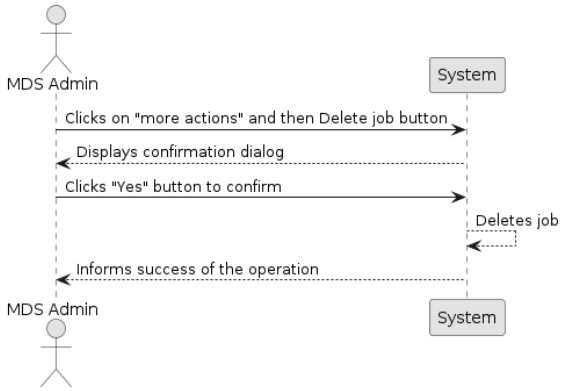

Figure 4.6: Delete Job System Sequence Diagram.

Get Job Statistics

The system should provide an endpoint on its API to get the statistical information from a job given its ID. An user interface should also be created to allow the users to consult that data for a selected job on the list. Figure 4.8 it's a SSD that presents this use case.

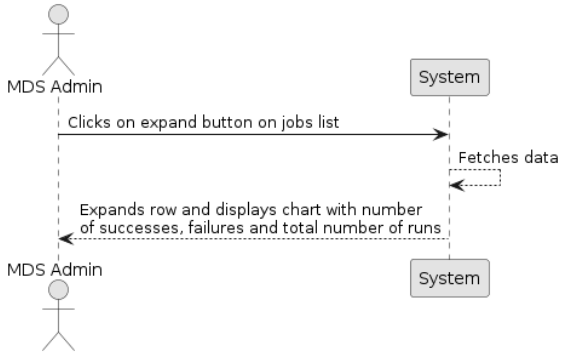

Figure 4.7: Get Job Statistics System Sequence Diagram.

Get Error Logs

The system should provide an endpoint on its API to get a paginated list of error logs available on the database sorted by created_at in descending order, so the most recent errors will appear first on the list. An user interface for consulting that information for a selected job should be created. Figure 4.8 it's a SSD that presents this process.

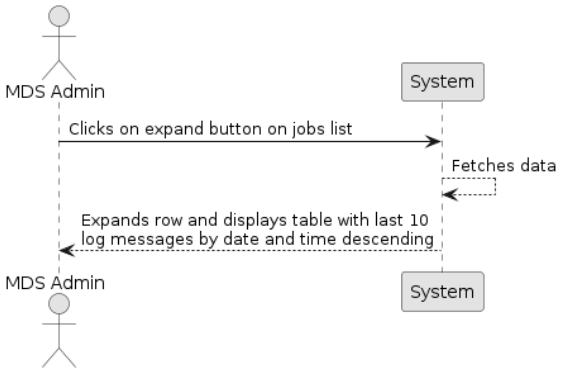

Figure 4.8: Get Job Error Logs System Sequence Diagram.

The following use cases are related with some system internal functionalities, not directly performed by the users.

Execute Job

A job should have one schedule type: immediate or delayed. The system should be able to trigger an immediate or delayed job.

If it is an immediate job, it should be triggered during the create/update process. If it's a delayed job, the system should trigger those jobs when the timestamp stored in the attribute next run at, constructed through the data chosen by the user - start date and time following ISO8601 notation, and number of repetitions with an interval of time between them - arrives.

Update Job Statistics

After a job execution, the system should persist some statistical information, such as the last run timestamp, number of times that the job was triggered, and number of successes or failures regarding the job execution.

Store Error Log

After a job execution, if the job's http or bash command terminates with an unexpected error, the system should persist an error log on the database.

4.1.2 Non-functional Requirements

This subsection presents the system's non-functional requirements, which usually consist of some constraints that should be taken into consideration when designing and implementing a software solution.

Functionality, Usability, Reliability, Performance, Supportability (FURPS+) classification is used for documenting the project's non-functional requirements.

Functionality

- The system should produce auditable information when a job is created, updated and deleted. The Jumia MDS platform user's email that performed the action should be stored in a job_audit table on the database, and also the attributes of the job in that timestamp;
- All the features related with jobs should only be allowed to Jumia MDS users with admin permissions to perform these actions.

Usability

- Jumia MDS platform's job section should follow the same User Interface/User Experience (UI/UX) for table listings and forms as exists in the other menus.

Reliability

- Errors with job's execution should be limited to as few as possible;

Supportability

- The system should be implemented using "clean code" to allow easy maintainability;
- The system should be designed and implemented with maximum capability to scale without being restricted by none of its modules.

+ (Others)

- **Design Constraints**
  - The usage of design patterns like GRASP and SOLID is recommended.
- **Implementation Constraints**
  - Only open-source libraries can be used if needed.

4.1.3 Domain Model

This subsection presents this dissertation project's domain model using an UML diagram that contains the system's entities and their relationships as can be saw by Figure 4.9.

The main entity is Job, and it represents a task or work to be executed. Each Job has a Statistic that contains, for instance, the information about the last execution, and number of succeeded/failed executions. The last entity is JobLog, this entity has the purpose of being used as a history record for error messages that occurred from a job's execution. A Job also contains some attributes, classified here as enumeration types, namely its Status and Type.

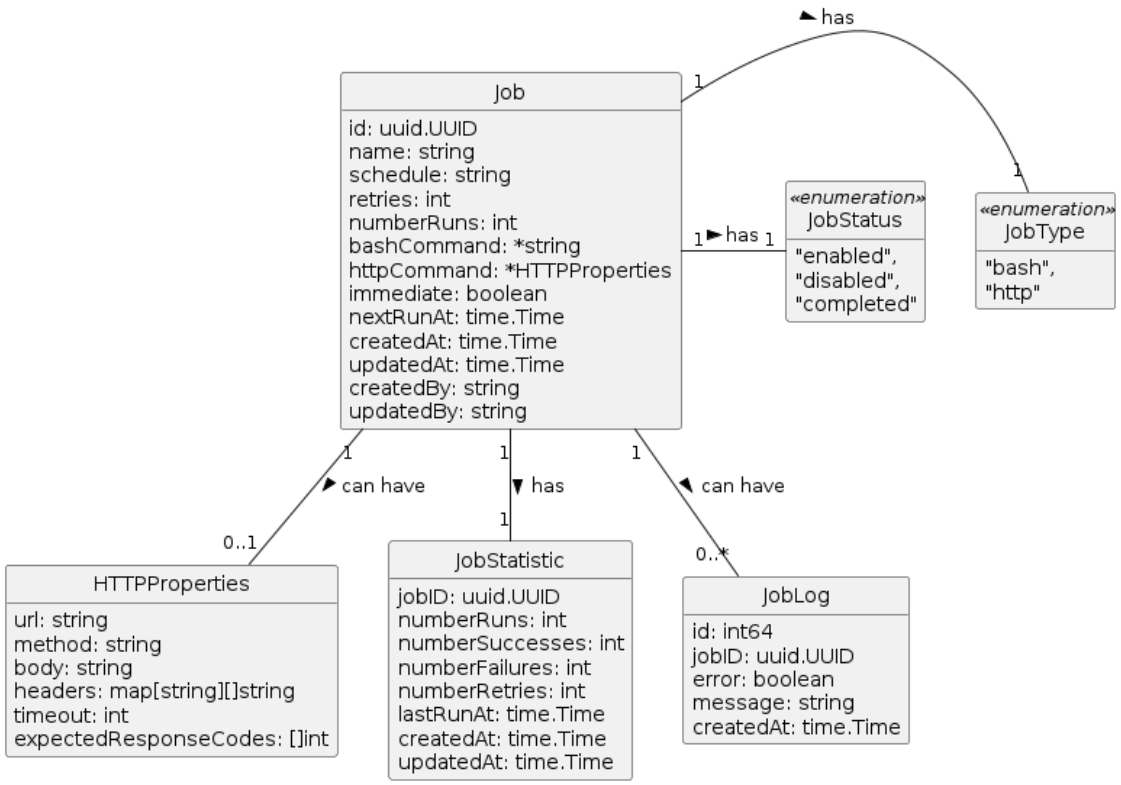

Figure 4.9: Project's domain model.

4.2 Design

This section has the objective of describing two design approaches for the development of the system proposed with this dissertation project.

In the end it's selected the alternative that is used for the implementation based on the advantages and disadvantages of each approach.

4.2.1 First Design Alternative

Figure 4.10 suggests the first design approach for developing this project's system.

This architectural alternative consists of three main components, the API, a Watcher, and a Consumer.

The API contains a HTTP server that aims to receive requests to create, update and delete Jobs, as well as obtain statistics and error logs to be made available to users in the user interface.

The second main component is a Watcher, which is a consumer that searches, through a configurable time interval, for example, ten seconds, all jobs that have their next run at scheduled in the last ten seconds when this request is made to the database. After that, for each job is created a message that is inserted in a Kafka topic, so this component also acts as a producer.

There is another consumer, the Jobs Consumer, that will read the jobs from the topic one by one and execute the http or bash command configured in those jobs.

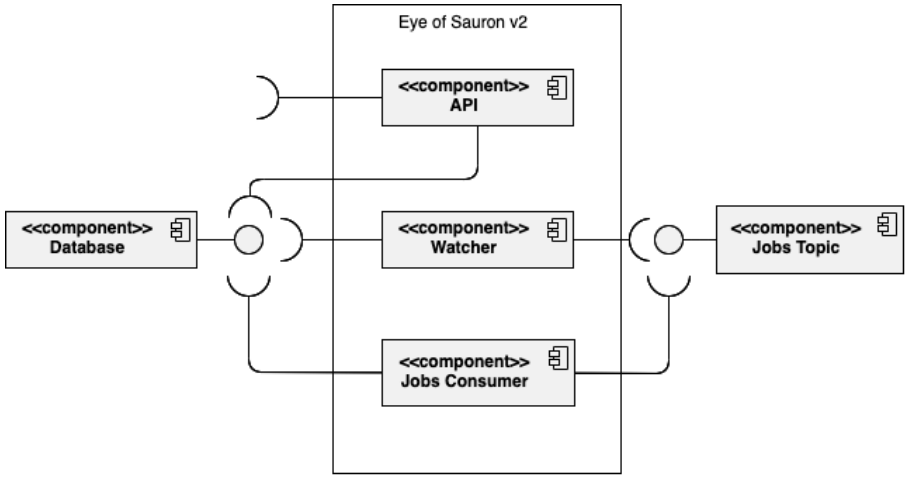

Figure 4.10: First design alternative.

4.2.2 Second Design Alternative

Figure 4.11 presents the second design alternative to implement the solution proposed with this project.

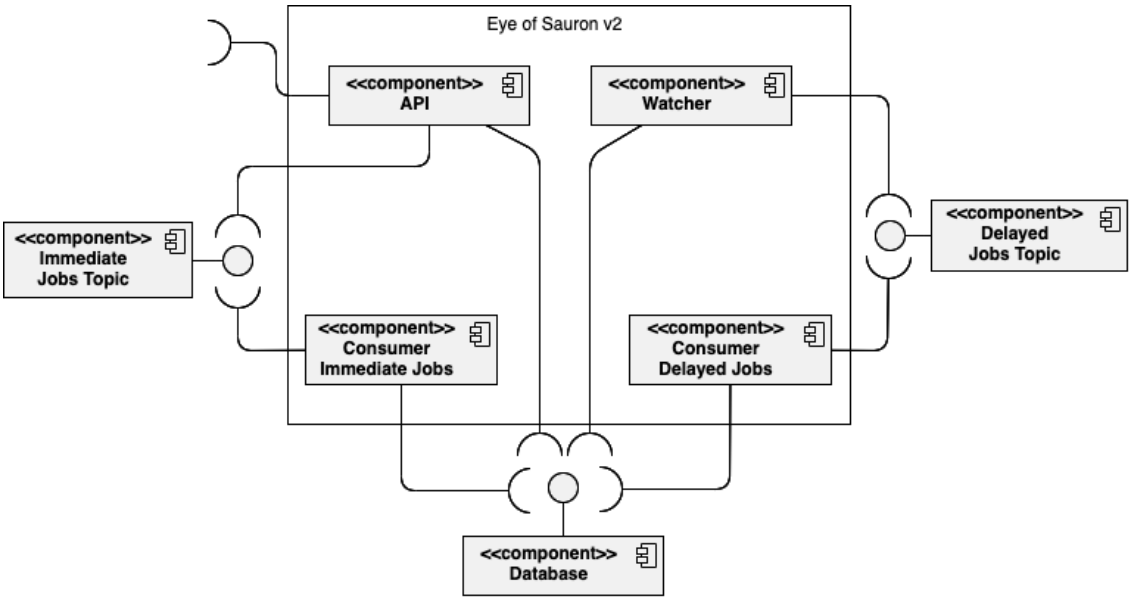

Figure 4.11: Second design alternative.

Analyzing the figure, it is possible to see that this architecture is more complex compared to the first one because is based on the scheduling types for a job - immediate and delayed.

This design has more elements, not only the API and a Watcher with the same purposes as the architecture presented in the previous subsection, but also two Kafka topics, one for immediate jobs and another for delayed jobs.

There are also two types of consumers, based on the same division by job schedule type, which will read the messages (the job objects) from the topics and they will execute their command.

4.2.3 Design Alternatives Comparison

This subsection aims to compare the different architectural alternatives from the previous subsections 4.2.1 and 4.2.2 and identify the one used for the implementation of the system proposed in this dissertation project.

It can be said that the two architectural alternatives meet the requirements in terms of being able to scale. Both alternatives are based on a distributed architecture that uses a REST API to create, update and delete jobs that are then stored in a relational database. They also make use of a message broker for asynchronous communication and execution of the job's command.

The components of both alternatives can also be replaced more easily when comparing with the original Eye of Sauron. For example, the database can change and the only thing that is necessary to do in the source code is to implement another persistence/data access layer to communicate with a different storage solution.

With the use of an architecture based on the elements mentioned above the reliability problems of the system are significantly reduced compared with the original system. This happens because these components can be scaled individually.

The system can scale horizontally by adding more instances of the consumers to read the jobs from the message broker, because the management and persistence of jobs is no longer done in memory using Redis, but with a relational database.

The API, the Watcher, and the consumers can also be scaled vertically by adding more available resources like RAM and CPU to accommodate existing and future individual needs without affecting the whole service.

After evaluating both alternatives, the approach that was chosen for the implementation of this service was the second one, since the design matches more the domain of the problem and its needs.

When observing the design of the second approach it's evident that a division between immediate and delayed jobs exists. This way, the context of the problem is also reflected in the system's architecture and makes future development based on the type of job schedule easier depending on new requirements or expected behavior, for instance, making immediate jobs only available to certain internal system tasks, or to block immediate jobs from being used in marketing campaigns, and force the users to plan even more the contents for a future date, and not in less than a few hours.

Chapter 5

Implementation

This chapter's purpose is to present and describe how the solution chosen for this project in chapter 4 was implemented.

In the first and second sections, a brief description of the overall plan for the implementation of the project after its design in the previous chapter is made.

In the third section, the implementation of the backend microservice, the new distributed job scheduler, named Eye of Sauron v2, is detailed, and in the last section, its possible to find the user interface implemented to support the new functional requirements.

5.1 Three-tier Architecture

According to IBM (2020), three-tier architecture "is a well-established software application architecture that organizes applications into three logical and physical computing tiers: the presentation tier, or user interface; the application tier, where data is processed; and the data tier, where the data associated with the application is stored and managed".

Regarding the benefits of using a three-tier architecture it's possible to say that because each tier runs on its own infrastructure, each tier can be developed simultaneously by a separate development team, and can be updated or scaled as needed without impacting the other tiers. Other benefits include improved reliability, because an outage in one tier causes less impact in the availability or performance of the other tiers, and improved security, as the presentation tier and data tier can't communicate directly, a well-designed application tier can function as a sort of internal firewall, preventing SQL injections and other malicious exploits. (IBM 2020).

The design for this system follows a three-tier pattern, where the presentation tier is represented by the HTTP client that the users will use (a web browser on their computer) running Andromeda (frontend microservice), the logic tier is the application server which includes the new distributed Eye of Sauron job scheduler, and finally, the data tier is represented by the PostgreSQL database that stores job's data and allows to query it. Figure 5.1 represents a generic three-tier design.

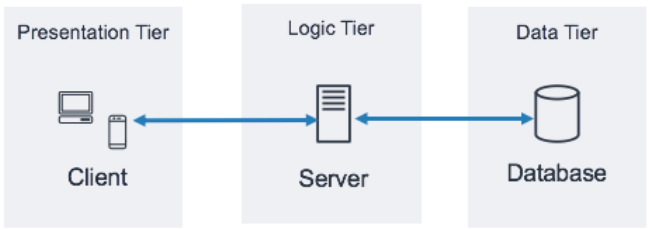

Figure 5.1: Generic Three-tier Architecture (Baird et al. 2022).

5.2 Layered Architecture

The existing and the new developed system uses a layered architecture. Figure 5.2 shows a generic layered architecture and how the multiple layers are split and the common associations between each one of them.

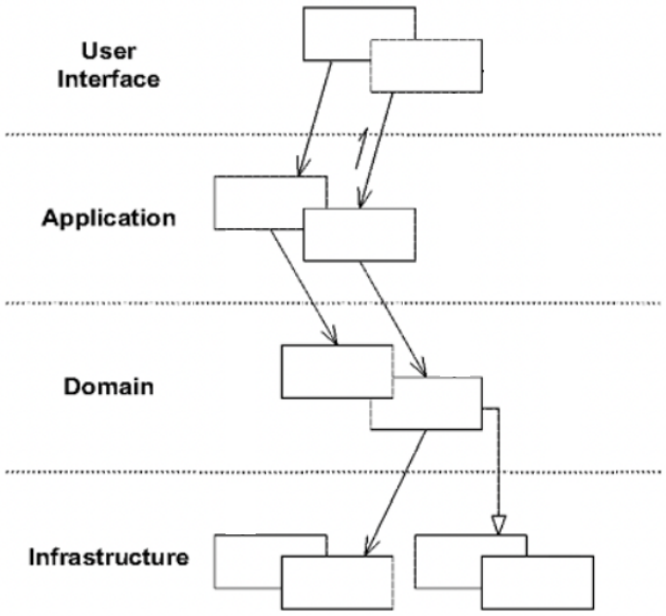

Figure 5.2: Layered Architecture. Adapted from Evans and Wesley 2003.

The next list describes the purpose of each one of the layers visible in Figure 5.2 according to (Evans and Wesley 2003):

- **User Interface (or Presentation Layer)** - Responsible for sending information to be shown to the user, and for interpreting the user's inputs;
- **Application Layer** - Defines the tasks the software is supposed to do. "It does not contain business rules or knowledge, but only coordinates tasks and delegates work to collaborations of domain objects";
- **Domain Layer (or Model Layer)** - Represents concepts of the business and its rules. "This layer is the heart of business software";

- **Infrastructure Layer** - Supports the other layers by providing technical capabilities, such as persistence for the domain, message sending for the application, and so on.

In the context of this project, it's possible to say that the controllers belong to the presentation layer, the services to the application layer, the model objects to the domain layer, and the repositories, and Kafka producers/consumers to the infrastructure layer.

5.3 Distributed Job Scheduler Implementation

This section details the backend service, the distributed job scheduler, that was named eyeofsauron-v2. It starts by giving some insights on how the project was created and configured, its organization, code decisions and implementation, then it's possible to find a description of the REST API endpoints created, what are the producers and consumers for asynchronous communication in this project, how the different types of jobs (HTTP and Bash) are executed and, finally, the description of the communication with the database and some aspects related the reverse-proxy used and with the virtualization of the service.

5.3.1 Project Setup and Packages Organization

In this subsection it's possible to find some of the aspects taken into consideration during the setup of the backend part of the project, namely which are the packages used and what is the purpose of each one of them.

Every Go project is different. There are many layout patterns in the Go ecosystem that can be adopted, some being more popular than others (golang-standards 2022). Next is described what is the adopted layout for the distributed job scheduler related with this dissertation project.

According to Quest (2017), two of the most common packages that is possible to find in Go projects are cmd/ and pkg/. Since these are also two of the most used packages in Jumia Marketing and Digital Services' projects, this was the choice for this one as well.

Figure 5.3 shows the packages used in the project and other relevant files.

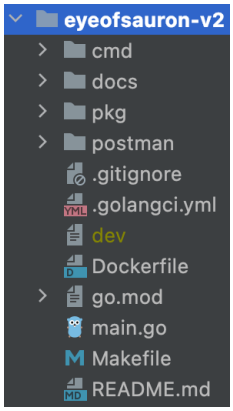

Figure 5.3: Project's chosen layout.

The following list has the objective of provide a description for each one of the packages shown in Figure 5.3:

- **cmd -** This package holds the files for the application's commands such as, the REST API with the available endpoints, the immediate and delayed jobs workers, and the jobs watcher;
- **docs -** Contains a README file with the documentation created for the project;
- **pkg -** Has the main code of the microservice, namely the database models, utilities, migrations, configuration files, interfaces, job's and workers' business logic;
- **postman -** Package with the Postman collection created to test the endpoints developed.

5.3.2 Jobs REST API

This subsection presents and describes the endpoints of the REST API that were created to support the functional requirements described in 4.1.1.

Create Job

This endpoint allows to create and store a new job into the system. It uses the HTTP method **POST** and the route **/jobs**. Following there's an example payload to create a HTTP type job.

```
{
    "name": "job1",
    "schedule": "R2/2022-06-20T14:05:16/P0Y0M0DT0H5M0S",
    "retries": 1,
    "status": "enabled",
    "type": "http",
    "http_command": {
        "url": "https://www.google.pt",
        "method": "GET",
        "body": "",
        "headers": {},
        "timeout": 10,
        "expected_response_codes": [200]
    },
    "immediate": false,
    "created_by": "john.doe@email.com"
}
```

Listing 5.1: JavaScript Object Notation (JSON) payload to create a HTTP type job.

The previous example contains:

- **name -** Unique name for the job;
- **schedule -** Follows ISO 8601 notation and can be split into three parts: Number of times to repeat/Start Datetime/Interval Between Runs.

  R starts the string and, if it's not succeeded by a number, it means that the job will repeat forever, otherwise, if it has the form R2, for instance, this means that the job will run one time in the start datetime and then it will be repeated twice with an interval of time between runs defined in the last part of the string.

The start datetime should have the format 2022-06-20T14:05:16 and it should be in UTC timezone.

The interval between runs starts with a P and follows this format P0Y0M0DT0H5M0S, where 0Y means "zero years", 0M is "zero months", 0D is "zero days", 0H means "zero hours", 5M is "five minutes", 0S means "zero seconds", and T splits years, months and days from hours, minutes and seconds.

In this example, the job will start and run the first time in June 20, 2022 at 14:05:16 UTC and it will repeat after five minutes and, finally, after ten minutes from the start datetime;

- **retries -** Number of times to repeat the command in case of failure in the first execution;
- **status -** Job status. Should be one of the following: enabled, disable, complete;
- **type -** Job type. It determines if the command to be executed is http or bash;
- **http_command or bash_command -** The attribute that holds the data to be executed. The payload should have one or another.

  The http_command object has:

  - *url* - Uniform Resource Locator (URL) of the HTTP request;
  - *method* - One of the following HTTP methods: GET, POST, PUT, PATCH, DELETE;
  - *body* - If the method is POST, PUT or PATCH, this is the request body for the HTTP request;
  - *headers* - A key/value object, where the key is the header to include in the HTTP request;
  - *timeout* - Timeout used to wait for a response;
  - *expected_response_codes* - An integer array holding the expected response code(s).

  The bash_command attribute has as value a string holding the command, for example *ls -l* ;

- **immediate -** Boolean value. True means that it's an immediate job and should be executed right after being created in the system. False means that this is a delayed type job which is going to be executed at the start datetime scheduled;
- **created_by (or updated_by when calling the endpoint to update a job) -** Email from the user who creates/updates the job.

In this particular case, when describing the interactions between the layers inside Eye of Sauron to create a new job, Figure 5.4 shows a sequence diagram that allows to understand it better.

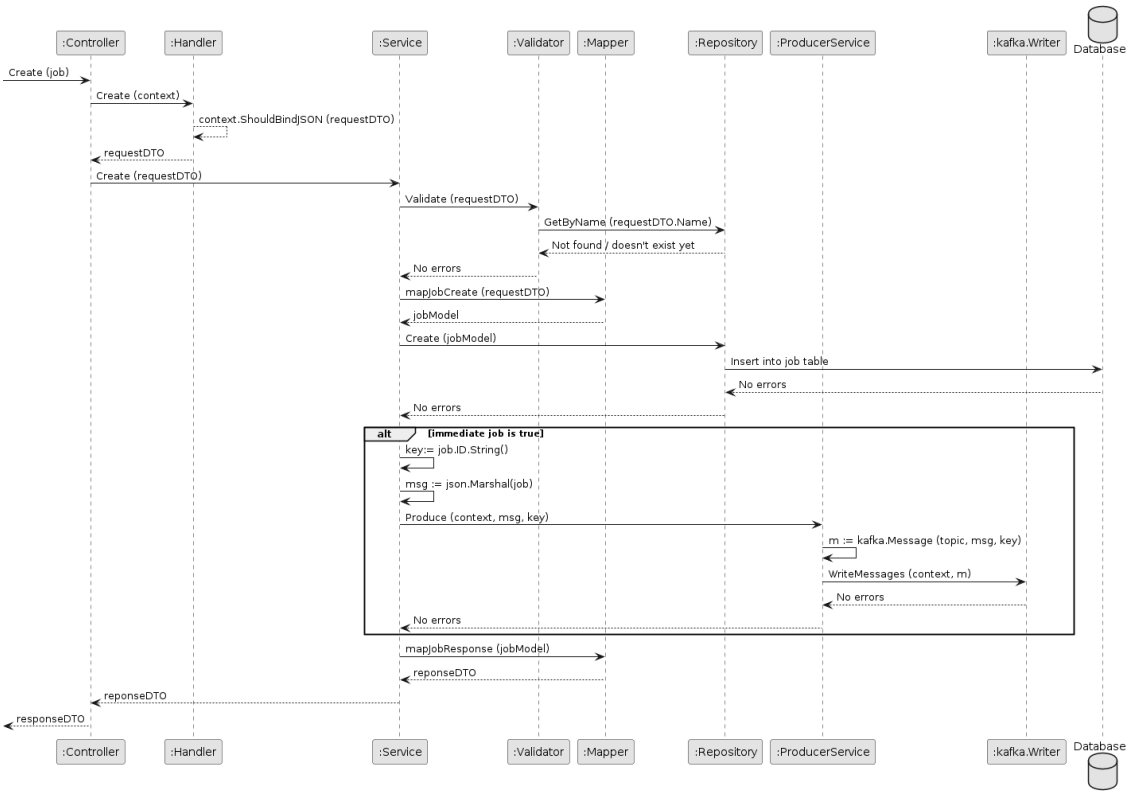

Figure 5.4: Create Job Sequence Diagram.

When a create job request arrives at Eye of Sauron, it is handled by the job's Controller that calls its Handler to bind the JSON payload into a request DTO. If succeeded, that DTO is sent to the Service where a validation method is called. This validation is nothing more that evaluating if a job with the same name already exists in the database or not and, if it is unique, then it can be created. After this step, the DTO is mapped to a model object and the Service calls the Repository method to persist that job in the database. One of the job's attributes is if it is immediate or not. At this point, if the job is immediate, then it's encoded and written in a Kafka topic called "immediate_jobs". To finalize, the created job is mapped into a response DTO and it is sent in the response body. If everything was done with success, the response status code is 201 Created.

Update Job

The endpoint to update a job in Eye of Sauron uses the HTTP method **PUT**, and the route **/jobs/:id**, where :id should be replaced by the job's ID of type Universal Unique Identifier (UUID).

The process to update a job is similar to the create one, but there are some minor differences. One of them is that in the Handler, although the bind of the JSON request payload is done, a bind from the Uniform Resource Identifier (URI) to a struct is also made, since this is the way to pick the job ID from the parameters. Then, in the Service, the ID is used to get the job from the database and obtain the old stored data. After this step, the same validation on the job's name is done, the only new step is exclusion of the name if the job ID is the same. At this point the DTO is mapped to a model that will be used to update the entry in the database by calling the Repository from the Service. If the job is immediate, it is sent

into Kafka's "immediate_jobs" topic and, in the end, the updated model is mapped to a response DTO which will be sent in the response with the status code 200 OK.

The next image, Figure 5.5, allows to see this process visually in the format of a sequence diagram.

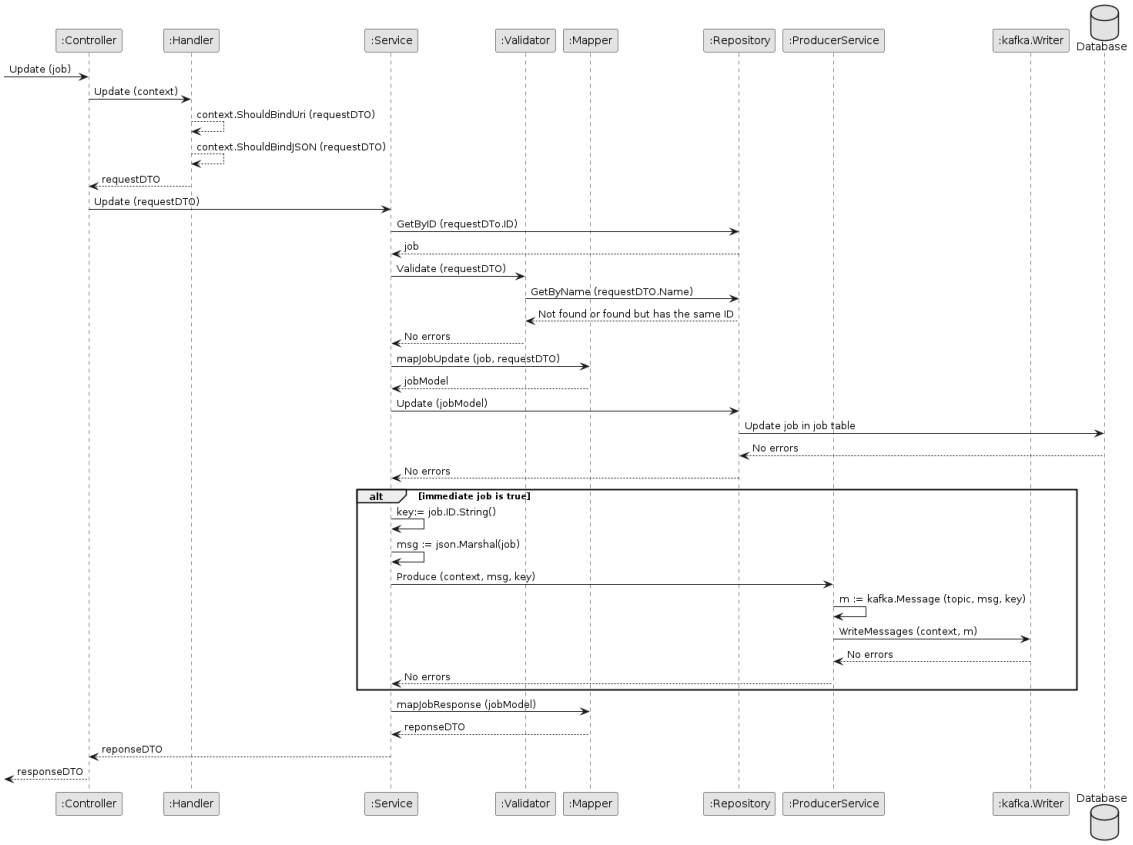

Figure 5.5: Update Job Sequence Diagram.

Delete Job

In Eye of Sauron's REST API, to delete a job the only thing necessary is to do a **DELETE** HTTP request to **/jobs/:id**, where :id is the ID of the job to be deleted from the system. During this process, the ID is binded to a request DTO struct which is sent to the Service. Here, the Service calls a Repository method to get the job by this ID. If the job exists on the system, the Service calls another Repository method but this time to delete that job from the database. If the job doesn't exist, a 404 Not Found Resource error is returned, otherwise, if succeeded, 204 No Content, and that means that the job was deleted with success.

Figure 5.6 illustrates all the steps in a sequence diagram.

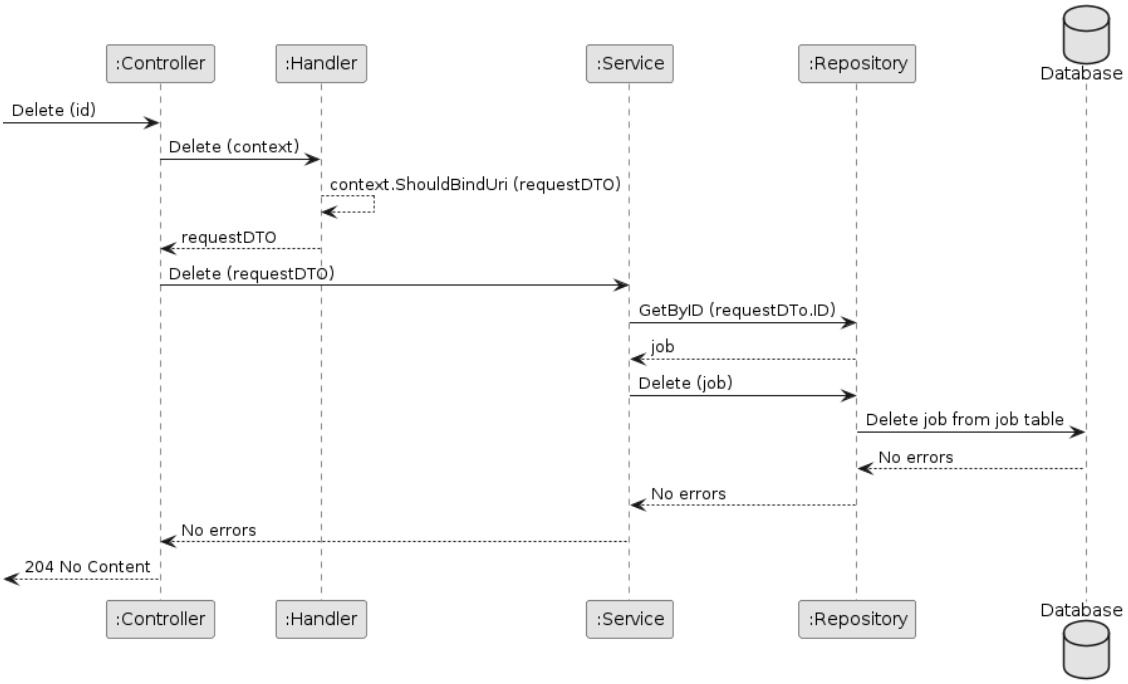

Figure 5.6: Delete Job Sequence Diagram.

Get Job

The process behind this endpoint is very similar to delete, but has less steps. The HTTP method used id **GET** to the route **/jobs/:id**, where ID of the job to obtain.

Figure 5.7 visually represents this process using a sequence diagram.

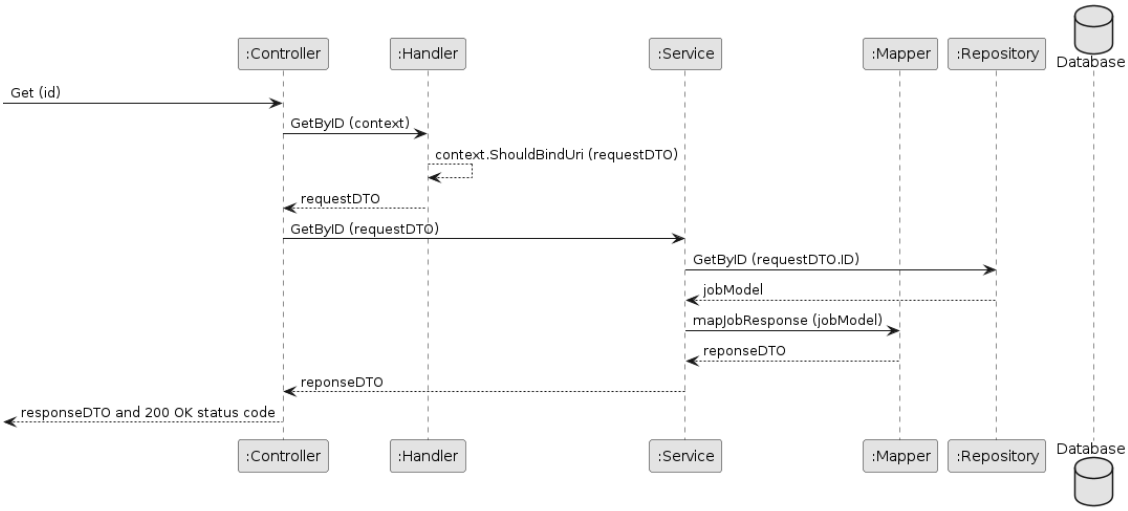

Figure 5.7: Get Job by ID Sequence Diagram.

List Jobs

This is the endpoint to get a list of jobs that can be filtered, sorted, and paginated. The HTTP method to use is **GET** to the route **/jobs**. It is possible to use query parameters when doing the request to the API, for instance:

/jobs?sort_by=nameorder=ascpage=1per_page=10immediate=false

This route returns a paginated list with a maximum amount of ten delayed jobs (because immediate is false), starting in the first page, and that is ordered by ascending name.

The Figure 5.8 shows the internal process to obtain a list of jobs from Eye of Sauron.

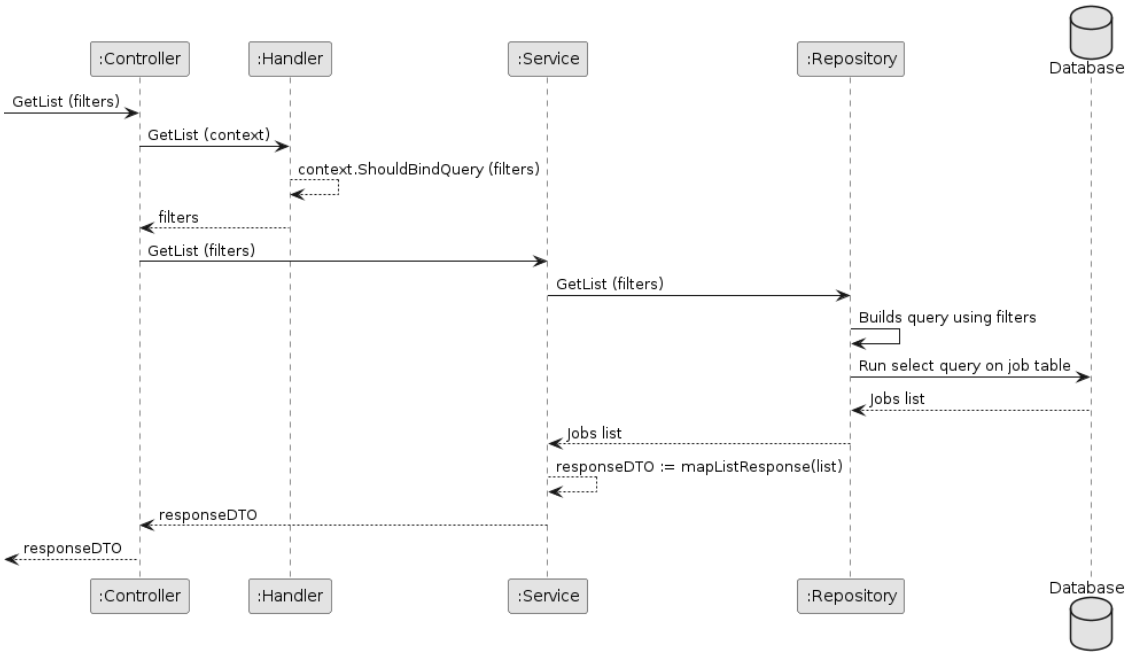

Figure 5.8: List Jobs Sequence Diagram.

Get Job Statistics

/jobs/:id/stats is the route to use in order to obtain statistics from a job with a certain ID when doing a **GET** HTTP request in Eye of Sauron's REST API.

In Figure 5.9, it is illustrated how this process happens. It starts in the Controller by calling an Handler method to bind the URI to a resquest DTO struct. Then, the Controller sends this DTO to the Service that uses it to get the job ID and send it to a Repository method that queries the database and picks the necessary data. After this step, the Repository sends the data back to the Service that maps it to a response DTO. The response body includes the number of runs, number of successes, failures and retries, and when was the last execution datetime. The status code is 200 OK.

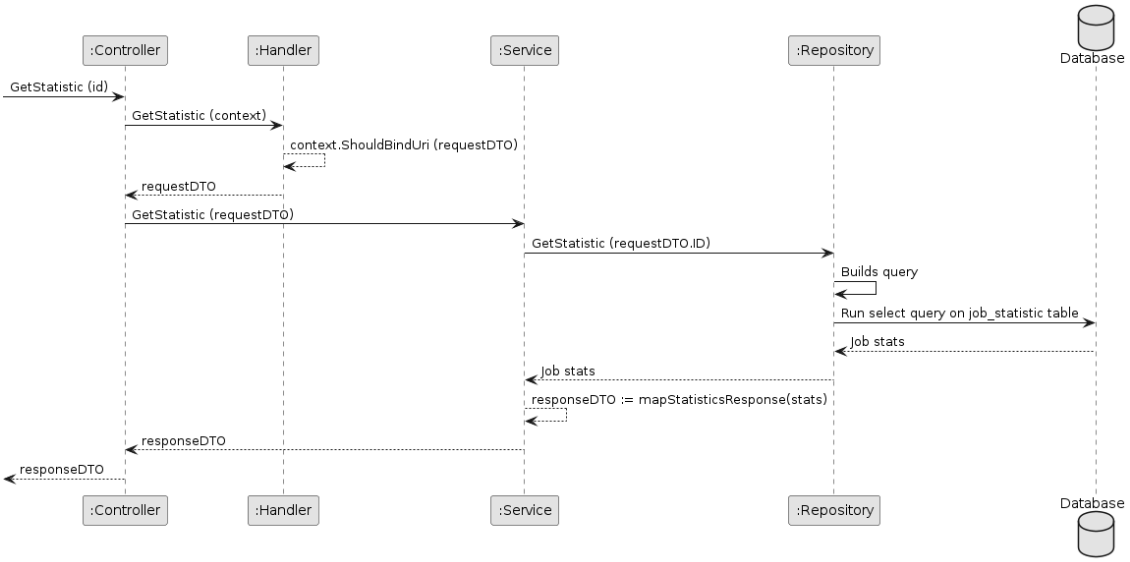

Figure 5.9: Get Job Statistics Sequence Diagram.

Get Job Logs

To get a list with logs about a job execution there is a route accessible doing a HTTP **GET** request to **/jobs/:id/logs**. Once again, :id is the ID of the job to obtain logs from. This endpoint supports a query parameter which is error=true or error=false, and filters the results according if they are related with error messages or not, respectively.

Figure 5.10, represents the sequence diagram that illustrates the process when calling this endpoint from Eye of Sauron's REST API. It starts by binding the parameter :id to a struct that is used to query the database later on. The query parameter error is optional, so it means that if not sent in the request, the API returns a list with all the logs independently if they are related with errors or not. After having the list of logs, the Service calls the mapper method to construct a response DTO that is sent in the response body. This response has as status code 200 OK.

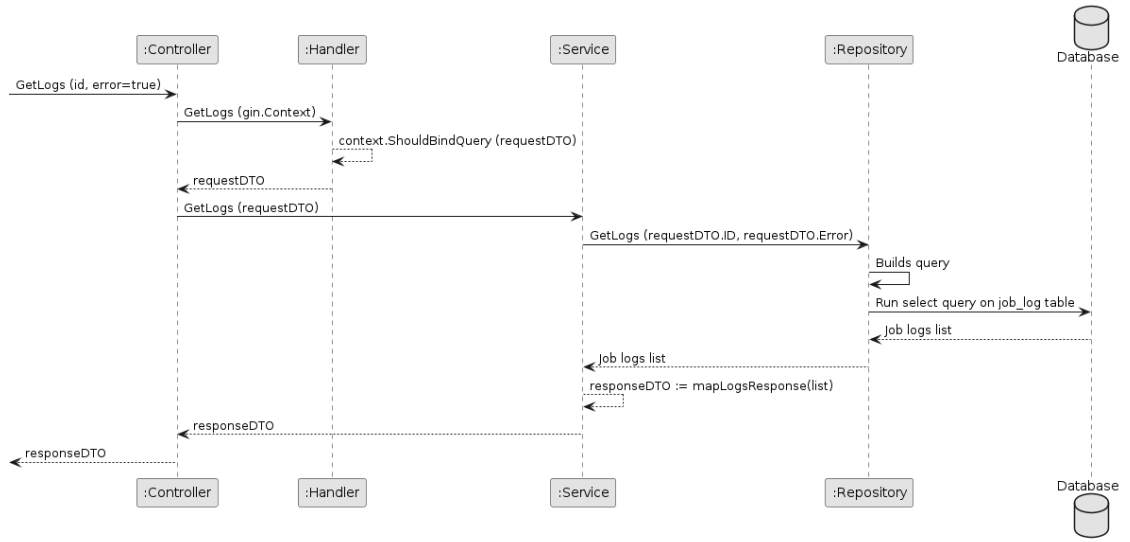

Figure 5.10: Get Job Error Logs Sequence Diagram.

5.3.3 Immediate Jobs

Immediate jobs are the ones with schedule for now. When a user creates or edits a job and selects this option, Eye of Sauron creates/updates the job on the database and then produces a message (Figures 5.4 and 5.5) into a Kafka topic named "immediate_jobs", as can be seen below in Figure 5.11.

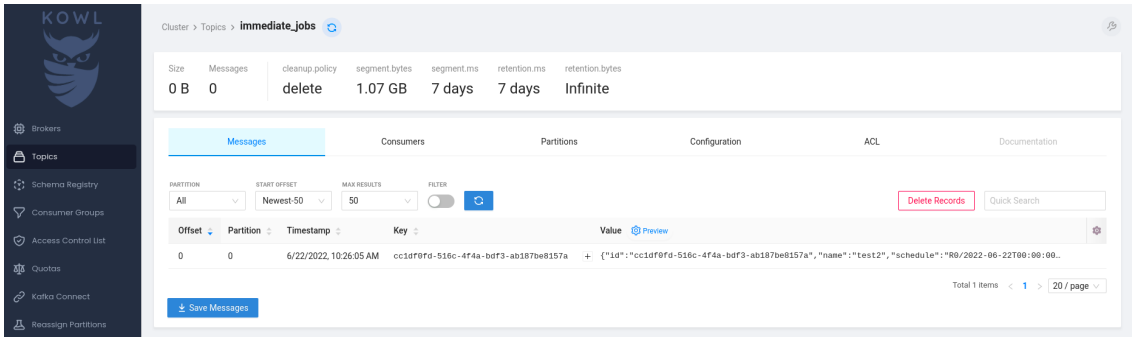

Figure 5.11: Message inside immediate jobs topic. Kowl is an open-source software that provides an user interface for Kafka.

After this, there is a consumer, the immediate jobs consumer, that is always listening for incoming messages in Kafka. Every time that a message arrives, it picks the job ID from the key in the Kafka message and calls the method Exec from the jobs executor service that executes the job immediately. Figure 5.12 presents a sequence diagram for this process.

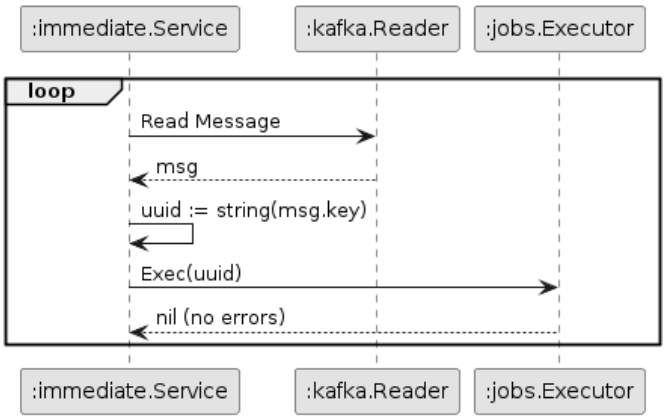

Figure 5.12: Immediate Jobs Consumer Sequence Diagram.

5.3.4 Delayed Jobs

Delayed jobs are the ones that have a schedule with a start date and time configured by the users. This is the go to option when they want to program a job to trigger a HTTP or Bash command for later in the same day or any other date and time in the future using UTC timezone. As for immediate jobs, this scheduling option also permits to select the amount of times to repeat the job with a configurable interval of time.

When creating or updating a delayed job, Eye of Sauron stores it in the database. In this case, to execute this kind of jobs, a component named "Watcher" queries the database every ten seconds (configurable value in Eye of Sauron) to obtain a list of jobs that are

scheduled to run in the last ten seconds. If that list contains one or more jobs, the watcher service emits a message into a Kafka topic called "delayed_jobs" for each one of them, so it acts as a producer. Figure 5.13 is a sequence diagram that represents this process.

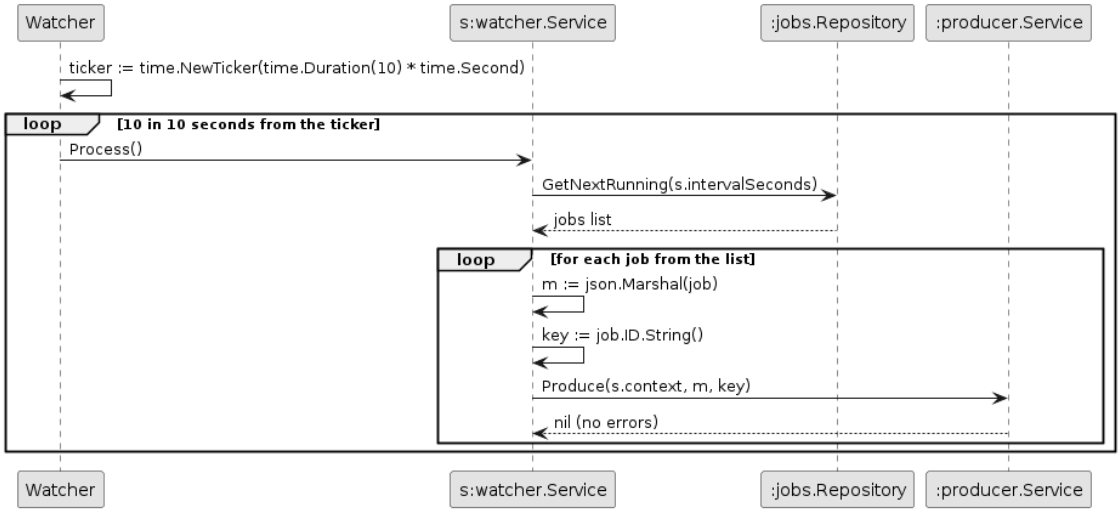

Figure 5.13: Delayed Jobs Producer Sequence Diagram.

The rest of the process is very similar to the immediate jobs. There is a consumer for delayed jobs that is checking the Kafka topic "delayed_jobs" and, if it has new messages, it reads them and calls the executor service to take care of execute them, as can be seen by Figure 5.14.

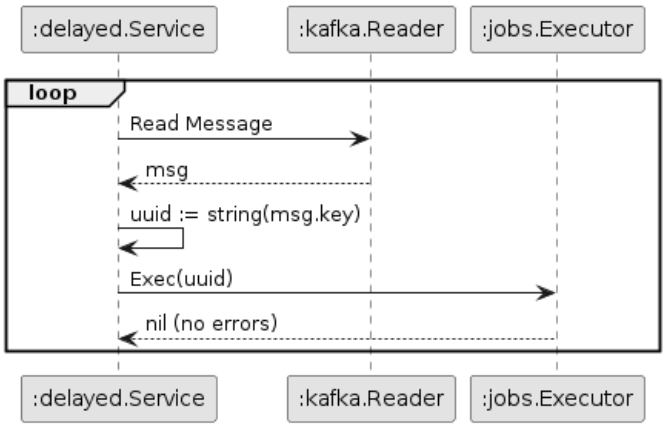

Figure 5.14: Delayed Jobs Consumer Sequence Diagram.

5.3.5 Job Executor

The job executor is a service that has the responsibility of doing an HTTP request if it's a job of type "http", and executing a command in a shell if it's a "bash" command.

The executor service has a bind method, named UseStrategy, that receives, as first parameter, a job type, and also an instance of an object that implements the jobs.Strategy interface, which contains the Exec method. This service is able to store multiple execution strategies, one for each job type.

This way, the executor only has to call the Exec method from a certain strategy for a job and it doesn't need to know the details of the implementation, that responsibility belongs to the execution strategy.

The code snippet below has the two interfaces used. The first one is related with the executor service, and shows its exported functions. The second one is the interface for the strategy pattern with the Exec method, that is called inside of the executor during the execution of a job.

```go
package jobs

import (
  "eyeofsauron/pkg/models"
)

type Executor interface {
  UseStrategy(jobType models.JobType, strategy Strategy) Executor
  Exec(id string) error
}

type Strategy interface {
  Exec(job models.Job) error
}
```

Listing 5.2: Executor and Strategy interfaces.

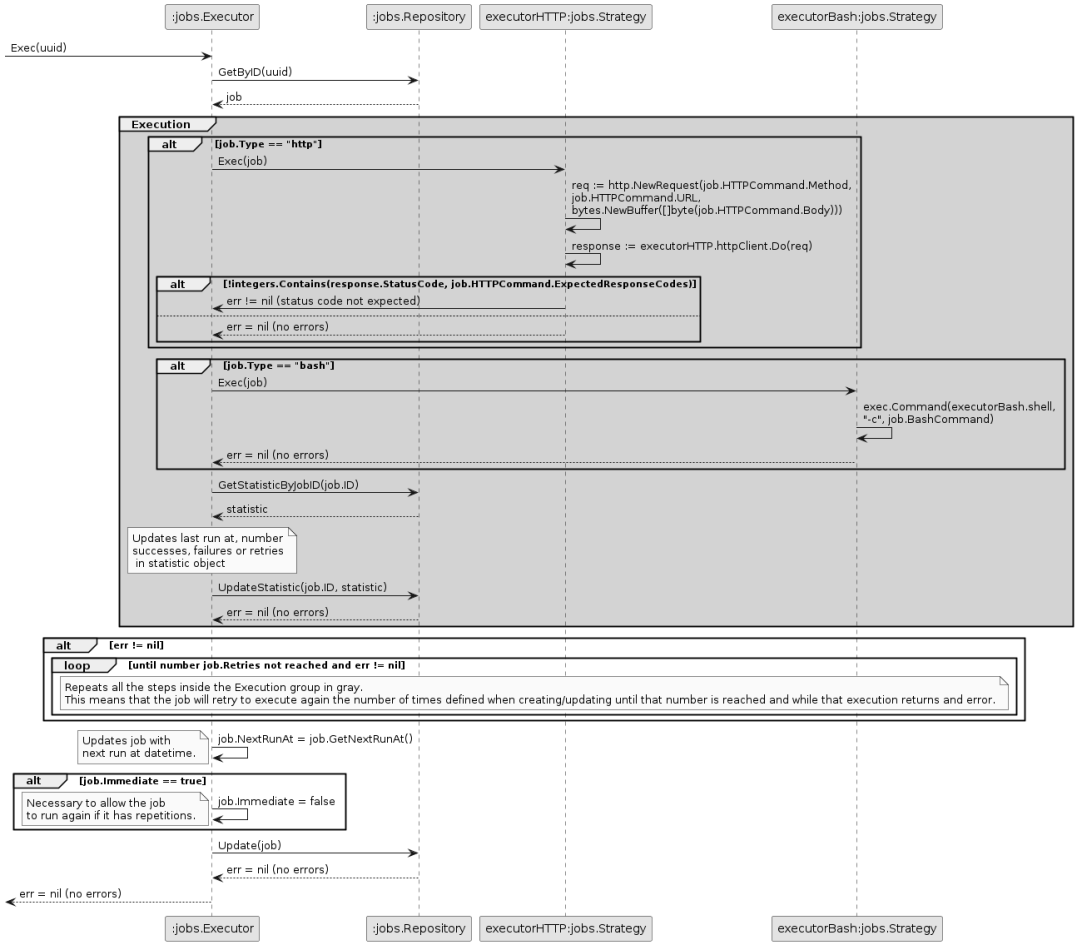

Figure 5.15: Job Executors Sequence Diagram.

Figure 5.15 is a sequence diagram that presents the steps to execute a HTTP or Bash job.

When the immediate or delayed jobs consumer sends a job ID to the executor service, as mentioned in 5.3.3 and 5.3.4, then this service queries the database, through the repository method GetByID, to obtain that job and its data.

After this first step, the job type is evaluated. Here the strategy pattern is used to execute the job according to its type: "http" or "bash". The executor service, which has the two strategies already configured, is able to pick the right one and call the Exec method, from that strategy, sending the job as parameter.

The code snippet bellow is related with the executor service and shows a partial implementation that makes use of the strategy pattern.

```
func (e executor) Exec(id string) error {
  job, err := e.repository.GetByID(id)
  if err != nil {
    log.Errorf("[Executor] Error getting job with ID %s, Err: %s", job.ID, err)
    return err
  }
  err = e.exec(job, false)
  // ...
}

func (e executor) exec(job models.Job, isRetry bool) error {
  strategy, err := e.getStrategy(job.Type)
  if err != nil {
    log.Errorf("[Executor] Strategy '%s' is not available", job.Type.String())
    return err
  }

  // ...

  execErr := strategy.Exec(job)

  // ...
}

func (e executor) getStrategy(jobType models.JobType) (jobs.Strategy, error) {
  strategy, exists := e.strategies[jobType]
  if !exists {
    return nil, errors.New("strategy not found")
  }
  return strategy, nil
}
```

Listing 5.3: Partial implementation of the strategy pattern.

When a call to the Exec method from a strategy returns an error, if the job has a number of retries greater than zero, the service will call the execution again until the method returns an error and the number of retries isn't reached. Considering the Figure 5.15, it's possible to say that the part that it's repeated is the one inside the gray box.

For each execution, the service also gets from the database the statistical data for that job, or creates it if doesn't exist yet. It updates the attributes last run at, number of successes, number of failures, and number of retries in the statistic object. After it, it calls the method UpdateStatistic from job's repository to store this data in the database.

In the end, the job object is also updated with the next execution datetime (next run at attribute), and its data is stored in the database. Finally, if everything went well, the executor service returns nil, meaning that this process was completed with success.

5.3.6 Database

Each microservice in Jumia Marketing and Digital Services has its own database so, to store the jobs, statistics, and execution logs, a PostgreSQL database was used for the Eye of Sauron microservice.

The next topics are related with the data model designed, how that model was created in the database, and what is the tool used to connect to the database from Eye of Sauron.

Data Model

A data model is a visual way of representing entities and an abstraction for the relation between these entities in a relational database. This kind of model allows to plan and design the database tables ahead of time in order to promote data consistency.

Figure 5.16 is a data model diagram that shows the entities Job, JobStatistic, and JobLog and the relations between them. It also presents two other tables, job audit and job statistic audit, which contain all the data related with the life-cycle of all records for the entities Job and JobStatistic.

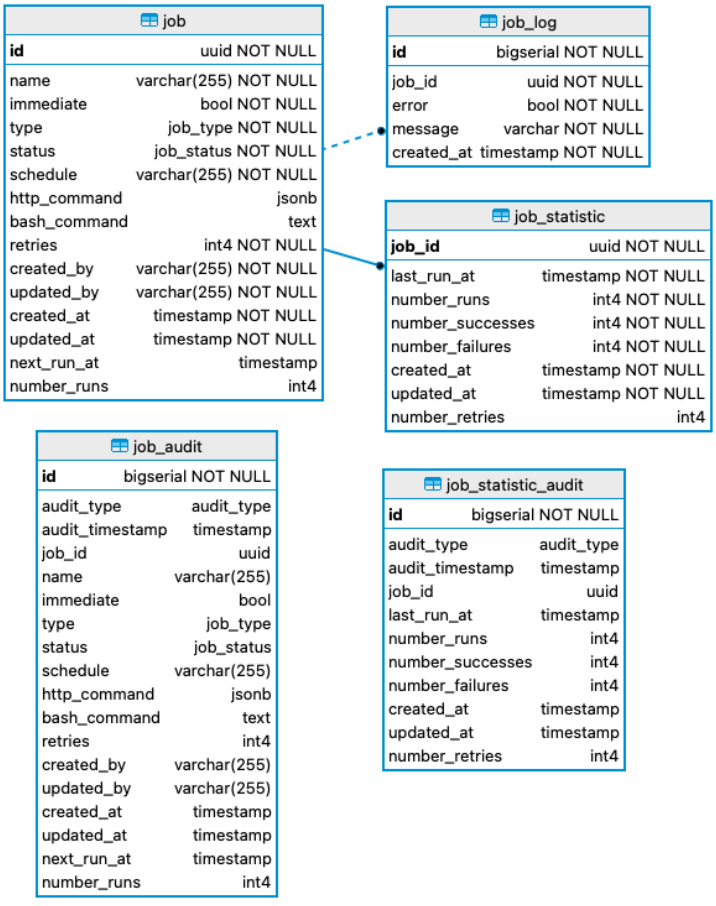

Figure 5.16: Data Model Diagram.

Migrations

A migration, in relational databases, is a Structured Query Language (SQL) script file, or collection of files, that allows to alter the database schema and, sometimes, to migrate existing data from one format or place into a new one.

In the context of this dissertation project, and taking into consideration the distributed job scheduler service Eye of Sauron v2, a migration is a collection of two SQL files where one of them has the suffix ".up" and the other one has the same name but the suffix ".down" and followed by the file extension ".sql", as can be seen by the Figure 5.17.

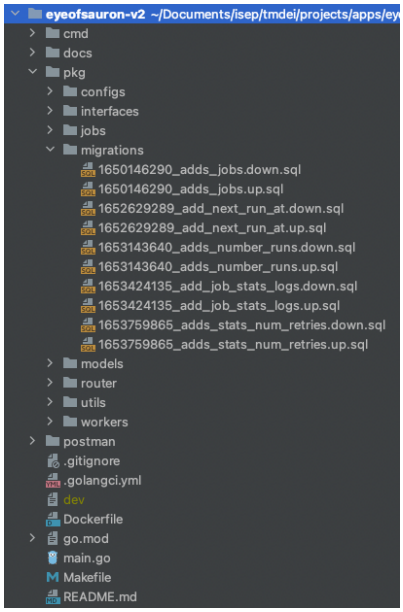

Figure 5.17: Project packages tree with migrations package and SQL scripts created.

In this case, the ".up" file has the new database schema and the ".down" SQL file has the old schema. This allows to migrate the database to a new state or to roll it back to the previous one, for example, by adding or removing data, adding or removing tables or columns.

As an example, the next code snippet has the contents of the file *1650146290_adds_jobs.up.sql* that contains the SQL necessary to create the Job entity, some custom types, and the audit table for the jobs. It also contains a function and a trigger that is executed every time that a record is inserted, updated or deleted in the job table, which adds a new row in the table job_audit keeping, this way, a history for each job.

```
1  BEGIN;
2
3  CREATE EXTENSION IF NOT EXISTS pgcrypto;
4  CREATE TYPE job_type AS ENUM ('bash', 'http');
5  CREATE TYPE job_status AS ENUM ('enabled', 'disabled', 'complete');
6
7  CREATE TABLE job
8  (
9      id              UUID DEFAULT gen_random_uuid() PRIMARY KEY,
10     name            VARCHAR(255) UNIQUE          NOT NULL,
11     immediate       BOOLEAN                      NOT NULL,
```

```
    type          job_type                    NOT NULL,
    status        job_status                  NOT NULL,
    schedule      VARCHAR(255)                NOT NULL,
    http_command JSONB,
    bash_command TEXT,
    retries       INTEGER                     NOT NULL,
    created_by    VARCHAR(255)                NOT NULL,
    updated_by    VARCHAR(255)                NOT NULL,
    created_at    TIMESTAMP WITHOUT TIME ZONE NOT NULL,
    updated_at    TIMESTAMP WITHOUT TIME ZONE NOT NULL,
    CONSTRAINT constraint_job_command CHECK (http_command IS NOT NULL OR
    bash_command IS NOT NULL)
);

CREATE TYPE audit_type AS ENUM ('INSERT', 'UPDATE', 'DELETE');

CREATE TABLE job_audit
(
    id               BIGSERIAL PRIMARY KEY,
    audit_type       audit_type,
    audit_timestamp TIMESTAMP WITHOUT TIME ZONE,
    job_id           UUID,
    name             VARCHAR(255),
    immediate        BOOLEAN,
    type             job_type,
    status           job_status,
    schedule         VARCHAR(255),
    http_command     JSONB,
    bash_command     TEXT,
    retries          INTEGER,
    created_by       VARCHAR(255),
    updated_by       VARCHAR(255),
    created_at       TIMESTAMP WITHOUT TIME ZONE,
    updated_at       TIMESTAMP WITHOUT TIME ZONE
);

CREATE OR REPLACE FUNCTION job_audit() RETURNS TRIGGER AS
$job_audit$
BEGIN
    IF (TG_OP IN ('INSERT', 'UPDATE')) THEN
        INSERT INTO job_audit (audit_type, audit_timestamp, job_id, "name",
    immediate, "type", status, schedule,
                               http_command, bash_command, retries, created_by,
     updated_by, created_at, updated_at)
        SELECT TG_OP::audit_type,
               CURRENT_TIMESTAMP,
               NEW.id,
               NEW.name,
               NEW.immediate,
               NEW.type,
               NEW.status,
               NEW.schedule,
               NEW.http_command,
               NEW.bash_command,
               NEW.retries,
               NEW.created_by,
               NEW.updated_by,
               NEW.created_at,
               NEW.updated_at;
        RETURN NEW;
    ELSE
        INSERT INTO job_audit (audit_type, audit_timestamp, job_id, "name",
    immediate, "type", status, schedule,
                               http_command, bash_command, retries, created_by,
     updated_by, created_at, updated_at)
        SELECT TG_OP::audit_type,
               CURRENT_TIMESTAMP,
```

```
                OLD.id,
                OLD.name,
                OLD.immediate,
                OLD.type,
                OLD.status,
                OLD.schedule,
                OLD.http_command,
                OLD.bash_command,
                OLD.retries,
                OLD.created_by,
                OLD.updated_by,
                OLD.created_at,
                OLD.updated_at;
        RETURN OLD;
    END IF;
END;
$job_audit$ LANGUAGE plpgsql;

CREATE TRIGGER job_audit
    AFTER INSERT OR UPDATE OR DELETE
    ON job
    FOR EACH ROW
EXECUTE PROCEDURE job_audit();

COMMIT;
```

Listing 5.4: Add jobs up migration.

GORM

GORM is an Object–Relational Mapping (ORM) library for Go. It has multiple features such as being able to create, update, delete records, and query database tables. It also has other functions, like the ability to do associations between tables, eager loading, and many others (Jinzhu 2022). GORM is used in Eye of Sauron v2 since is simple to use and has everything needed built-in.

To use GORM in a Go project, it is necessary to define the structs that represent the model objects. The code snippet bellow presents a partial implementation for the Job model and some of its related types with GORM tags.

```
package models

import (
  "database/sql/driver"
  "encoding/json"
  "errors"
  "eyeofsauron/pkg/utils/iso8601"
  "github.com/google/uuid"
  "regexp"
  "strconv"
  "time"
)

type Job struct {
  ID           uuid.UUID       `gorm:"type:uuid;default:gen_random_uuid();primaryKey" json:"id"`
  Name         string          `gorm:"type:varchar(255); column:name" json:"name"`
  Schedule     string          `gorm:"type:varchar(255); column:schedule" json:"schedule"`
  Retries      int             `gorm:"type:integer; column:retries" json:"retries"`
  NumberRuns   int             `gorm:"type:integer; column:number_runs" json:"number_runs"`
  Status       JobStatus       `gorm:"type:job_status; column:status" json:"status"`
  Type         JobType         `gorm:"type:job_type; column:type" json:"type"`
  BashCommand  *string         `gorm:"type:text; column:bash_command" json:"bash_command"`
  HTTPCommand *HTTPProperties `gorm:"type:jsonb; column:http_command" json:"http_command"`
  Immediate    bool            `gorm:"type:boolean; column:immediate" json:"immediate"`
  NextRunAt    time.Time       `gorm:"type:timestamp; column:next_run_at" json:"next_run_at"`
  CreatedAt    time.Time       `gorm:"type:timestamp; column:created_at" json:"created_at"`
  UpdatedAt    time.Time       `gorm:"type:timestamp; column:updated_at" json:"updated_at"`
  CreatedBy    string          `gorm:"type:varchar(255); column:created_by" json:"created_by"`
  UpdatedBy    string          `gorm:"type:varchar(255); column:updated_by" json:"updated_by"`
  Statistic    JobStatistic    `gorm:"association_autoupdate:false;association_autocreate:false" json:"
     statistic"`
```

```
}

type JobStatus string

const (
  StatusEnabled  JobStatus = "enabled"
  StatusDisabled JobStatus = "disabled"
  StatusComplete JobStatus = "complete"
)

type JobType string

const (
  TypeBash JobType = "bash"
  TypeHTTP JobType = "http"
)

type HTTPProperties struct {
  URL                   string              `json:"url"`
  Method                string              `json:"method"`
  Body                  string              `json:"body"`
  Headers               map[string][]string `json:"headers"`
  Timeout               int                 `json:"timeout"`
  ExpectedResponseCodes []int               `json:"expected_response_codes"`
}
```

Listing 5.5: Job model and related types.

Using GORM in a Repository method is really easy and fast, the only thing needed is to call its methods that support chaining. In the end, the query is auto generated as it's done by other ORMs available. Below is possible to find a code snippet with a partial implementation of the jobs Repository which makes use of GORM´s Where, Preload, Take, Find, Create, Delete, and many other available methods.

```
package jobs

import (
  "errors"
  "eyeofsauron/pkg/interfaces/jobs"
  "eyeofsauron/pkg/interfaces/utils/db"
  "eyeofsauron/pkg/models"
  errs "eyeofsauron/pkg/utils/errors"
  "eyeofsauron/pkg/utils/pagination"
  "fmt"
  "gorm.io/gorm"
)

type repository struct {
  db         db.Database
  pagination pagination.Service
}

func NewRepository(dbCli db.Database, pag pagination.Service) jobs.Repository {
  return repository{
    dbCli,
    pag,
  }
}

func (r repository) GetByID(id string) (models.Job, error) {
  var job models.Job
  if err := r.db.Where("id = ?", id).Preload("Statistic").Take(&job).Error();
    err != nil {
    if errors.Is(err, gorm.ErrRecordNotFound) {
      return models.Job{}, errs.NewNotFoundResourceError(errs.NotFound, "ID", "
    Job not found", nil)
    }
    return models.Job{}, errs.NewInternalError(errs.InternalErr, "ID", "
    Unexpected error getting job", err)
  }
  return job, nil
```

```
35 }
36
37 func (r repository) Create(job models.Job) (models.Job, error) {
38   err := r.db.Create(&job).Error()
39   return job, err
40 }
41
42 func (r repository) Update(job models.Job) (models.Job, error) {
43   err := r.db.Omit("Statistic").Save(&job).Error()
44   return job, err
45 }
46
47 func (r repository) Delete(job models.Job) error {
48   return r.db.Delete(&job).Error()
49 }
50
51 func (r repository) GetNextRunning(intervalSecs int) ([]models.Job, error) {
52   var list []models.Job
53   err := r.db.
54     Where("immediate = false").
55     Where("status = 'enabled'").
56     Where(fmt.Sprintf("next_run_at >= (NOW() - interval '%d seconds') AT TIME
       ZONE 'UTC' AND next_run_at <= NOW() AT TIME ZONE 'UTC'", intervalSecs)).
57     Find(&list).
58     Error()
59   return list, err
60 }
```

Listing 5.6: Partial implementation of the jobs repository.

5.3.7 Reverse Proxy

A reverse proxy is a server that sits in front of one or more web servers, intercepting requests from clients. When a request is received from the clients, this server sends those requests to the backend services. It also receives the responses from the backend servers which are sent back to the clients (NGINX 2022).

Figure 5.18 represents how a reverse proxy works, and shows the flow of a request that is sent to the address *example.com* from the user's devices to the internet, which is then received by the reverse proxy server that sends those requests to backend (origin) servers. This avoids direct communication between the client and the servers and offers some benefits, such as load balancing, Secure Sockets Layer (SSL) encryption, and protection from attacks (Cloudflare 2022).

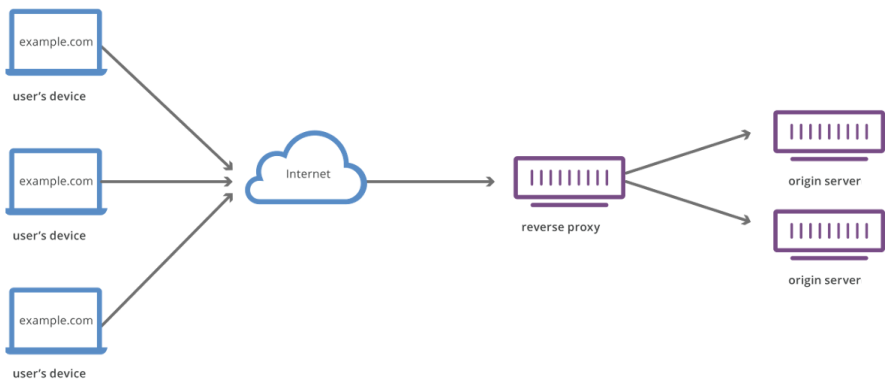

Figure 5.18: Reverse Proxy Flow. Adapted from Cloudflare (2022).

Nginx was used for local development to add SSL encryption to the requests coming from the HTTP clients, such as the frontend service Andromeda, or any other client as Postman. This allows, in the case of Andromeda, to avoid Cross-Origin Resource Sharing (CORS) errors.

Bellow is a code snippet with a partial configuration used to setup Nginx to intercept requests from HTTP clients to the host address *https://eostmdei.mds.dev*, and to send them to the new distributed job scheduler Eye of Sauron v2 running in a Docker container in a local environment.

```
server {
    listen 80;
    listen 443 ssl;
    server_name eostmdei.mds.dev;

    resolver 127.0.0.11;
    set $upstreamweb http://eostmdei:8080;

    ssl_certificate      /etc/nginx/ssl/server.pem;
    ssl_certificate_key  /etc/nginx/ssl/server.key;

    error_log /dev/stdout;
    access_log /dev/stdout;

    location / {
        proxy_pass $upstreamweb;
        proxy_pass_request_headers on;
    }
}
```

Listing 5.7: Partial configuration for Nginx.

To the requests to *https://eostmdei.mds.dev* be sent to Nginx, running in a Docker container with the IP address 127.0.0.11, the following line had to be included in the file /etc/hosts.

```
127.0.0.11 eostmdei.mds.dev
```

5.3.8 Virtualization

Docker was used to run locally all the microservices, and also the PostgreSQL database, Nginx, and Apache Kafka. An existent configuration of a docker-compose file was adapted to add the new microservice Eye of Sauron v2, with the name *eostmdei* as can be saw in the next code snippet.

```
eostmdei:
build:
  context: ../../apps/eyeofsauron-v2
  dockerfile: $PWD/eostmdei/Dockerfile
  args:
    GOPATH: "/go"
    GOROOT: "/usr/local/go"
restart: always
labels:
  service: "eostmdei"
```

```
entrypoint: /bin/bash /opt/entrypoint.sh
working_dir: /go/src/eostmdei
container_name: eostmdei
environment:
  APP_ENV: "staging"
  DB_HOST: "postgres"
  DB_PORT: "5432"
  DB_USER: "mds"
  DB_PASSWORD: "mds"
  DB_NAME: "eostmdei"
  DB_DEBUG: "true"
  DB_MAX_OPEN_CONNECTIONS: 100
  DB_MAX_IDLE_CONNECTIONS: 5
  DB_CONNECTION_MAX_LIFETIME: 300
  KAFKA_HOST: "kafka"
  KAFKA_PORT: "29092"
  KAFKA_IMMEDIATE_JOBS_TOPIC: "immediate_jobs"
  KAFKA_GROUP_ID_IMMEDIATE_JOBS: "immediate-jobs-group"
  KAFKA_DELAYED_JOBS_TOPIC: "delayed_jobs"
  KAFKA_GROUP_ID_DELAYED_JOBS: "delayed-jobs-group"
deploy:
  resources:
    limits:
      memory: 500M
ports:
  - "2380:8080"
volumes:
  - ../../apps/eyeofsauron-v2:/go/src/eostmdei
  - $PWD/eostmdei:/opt
networks:
  localnet:
    ipv4_address: 172.21.0.49
depends_on:
  - postgres
```

Listing 5.8: Partial docker-compose file with the configuration for Eye of Sauron v2's container.

5.3.9 Code Coverage

Unit tests were written to cover the microservice functionalities. The code snippet bellow is an example of a test for the Create method of the jobs Service, and represents an example of a success scenario for creating a delayed job.

```
package jobs

import (
  "context"
  "eyeofsauron/pkg/interfaces/jobs"
  jobMocks "eyeofsauron/pkg/mocks/interfaces/jobs"
  kafkaMocks "eyeofsauron/pkg/mocks/interfaces/utils/kafka/producer"
  "eyeofsauron/pkg/models"
  "github.com/google/uuid"
  "github.com/stretchr/testify/assert"
  "testing"
  "time"
)

type srvMocks struct {
  repo              *jobMocks.Repository
  validator         *jobMocks.Validator
  immediateProducer *kafkaMocks.Service
}
```

```
20
21 func setupService() (jobs.Service, srvMocks) {
22   repo := new(jobMocks.Repository)
23   mapper := NewMapper()
24   validator := new(jobMocks.Validator)
25   immediateProducer := new(kafkaMocks.Service)
26   s := NewService(repo, mapper, validator, immediateProducer)
27   m := srvMocks{repo, validator, immediateProducer}
28   return s, m
29 }
30
31 func TestService_Create(t *testing.T) {
32   t.Run("success - delayed job", func(t *testing.T) {
33     s, m := setupService()
34
35     ctx := context.Background()
36     command := "ls -la"
37     request := jobs.Create{
38       Name:        "test_job",
39       Schedule:    "R0/2022-08-15T18:30:00/P0Y0M0DT0H0M0S",
40       Retries:     0,
41       Status:      "enabled",
42       Type:        "bash",
43       BashCommand: &command,
44       Immediate:   false,
45       CreatedBy:   "john.doe@email.com",
46     }
47     job := models.Job{
48       Name:        "test_job",
49       Schedule:    "R0/2022-08-15T18:30:00/P0Y0M0DT0H0M0S",
50       Retries:     0,
51       Status:      "enabled",
52       Type:        "bash",
53       BashCommand: &command,
54       Immediate:   false,
55       NextRunAt:   time.Date(2022, 8, 15, 18, 30, 0, 0, time.UTC),
56       CreatedBy:   "john.doe@email.com",
57       UpdatedBy:   "john.doe@email.com",
58     }
59     createdJob := job
60     createdJob.ID = uuid.New()
61     createdJob.CreatedAt = time.Date(2022, 5, 12, 18, 27, 10, 0, time.UTC)
62     createdJob.UpdatedAt = time.Date(2022, 5, 12, 18, 27, 10, 0, time.UTC)
63
64     m.validator.On("Validate", request).Return(nil)
65     m.repo.On("Create", job).Return(createdJob, nil)
66
67     result, err := s.Create(ctx, request)
68
69     expected := jobs.Job{
70       ID:          createdJob.ID,
71       Name:        "test_job",
72       Schedule:    "R0/2022-08-15T18:30:00/P0Y0M0DT0H0M0S",
73       Retries:     0,
74       Status:      "enabled",
75       Type:        "bash",
76       BashCommand: &command,
77       Immediate:   false,
78       NextRunAt:   time.Date(2022, 8, 15, 18, 30, 0, 0, time.UTC),
79       CreatedBy:   "john.doe@email.com",
80       UpdatedBy:   "john.doe@email.com",
81       CreatedAt:   time.Date(2022, 5, 12, 18, 27, 10, 0, time.UTC),
82       UpdatedAt:   time.Date(2022, 5, 12, 18, 27, 10, 0, time.UTC),
83     }
84
85     assert.Nil(t, err)
86     assert.Equal(t, expected, result)
```

```
87      m.repo.AssertExpectations(t)
88      m.validator.AssertExpectations(t)
89    })
90
91    // ...
92  }
```

Listing 5.9: Unit test for the success scenario when creating a delayed job.

5.4 Frontend Implementation

To add the new features in a frontend for the users be able to create and manage jobs, the existing Andromeda microservice was used.

A jobs model already existed, but only supported the listing of jobs. The creation of a job was always made by backend services, either by calling directly the original Eye of Sauron API or by the campaigns service calling Eye of Sauron.

Sometimes there is the need of setting up a cron job for recurring system tasks, and that had to be made in a HTTP client by the development team. Having those features in the frontend allows other persons to do that instead, and also offers other possibilities like the capability of checking statistic and execution error logs more easily.

The next figures show the user interface developed with the features that are now available such as: list (with or without filters), create, edit, clone, delete, consult statistics and error logs.

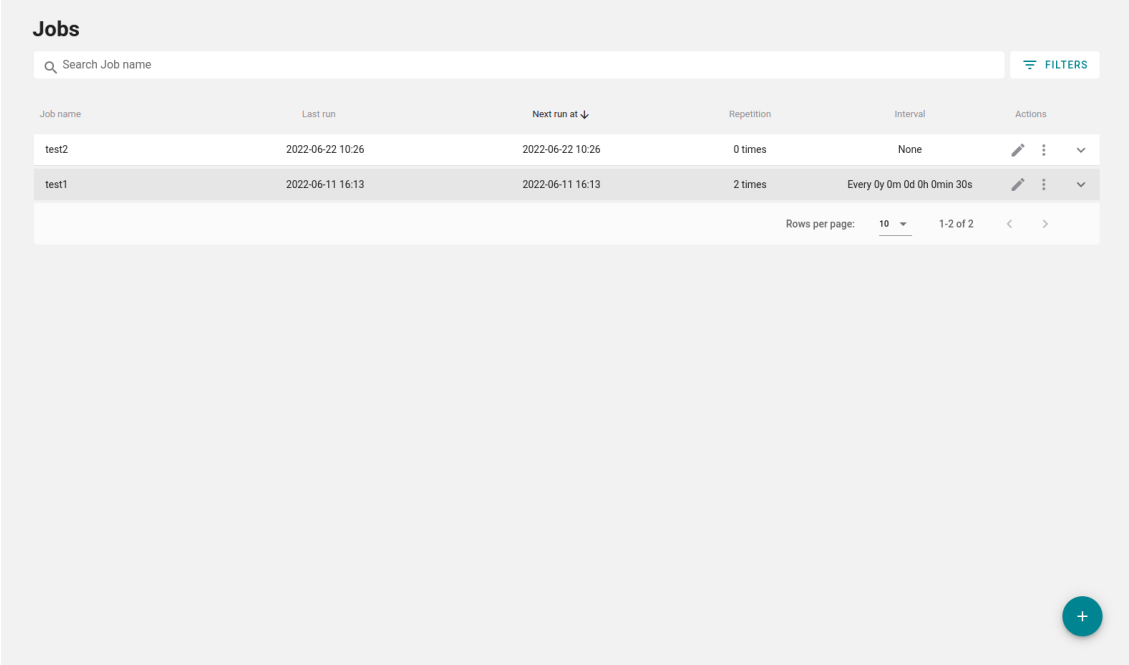

Figure 5.19: Jobs List.

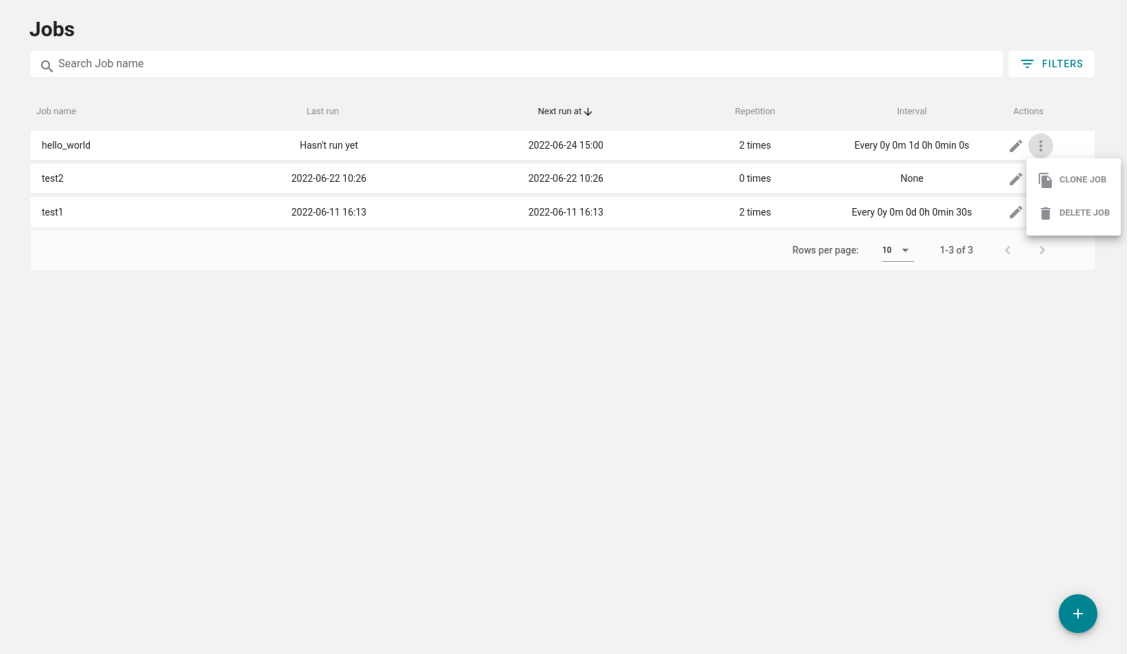

Figure 5.20: Clone and Delete Job.

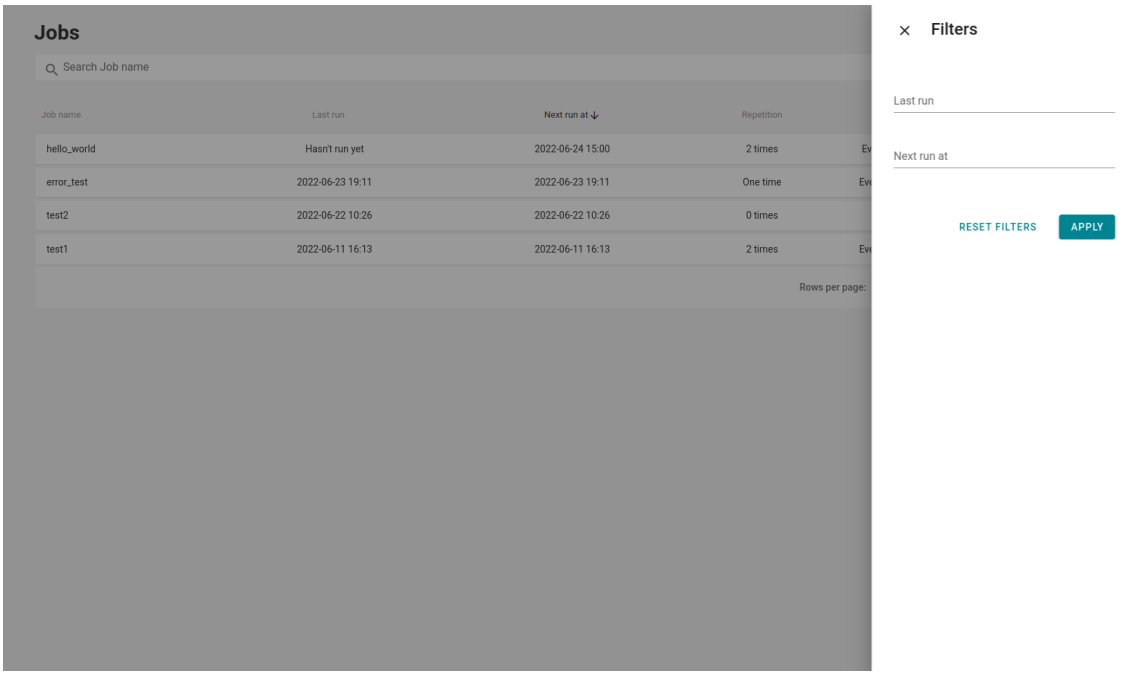

Figure 5.21: Filter Jobs.

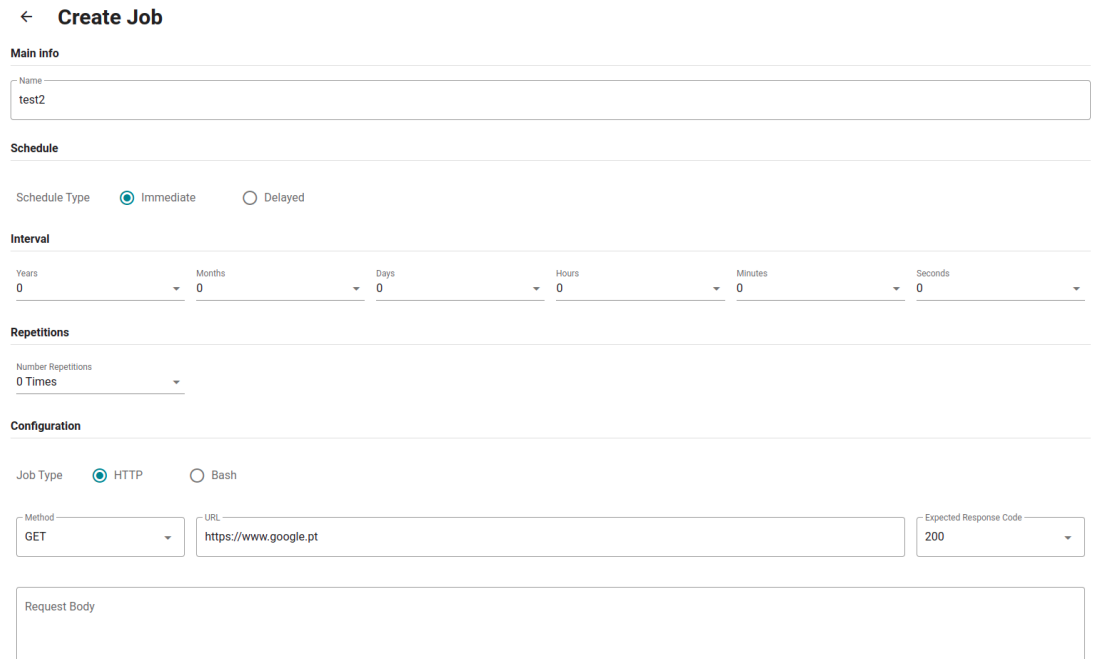

Figure 5.22: Create Immediate Job Form.

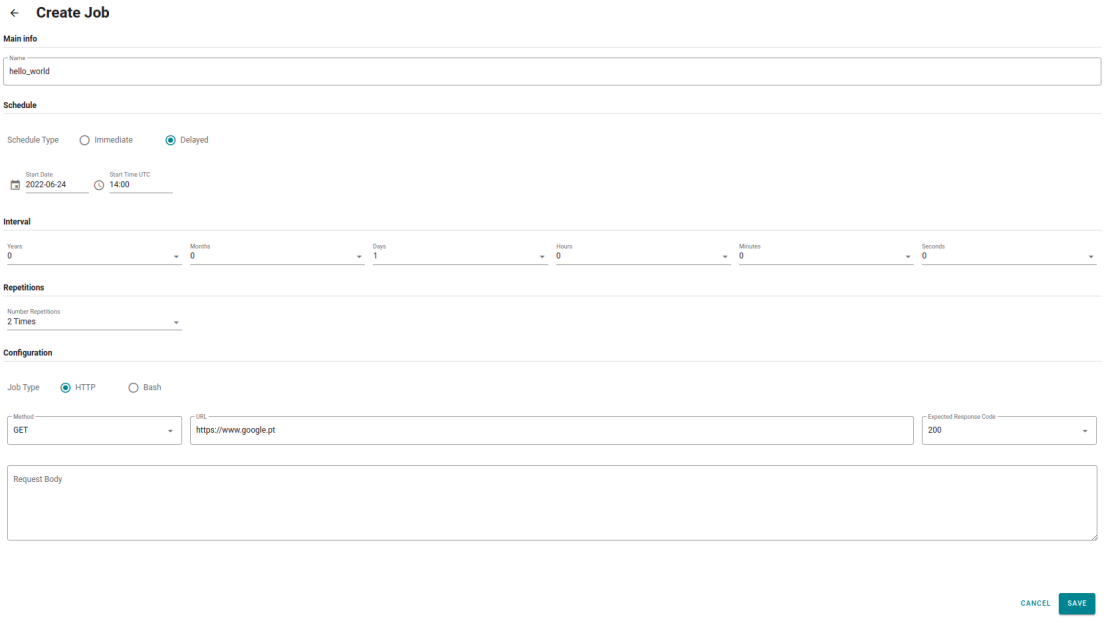

Figure 5.23: Create Delayed Job Form.

Figure 5.24: Bash Job Form.

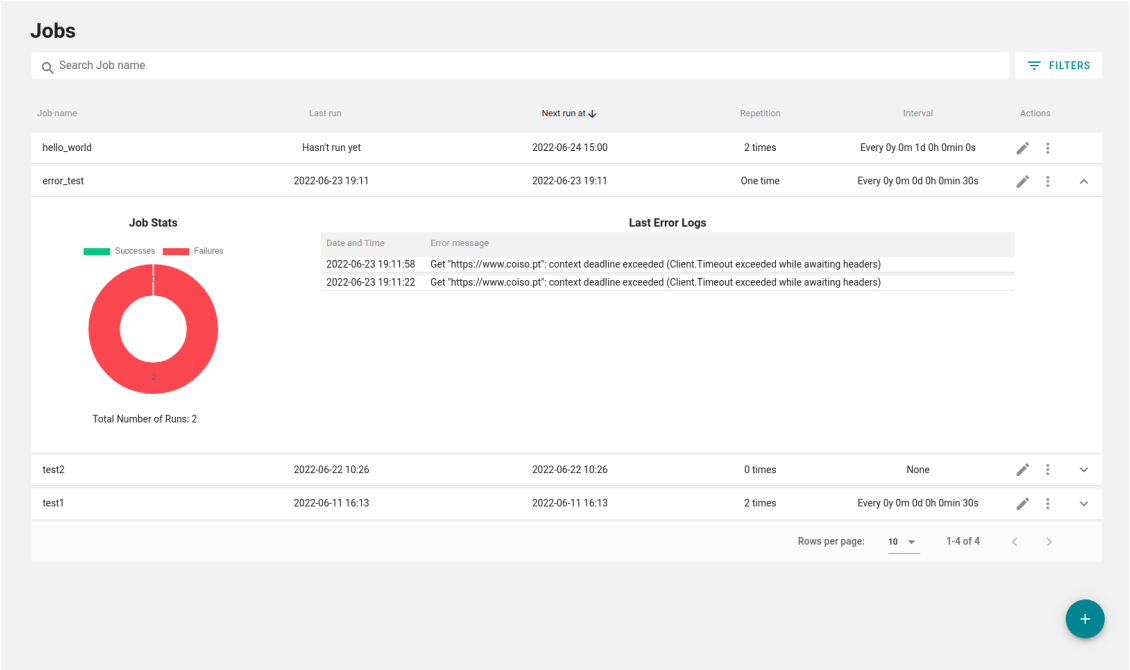

Figure 5.25: Job Statistics and Error Logs.

Chapter 6

Evaluation

In the software engineering industry quality is an important aspect that should be considered when developing any kind of project. A good system is considered the one that not only fulfills the customer's needs but also does it effectively.

In this industry, saying that a system has a good quality means that it must reunite a set of predefined attributes and fulfill them with a percentage above a certain threshold.

The purpose of this chapter is to describe the hypothesis, indicators, information sources, and methodology to evaluate the quality of the solution proposed with this project in relation to the established objectives and requirements. In the end the final quality of the project is presented and what is the tool used to obtain it.

6.1 Hypothesis

This section presents the hypothesis that guide the evaluation to this project's system and allow to tell if it has quality or not.

According with the objectives and requirements (functional and non-functional) described in previous chapters, this project is considered viable if fulfills some quality dimensions that need to be measured.

The dimensions that will be used to evaluate the system's quality are:

- Functionality - evaluation of use cases, user interaction, interface's navigation and content quality;
- Adaptability, Supportability, and Reliability - assessment of the system's maintainability and scalability capabilities, and execution errors.

Considering a predefined percentage of 60% for the end result of the system's quality assessment, a null hypothesis (H_0) and an alternative (H_1) can be defined:

- H_0 - The system isn't viable since it is evaluated below the predefined percentage;
- H_1 - The system is viable since it is evaluated with the same percentage or above the predefined value.

The quality of the solution proposed with this system is then assessed, and a final conclusion about if it's viable or not is made in section 6.4.

6.2 Indicators and Sources of Information

This section presents the indicators and sources of information that need to be collected for later evaluation of the solution's quality and validity.

The dimensions presented in the previous section can be divided into two groups, each one with different possible sources of information:

- Functionality - User Interface/User Experience (UI/UX) validation by the developer and the Quality Assurance (QA) of the MDS team;
- Reliability, Adaptability and Supportability - General tests to the system by the developer and peer review, and others such as unit tests.

6.3 Evaluation Methodology

This section describes the methodology that was used to assess the quality of the project's solution according with the dimensions and sources of information mentioned in the previous section.

This project had two major parts - the implementation of a new microservice, which is the distributed job scheduler Eye of Sauron v2, and the a frontend part that involved the improvement and addition of new features to the user interface.

Regarding the job scheduler microservice, its evaluation was done using specific aspects. One on them is if the system contemplated all the functional and non-functional requirements from subsections 4.1.1 and 4.1.2 that were collected from the project's stakeholders during the analysis phase. This was evaluated by the developer by testing the REST API of the service and checking if all functionalities were present and if they behave as expected.

Other aspects related with the scalability and maintainability of the software solution were also taken into consideration, such as if the microservice components are capable of being scaled horizontally and/or vertically, if the source code has a good code coverage, and if it follows good industry practices and design patterns. These were also checked by the developer and other MDS' developers during code reviews.

The frontend evaluation was done by the developer and MDS' QA that, for each functional requirement checked if it was implemented as expected by its context and final purpose descriptions. Other characteristics were also assessed like the capability of the system to handle errors and present useful messages to the users, and if the forms are correctly structured.

6.4 Quality Measurement

This section aims to get this project's final quality percentage using a tool named Quantitative Evaluation Framework (QEF), that is described in the first subsection. In the end, the application of QEF to this project is made and the quality measure is obtained.

6.4.1 Quantitative Evaluation Framework

QEF is a model developed by Escudeiro and Bidarra (2008) that evaluates a real solution (system implemented) compared with an ideal solution (system that achieves all evaluation parameters considered).

It can be applied to any kind of software solution to evaluate and validate the system's implementation in any phase of its development life cycle, which allows predicting deviations from the initial specifications and requirements.

This model evaluates a system in a three-dimensional quality space where each dimension aggregates a set of factors. These factors are components that represent the system's performance according with predefined criteria.

QEF involves three concepts (Escudeiro and Bidarra 2008):

- **Dimension** - Calculated from the values of importance and compliance of the factors that compose it;
- **Factor** - A factor is composed by two elements:
  - Importance - The weight of importance is calculated from two components: the sum of the relevance of the requirements that compose the factor and the total of those that make up the dimension. Dividing these two values, a percentage corresponding to the relative importance of the factor in the context of the dimension is obtained;
  - Compliance - Calculated from the percentage of total evaluation obtained in the set of requirements that compose the factor.
- **Requirement** - A requirement is composed by two elements:
  - Evaluation - It's a percentage value that represents the rate of compliance with the requirement. Usually it takes the values 0%, 50%, or 100%, but it can be any kind of predefined value between 0% and 100%. These percentages are related with a quality step that needs to be fulfilled to acquire that percentage;
  - Relevance - It's a value between 2 and 10 inclusive, with increments of 2 (2,4,6,8,10), that represents the relevance of the requirement in the context of the factor where it fits, being 2 the less relevant and 10 the most.

System's quality is then measured, in percentage, regarding the fulfillment of the criteria defined for the ideal solution. For that, it's calculated the distance between the real and the ideal solution. The greater this distance, the further the real solution is from being comparable to the ideal solution, which means that the quality is low.

6.4.2 QEF application to the project

To obtain the final quality percentage of the project a QEF model was designed. Next figures present that model and its metrics for each one of factors considered.

q	D	Qi	Dimension	Q_j	W_{ij} (Factor Weight j in Dim i) [0,1]	Factor	rw_{jk} (requirement weight k in Factor j) {2, 4, 6, 8, 10}	Requirement	wf_k % requirement fulfillment k) [0,100]
87%	0,34	92,9972	Functionality	90	0,48	Functional (Refering Use Cases)	10	FF01 - List Jobs	100
							10	FF02 - Get Job by ID	100
							10	FF03 - Create Job	50
							8	FF04 - Update Job	50
							10	FF05 - Delete Job	100
							10	FF06 - Get Job Statistics	100
							8	FF07 - Get Job Error Logs	100
							8	FF08 - Execute Job	100
							8	FF09 - Update Job Statistics	100
							8	FF10 - Store Job Execution Error Log	100
				94,11765	0,38	User Interaction	10	FUI01 - Application is intuitive and presents the same design experience	100
							8	FUI02 - Application presents the same control experience	100
							8	FUI03 - Application presents the same navigation experience	100
							10	FUI04 - Permissions and user type specific options are guaranteed	100
							8	FUI05 - Application has quick access to main functions	100
							6	FUI06 - Application has navigation controls	100
							8	FUI07 - Application presents search controls	50
							10	FUI08 - All statistics are well organized and present relevant information on the situation analysis	100
				100	0,14	Content Quality	8	FCQ01 - All job information is well organized	100
							8	FCQ02 - The texts are well written and all the setences make perfect sense	100
							8	FCQ03 - All the messages are easy to understand and human personified	100
		68,8889	Adaptability and Supportability	61,11111	0,80	Maintainability	8	ASM01 - Application should be implemented using clean code, following best practices and design patterns	100
							10	ASM02 - Application should have a good code coverage	0
							8	ASM03 - Application should have useful comments and only when essential to understand the source code/behaviour	50
							10	ASM04 - Application presents a set of interfaces, adding the possiblity of introducing new features	100
				100	0,20	Scalability	10	ASS01 - System should be designed and implemented with maximum capability to scale	100
		87,5	Reliability	100	0,25	Errors	10	RE01 - The system should trigger again the jobs that were already executed but failed if the job was configured with a number of retries greater than zero	100
				83,33333	0,75	Navigation	8	RN01 - Application has a good structure and allows users to access contents in a intuitive way to the main functions	100
							8	RN02 - Application user interface is quick and fast responsible, with progress information	50
							8	RN03 - Application runtime does not have errors, and unexpectable errors should be well treated	100

Figure 6.1: Designed QEF model and final quality evaluation.

Dimension	Functionality
Factor	Functional

		Wfk - Fullfilment (%)		
Requirement	**Metric Evaluation**	**0**	**50**	**100**
FF01 - List Jobs	User can view a jobs list and perform some actions over them (edit, delete, statistics, error logs)	No access to functionality	Partial access to the functionality (actions not available and/or list isn't paginated)	Access to the functionality
FF02 - Get Job by ID	When clicking a button to view/edit job the user must see a form with the job stored data	No access to functionality	-	Access to the functionality
FF03 - Create Job	User can create a job and persist it	No access to functionality	Partial access to the functionality (not all job attributes are configurable)	Access to the functionality
FF04 - Update Job	User can view and update and existing job and persist it	No access to functionality	Partial access to the functionality (not all job attributes are configurable)	Access to the functionality
FF05 - Delete Job	User can delete a job	No access to functionality	-	Access to the functionality
FF06 - Get Job Statistics	User is able to view job's stattistic about number of executions, number of success/failures	No access to functionality	-	Access to the functionality
FF07 - Get Job Error Logs	User can view the error logs if they exist	No access to functionality	-	Access to the functionality
FF08 - Execute Job	System triggers the job and job's command are executed	System doesn't perform functionality	Partial access to the functionality (if the job has a number of retries greater than zero and its execution failed and it is not triggered again)	System performs the functionality
FF09 - Update Job Statistics	System is able to update job's statistic after a trigger	System doesn't perform functionality	-	System performs the functionality
FF10 - Store Job Execution Error Log	System stores error events/logs after job's trigger and failure	System doesn't perform functionality	-	System performs the functionality

Figure 6.2: Metrics and fulfillment percentage description for Functional factor.

Dimension	Functionality
Factor	User Interaction

		Wfk - Fullfilment (%)		
Requirement	Metric Evaluation	0	50	100
FUI01 - Application is intuitive and presents the same design experience	The design experience must be similar to the existing platform - backgrounds, colors, design of buttons, font types, logos, icons	Low similarity	Only half of the categories present a similar design	High similarity
FUI02 - Application presents the same control experience	All form screens must present a cancel/go back button and also a sidebar	No	-	Yes
FUI03 - Application presents the same navigation experience	The navigation between screens present the same experience. Buttons to menu, advance, confirm and cancel always in the same place.	No	-	Yes
FUI04 - Permissions and user type specific options are guaranteed	Functionalities must be available only to apropriated users.	One or more functionalities available to users without those permissions	-	Functionalities only available to users with the required permissions
FUI05 - Application has quick access to main functions	Executing a functionality shouldn't take more than 3 actions (clicks, selections,...)	>3 actions	-	<= 3 actions
FUI06 - Application has navigation controls	Forms should have a cancel and go back button, and in all screens the menu/sidebar should be visible	No buttons presented	Only half of the screens present those buttons	Buttons presented
FUI07 - Application presents search controls	Applications present a search bar	No search bar available	Search bar available but has some issues	Search bar available and works well
FUI08 - All statistics are well organized and present relevant information on the situation analysis	Statistics should be available and be visually appelative	No statistics available	Statistics available but not appelative	Statistics available and visually appelative

Figure 6.3: Metrics and fulfillment percentage description for User Interaction factor.

Dimension	Functionality
Factor	Content Quality

		Wfk - Fullfilment (%)	
Requirement	Metric Evaluation	0	100
FCQ01 - All job information is well organized	All job information must be organized and visually appelative	No	Yes
FCQ02 - The texts are well written and all the sentences make perfect sense	The sentences are short and fullfill elementary grammar principles.	No	Yes
FCQ03 - All the messages are easy to understand and human personified	The messages doesn't present inegible codes or similar form of codification.	No	Yes

Figure 6.4: Metrics and fulfillment percentage description for Content Quality factor.

Dimension	Adaptability and Supportability
Factor	Maintainability, Scalability

		Wfk - Fullfilment (%)		
Requirement	Metric Evaluation	0	50	100
ASM01 - Application should be implemented using clean code, following best practices and design patterns	The system should always follow design patterns and best practices when possible and when make sense	No design patterns applyied if made sense to use them	-	Design patterns applyied if made sense to use them
ASM02 - Application should have a good code coverage	Runs unit tests with coverage	Code coverage < 50%	Code coverage between 50% and 80%	Code coverage > 80%
ASM03 - Application should have useful comments and only when essential to understand the source code/behaviour	System should have comments to describe complex algorithms and other logics that are difficult to understand just by reading the source code	No comments available	Not all complex code is commented	All complex code is commented
ASM04 - Application presents a set of interfaces, adding the possiblity of introducing new features	It should be possible to add new features to the system with little effort	It's not possible to add new features	It's possible to add new features with some source code changes	It's possible to add new features with little source code changes
ASS01 - System should be designed and implemented with maximum capability to scale	The system's components should be able to scale horizontally, and none of the system's components/modules can prevent that from happening	System is not able to scale	-	System is able to scale horizontally

Figure 6.5: Metrics and fulfillment percentage description for Maintainability and Scalability factors.

Dimension	Efficiency
Factor	Errors, Navigation

		Wfk - Fullfilment (%)		
Requirement	Metric Evaluation	0	50	100
RE01 - The system should trigger again the jobs that were already executed but failed if the job was configured with a number of retries greater than zero	When a job is triggered and its command is executed but fails, the job should try to be triggered again if it has a number of retries greater than zero	Failed jobs aren't triggered again	Some failed jobs are triggered again	Failed jobs are triggered again
RN01 - Application has a good structure and allows users to access contents in a intuitive way to the main functions	User interface should be intuitive and creating, editing, deleting, accessing statistics and error logs should not take many effort	User interface not intuitive	Use actions aren't intuitive	All actions are intuitive
RN02 - Application user interface is quick and fast responsible, with progress information	All requests from the user interface to the system's API should take less than 400 miliseconds. The user interface should show progress information.	All requests take more than 400 miliseconds and the user doesn't see progress information	Some requests take more than 400 miliseconds and the user doesn't see progress information	All requests are processed in less than 400 miliseconds and the user view progress information
RN03 - Application runtime does not have errors, and unexpectable erros should be well treated	All errors should be treated and the user should be informed with a useful and understandable message if those happen	All or some errors aren't treated	All errors are treated but some have misleading messages	All errors are treated and have a understandable and useful message

Figure 6.6: Metrics and fulfillment percentage description for Reliability and Navigation factors.

All the requirements cover the most important aspects related with the development and final result expected from this software solution, so the quality measure is relative to the project as a whole, not only related with the implementation itself, but also with other aspects like the user interface and its experience for the users.

Regarding the results from the application of the QEF model to this project, from the thirty requirements considered, twenty-four were fully achieved, only five are partially fulfilled, and one was not accomplished.

The dimension with higher quality is the functionality, with approximately 93%. This was possible due to high percentage of fulfillment for almost all the system's functional requirements that were evaluated by the developer and the QA, in fact, only two of them were partially achieved.

The reliability is the second dimension with highest quality, 87.5% more precisely. This value was obtained due to the capability of the system to trigger again failed jobs, if the first execution fails, and if the users configure the amount of retires to be executed. The fact that the error messages are presented to the user with useful information also contributed to this quality percentage.

Adaptability and supportability were evaluated with approximately 68.89% of quality since the code coverage was not completed for whole system, and extra work it also needed on the comments for some source code with more complexity.

The final quality of the software solution is given by the value of q, the first column of the QEF model from Figure 6.1 and, in this case, it is 87%. The global deviance D from the ideal system is 0.34, value that can be observed in the second column of the model.

Considering the hypothesis from the section 6.1 and its predefined percentage of 60%, because the final quality result is 87%, the alternative hypothesis is the one that is verified, so the system is considered viable since it is evaluated with a percentage above of the predefined value.

Chapter 7

Conclusions

This chapter has as its main objective to present the conclusions about this project, in particular, the achieved objectives, limitations, and future work.

7.1 Achieved Objectives

This project consisted in a proof of concept that involved the development of a new distributed job scheduler to later replace the existing one named Eye of Sauron.

The original service had a few reliability and scalability problems, so one of the objectives for this project was to design and implement a new software solution, using a distributed architecture.

Another objective was the development of a user interface to create and manage jobs, and also to have the statistics of those to promote consistency, visibility and control for the admin users of the Jumia MDS platform.

Considering the main goals for this project mentioned in 1.4, it's possible to say that they were achieved. A new microservice was built, which received the name Eye of Sauron v2, that is capable of doing not only the existing features, but it also has new ones, for example, consulting the error logs from a job's execution.

The major problems related with the reliability and scalability from the original Eye of Sauron were taken into consideration when designing and implementing the new microservice. The new system uses a distributed architecture, which was built using several small components, each one capable of being scaled horizontally by adding more instances of the jobs' consumers, and/or vertically by providing more available resources to those components. The design also makes use of a message broker for asynchronous communication inside the service, and a relational database for storing the jobs, statistics, logs, and the job's historical data.

A HTTP or Bash job can be created in the new service with different types of schedule options - immediate or delayed. Immediate jobs are triggered right after they are stored in the database. Delayed jobs are the ones that have a date and time configurable by the users to be triggered later, for example, in two hours from the moment were they are created in the user interface.

The jobs that have the last type mentioned above are then obtained by a component named Watcher. This component queries the database, every ten seconds, to get the jobs that need to run in that interval of time. For each one of them, the Watcher produces a message

into a Kafka topic. Listening to that Kafka topic is one or more instances of a delayed jobs consumer that will execute those jobs.

In the new system, the jobs are executed by different consumers, it depends if they are immediate or delayed. This separation allows to implement different behaviors according to each job schedule type, and it also grants that the execution of immediate jobs is not affected by a problem with the delayed consumers (and vice-versa).

All the jobs can also have recurrence, it's just necessary to select in the job's form the interval of time between runs and the number of repetitions that the users want to trigger again the same job after its first execution.

The admin users from Jumia MDS platform are now capable of consulting the statistics and error logs from the job's execution, something that was not possible until this project was developed, and that is a major contribution for those users' daily workflow.

Having a reliable job scheduler is considered a powerful tool in the company since it offers a lot of possibilities, not only for marketing communications and internal tasks, but also to help in the automation of countless existing and future processes.

7.2 Limitations and Future Work

Although the project was successful, there are some aspects that could be addressed to improve or complement the work that was done.

One of the limitations at the moment is the lack of deep testing of the software solution that was implemented with this proof of concept because the service wasn't deployed into a testing environment yet. To proceed with proper performance and integration tests with the campaigns service (Butterfly), this is the first thing that needs to be done now.

Also related with testing, the code coverage of the new system needs to be improved.

Another limitation is the Watcher for delayed jobs. Considering the implementation that was done with this project, the Watcher is a component that queries the database to get the jobs that have next run at in the last ten seconds. If this component has a problem and goes down, during the time until it restarts and starts its operation again, there may be jobs that have not been executed and that will no longer be executed as the Watcher always looks for jobs that have to be executed in the last ten seconds, so this an important topic that needs to be addressed after the deployment and general testing of the service.

There is another aspect which is the addition of retry and error Kafka topics. During the execution of an immediate or delayed job errors can happen. Despite retries for failed jobs already exist, that only happens if the amount of times to retry is configured when creating or updating a job, and that option is not currently available in the user interface, it can only be done by calling the API by one of the developers. One of the possibilities can be the emission of a message to a retry topic, with an exponential wait time between retries, to execute the same job again until it reaches a common number of retries for all jobs or it succeeds. If the maximum number of retries is reached, the job message would move to an error Kafka topic.

Bibliography

Abdel-Basset, Mohamed et al. (2019). "A Hybrid Plithogenic Decision-Making Approach with Quality Function Deployment for Selecting Supply Chain Sustainability Metrics". In: *Symmetry* 11.7. (Accessed on 01/29/2022). issn: 2073-8994. doi: 10.3390/sym11070903. url: https://www.mdpi.com/2073-8994/11/7/903.

Academic (2021). *Windows Task Scheduler*. (Accessed on 02/11/2022). url: https://en-academic.com/dic.nsf/enwiki/11601281.

AirbnbEng (Mar. 2013). *Chronos: A Replacement for Cron*. Ed. by Florian Leibert, Harry Shoff, and Andy Kramolisch. (Accessed on 01/08/2022). url: https://medium.com/airbnb-engineering/chronos-a-replacement-for-cron-f05d7d986a9d.

Arguç, Mertcan (Dec. 2021). *Redis vs Kafka vs RabbitMQ*. (Accessed on 02/11/2022). url: https://medium.com/@mertcanarguc/redis-vs-kafka-vs-rabbitmq-e935ebbc7ec.

AWS (2022). *What is apache kafka?* (Accessed on 02/11/2022). url: https://aws.amazon.com/msk/what-is-kafka/.

Baird, Andrew et al. (2022). *Three-tier architecture overview*. (Accessed on 06/21/2022). url: https://docs.aws.amazon.com/whitepapers/latest/serverless-multi-tier-architectures-api-gateway-lambda/three-tier-architecture-overview.html.

Chan, Lai-Kow and Ming-Lu Wu (2002). "Quality function deployment: A literature review". In: *European Journal of Operational Research* 143.3. (Accessed on 01/29/2022), pp. 463–497. issn: 0377-2217. doi: https://doi.org/10.1016/S0377-2217(02)00178-9. url: https://www.sciencedirect.com/science/article/pii/S0377221702001789.

Chmiel, Marjee (2019). *JavaScript*. (Accessed on 02/18/2022). url: https://eds.s.ebscohost.com/eds/detail/detail?vid=3&sid=17dfe29c-da15-48b9-bb9b-75ae72b52bc4%40redis&bdata=JkF1dGhUeXBlPWlwLHNoaWIsdWlkJmxhbmc9cHQtcHQmc2l0ZT1lZHMtbG
AN=89550593&db=ers.

Cloudflare (2022). (Accessed on 06/24/2022). url: https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/.

Davidovič, Štěpán and Kavita Guliani (2015). *Reliable Cron across the Planet*. (Accessed on 02/10/2022). url: https://queue.acm.org/detail.cfm?id=2745840.

De Lauretis, Lorenzo (2019). "From monolithic architecture to microservices architecture". In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, pp. 93–96.

Dkron.io (2015a). *Dkron vs. Other Software*. (Accessed on 02/10/2022). url: https://dkron.io/intro/dkron_vs_other_software/.

– (2015b). *What is Dkron*. (Accessed on 02/10/2022). url: https://dkron.io/intro/.

Escudeiro, Paula and José Bidarra (Jan. 2008). "Quantitative Evaluation Framework (QEF)". In: *Conselho Editorial/Consejo Editorial*, p. 16.

Evans, Eric and Addison Wesley (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. (Accessed on 06/20/2022). url: https://sd.blackball.lv/library/Domain-Driven_Design_-_Tackling_Complexity_in_the_Heart_of_Software.pdf.

Fang, Andy (Dec. 2019). *Dynein: Building an Open-source Distributed Delayed Job Queueing System*. (Accessed on 02/12/2022). url: https://medium.com/airbnb-engineering/dynein-building-a-distributed-delayed-job-queueing-system-93ab10f05f99.

golang-standards (Mar. 2022). *Standard Go Project Layout*. (Accessed on 06/17/2022). url: https://github.com/golang-standards/project-layout.

Gos, Konrad and Wojciech Zabierowski (2020). "The comparison of microservice and monolithic architecture". In: *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*. IEEE, pp. 150–153.

Graf, Albert and Peter Maas (2008). "Customer value from a customer perspective: a comprehensive review". In: *Journal für Betriebswirtschaft* 58.1, pp. 1–20.

Hammink, John (Mar. 2018). *The Types of Modern Databases*. (Accessed on 02/19/2022). url: https://www.alooma.com/blog/types-of-modern-databases.

IBM, Cloud Education (Oct. 2020). *What is Three-Tier Architecture*. (Accessed on 06/20/2022). url: https://www.ibm.com/cloud/learn/three-tier-architecture.

International, Quality-One (Dec. 2017). (Accessed on 02/01/2022). url: https://quality-one.com/qfd/.

Jinzhu (June 2022). *GORM Guides*. (Accessed on 06/23/2022). url: https://gorm.io/docs/index.html.

Kazanov, Vladimir (Apr. 2020). *Cron in Linux: history, use and design - Bumble Tech - Medium*. (Accessed on 02/10/2022). url: https://medium.com/bumble-tech/cron-in-linux-history-use-and-structure-70d938569b40.

Kerrisk, Michael (Aug. 2021). *crontab(5) - Linux manual page*. (Accessed on 02/10/2022). url: https://man7.org/linux/man-pages/man5/crontab.5.html.

Koen, Peter et al. (2002). *FuzzyFrontEnd: Effective Methods, Tools, and Techniques*.

Kotler, Philip and Kevin Lane Keller (n.d.). *Marketing management 14E*. isbn: 9780132102926. url: http://eprints.stiperdharmawacana.ac.id/24/1/%5BPhillip_Kotler%5D_Marketing_Management_14th_Edition%28BookFi%29.pdf.

Leibert, Florian et al. (2017). *Chronos: A fault tolerant job scheduler for Mesos which handles dependencies and ISO8601 based schedules*. (Accessed on 02/11/2022). url: https://mesos.github.io/chronos/.

Lockhart, Luke (2019). *Java programming language*. (Accessed on 02/18/2022). url: https://eds.s.ebscohost.com/eds/detail/detail?vid=1&sid=17dfe29c-da15-48b9-bb9b-75ae72b52bc4%40redis&bdata=JkF1dGhUeXBlPWlwLHNoaWIsdWlkJmxhbmc9cHQtcHQmc2l0ZT1lZHMtb
AN=89550592&db=ers.

McFarlane, Donovan A (2013). *The Strategic Importance of Customer Value*. (Accessed on 01/15/2022). url: https://digitalcommons.kennesaw.edu/amj/vol2/iss1/5/?utm_source=digitalcommons.kennesaw.edu%2Famj%2Fvol2%2Fiss1%2F5&utm_medium=PDF&utm_campaign=PDFCoverPages.

Meyerson, Jeff (2014). (Accessed on 02/18/2022). url: https://eds.p.ebscohost.com/eds/detail/detail?vid=2&sid=5bba8e78-8830-445a-bf0e-56c877958011%40redis&bdata=JkF1dGhUeXBlPWlwLHNoaWIsdWlkJmxhbmc9cHQtcHQmc2l0ZT1lZHMtbGl2ZSZzY29wZT1zaXR
AN=edseee.6898707&db=edseee.

Mohn, Elizabeth (2022). (Accessed on 02/18/2022). url: https://eds.s.ebscohost.com/eds/detail/detail?vid=6&sid=17dfe29c-da15-48b9-bb9b-75ae72b52bc4%40redis&bdata=JkF1dGhUeXBlPWlwLHNoaWIsdWlkJmxhbmc9cHQtcHQmc2l0ZT1lZHMtbGl2ZSZzY29wZT1zaXR
AN=87322817&db=ers.

Mongodb.com (2020). *Introduction to MongoDB*. (Accessed on 02/19/2022). url: https://docs.mongodb.com/manual/introduction/#key-features.

Mysql (2022). *About MySQL*. (Accessed on 02/19/2022). url: https://www.mysql.com/about/.

NGINX (Jan. 2022). *What is a reverse proxy server?* (Accessed on 06/24/2022). url: https://www.nginx.com/resources/glossary/reverse-proxy-server/.

Node.js (2014). *About Node.js*. (Accessed on 02/18/2022). url: https://nodejs.org/en/about/.

Osterwalder, Alexander and Yves Pigneur (2003). "Modeling Value Propositions in E-Business". In: *Proceedings of the 5th International Conference on Electronic Commerce*. ICEC '03. (Accessed on 01/20/2022). Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, pp. 429–436. isbn: 1581137885. doi: 10.1145/948005.948061. url: https://doi.org/10.1145/948005.948061.

Osterwalder, Alexander, Yves Pigneur, et al. (2015). *Value proposition design: How to create products and services customers want*. Vol. 2. John Wiley & Sons.

Ponce, Francisco, Gastón Márquez, and Hernán Astudillo (2019). "Migrating from monolithic architecture to microservices: A Rapid Review". In: *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE, pp. 1–7.

Postgresql (2022). (Accessed on 02/19/2022). url: https://www.postgresql.org/about/.

Quest, Kyle C (Sept. 2017). *Go Project Layout*. (Accessed on 06/17/2022). url: https://medium.com/golang-learn/go-project-layout-e5213cdcfaa2.

Rabbitmq.com (2022). *What can RabbitMQ do for you?* (Accessed on 02/11/2022). url: https://www.rabbitmq.com/features.html.

Ravald, Annika and Christian Grönroos (Feb. 1996). "The value concept and relationship". In: *European Journal of Marketing* 30, pp. 19–30. doi: 10.1108/03090569610106626. url: https://www.researchgate.net/profile/Christian-Groenroos/publication/235270562_The_value_concept_and_relationship/links/0deec5224f05e75b8f000000/The-value-concept-and-relationship.pdf.

Redis.io (2022). *Introduction to Redis*. (Accessed on 02/19/2022). url: https://redis.io/topics/introduction.

Rich, Nick and Matthias Holweg (2000). "Value analysis". In: *Value engineering*.

Stenberg, Jan (Dec. 2019). *Dynein – an Asynchronous Background Job Service from Airbnb*. (Accessed on 02/12/2022). url: https://www.infoq.com/news/2019/12/dynein-job-queue-airbnb/.

TIOBE (Feb. 2022). *TIOBE Index for February 2022*. (Accessed on 02/17/2022). url: https://www.tiobe.com/tiobe-index/.

Tripathi, Adarsh (July 2021). *Best Front End Frameworks for Web Development of 2021: The Complete Guide*. (Accessed on 02/11/2022). url: https://medium.com/geekculture/best-front-end-frameworks-for-web-development-of-2021-the-complete-guide-ec30098fd1d0.

Ulaga, Wolfgang and Samir Chacour (2001). "Measuring Customer-Perceived Value in Business Markets: A Prerequisite for Marketing Strategy Development and Implementation". In: *Industrial Marketing Management* 30.6. (Accessed on 01/09/2022), pp. 525–540. issn: 0019-8501. doi: https://doi.org/10.1016/S0019-8501(99)00122-4. url: https://www.sciencedirect.com/science/article/pii/S0019850199001224.

White, Steven and Michael Satran (Aug. 2019). "About the Task Scheduler - Win32 apps". In: *Microsoft.com*. (Accessed on 02/11/2022). url: https://docs.microsoft.com/en-us/windows/win32/taskschd/about-the-task-scheduler.

White, Steven, Michael Satran, and Drew Batchelor (Aug. 2021). "Task Actions - Win32 apps". In: *Microsoft.com*. (Accessed on 02/11/2022). url: https://docs.microsoft.com/en-us/windows/win32/taskschd/task-actions.

Woodall, Tony (Jan. 2003). "Conceptualising 'Value for the Customer': An Attributional, Structural and Dispositional Analysis". In: *Academy of Marketing Science Review* 12. (Accessed on 01/15/2022). url: https://www.researchgate.net/publication/228576532_Conceptualising_%E2%80%99Value_for_the_Customer%E2%80%99_An_Attributional_Structural_and_Dispositional_Analysis.