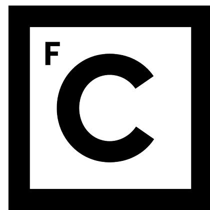
UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA



MICROSERVICES BASED ARCHITECTURE AND MOBILE APPLICATION TO SUPPORT CREW AND VESSEL INSPECTIONS

Pedro Prates Sobreira

MESTRADO EM ENGENHARIA INFORMÁTICA

Dissertação orientada por:
Prof. Doutor Mário João Barata Calha

2023

Acknowledgments

The accomplishment of this work wouldn't be possible without the collaboration of numerous people and to them, I would like to express gratitude.

First I would like to thank all my family, my parents, brother, grandparents and specially to my aunt for their understanding and support throughout all these years.

Secondly, I would like to acknowledge my dissertation supervisor, Prof. Mário Calha for all his guidance, insight, support and the reviews from the early stages of development of this work and for sharing his knowledge that has made this thesis possible.

Also, I would like to acknowledge the Drª Cláudia Lauro and the DGRM team, that greatly contributed for the quick understanding of the DGRM systems and all the needs regarding information and accesses.

Last but not least, to all my friends and colleagues that helped me achieve this milestone and complete this journey.

To each and every one of you – Thank you.

Dedicatória.

Abstract

With the ever increasing importance of the maritime services around the world, the need to control and monitor ports and vessels is born, thus allowing to increase/improve the level of productivity, reliability, safety and security in this field. When it comes to safety and security, vessel monitoring is one of the most important parts that enables the respective authorities to verify and validate the vessels, their crews, and their missions through vessel inspections.

These vessel inspection missions, as they can be carried out in various areas of the coastal zone, are subject to limitations that are not encountered in normal situations, such as adverse weather conditions or lack of connection to the network and therefore to the servers that support these types of inspections and store the relevant information. Another limitation that arises from this lack of connection, is the secure authentication of the inspectors and maintaining the access to the information.

Also due to the increase in the number of vessels, there may be scalability problems with the backend systems.

To help solve these problems, a backend architecture based on microservices and a mobile application were developed to support the inspectors by providing all the information, in a secure way, that is needed to perform the inspections, whether the inspector is in areas that have, or not, access to the network (online or offline).

The developed architecture consists of several independent microservices, deployed through a Kubernetes cluster, and that supports the mobile application used by the inspectors, allowing the inspectors to store and have access to the inspection information about the vessels, crews, vessel licenses and predictions about possible future inspection targets, for a limited period of time after the beginning of the inspection, thus improving security.

Keywords: Microservices, Mobile Application, Vessel Inspections, Kubernetes

Resumo

Com a crescente importância dos serviços marítimos em todo o mundo, nasce a necessidade de controlar e monitorizar portos e navios, permitindo assim aumentar e aperfeiçoar o nível de produtividade, fiabilidade, segurança e fiscalização neste domínio. Quando se trata de segurança e fiscalização, a monitorização de navios é uma das partes mais importantes que permite às respetivas autoridades verificar e validar os navios e os seus documentos, as suas tripulações, e as suas missões através de inspeções de navios.

Os processos das missões de inspeção, realizadas pela DGRM ou por outras entidades de vigilância e segurança marítima, são assim iniciados com a monitorização das embarcações que navegam na zona costeira portuguesa, podendo ser solicitadas por diferentes motivos, sendo que o mais comum deve-se às práticas de pesca ilegal, não declarada e não regulamentada. As inspeções são levadas a cabo presencialmente pelas autoridades responsáveis, abordando os sujeitos da inspeção e a embarcação definida. As informações sobre a embarcação, como as suas dimensões, se os sistemas de apoio à pesca e navegação (Diário de pesca eletrónico e sistema de monitorização) se encontram a funcionar de acordo com os regulamentos estabelecidos e se as licenças atuais se encontram válidas; e as informações sobre os elementos da tripulação são avaliadas e em caso de infração, os inspetores fazem o seu registo no sistema para futuros processos de contra-ordenações.

Estas missões de inspeção, uma vez que podem ser realizadas em várias áreas da zona costeira, estão sujeitas a limitações que não são encontradas em situações normais e que podem levar à dificuldade da realização da comunicação entre os terminais móveis e o sistema de backend, tais como condições meteorológicas adversas, como tempestades, chuva e névoa/nevoeiro, que impedem ou dificultam a transmissão dos dados ou a falta de ligação à rede, devido à distância a que os dispositivos se encontram da costa e das torres que auxiliam vários tipos de comunicação que tomamos por garantida quando em terra, e consequentemente, aos servidores que suportam este tipo de inspeções e armazenam a informação relevante às inspeções e aos seus participantes. Estas limitações afetam também o login e a autenticação dos inspectores sendo que é necessária a conexão com os serviços de autenticação.

Existem também outras limitações que podem influenciar o acesso e a utilização aos serviços do backend, um deles sendo a escalabilidade do sistema. Sendo o sistema atual baseado numa arquitetura monolítica, o processo de escalar a arquitetura em caso de aumento de utilizadores e embarcações pode ser difícil e dispendioso, contribuindo negativamente para a sua funcionalidade. Com este estilo de arquitetura nascem também outros desafios, como a criação de uma única base de código que ao crescer põe em causa a manutenção, desenvolvimento e modificações futuras do sistema, tendo de executar essas alterações num período em que o sistema esteja indisponível e inacessível aos utilizadores.

Para ajudar a resolver estes problemas, foi desenvolvida uma arquitetura backend baseada em microserviços e uma aplicação móvel para apoiar os inspetores, fornecendo toda a informação que é necessária para realizar as inspeções de forma segura, quer o inspetor se encontre em áreas com ou sem acesso à rede (online ou offline).

Com o intuito de desenvolver os sistemas mencionados, foram delimitados alguns casos principais que orientaram o desenvolvimento. O sistema deveria permitir que os utilizadores (inspetores): fossem autenticados através do dispositivo móvel utilizado nas inspeções; iniciassem um processo de inspeção prevendo trajetórias de possíveis embarcações a inspecionar no futuro, além da embarcação principal a ser investigada; adicionassem e editassem informação do relatório de inspeção; tivessem acesso aos dados da inspeção, quer sejam os dados obtidos no início da inspeção ou os dados recolhidos no decorrer da mesma; validassem documentos físicos (ex.: licenças de pesca) a bordo das embarcações; tivessem acesso às verificações e validações realizadas durante a inspeção; e ainda que conseguissem partilhar informação recolhida durante a inspeção com outros inspetores, elementos da equipa de inspeção;

Para apoiar estes elementos que compõem a inspeção, os microserviços desenvolvidos, onde o deployment é realizado com o apoio da ferramenta para orquestração de containers de software, Kubernetes, oferecem informação através de APIs REST decompostas em entidades definidas, como Embarcações, Pessoas Singulares e Coletivas, Licenças, Requisições de embarcações, Inspeções e Previsões. A arquitetura do backend é ainda composta por Bases de Dados que apoiam e guardam os dados expostos através dos microserviços e um Controlador Ingress que serve para como um balanceador de carga para o sistema, redirecionando os pedidos http realizados para os microserviços corretos.

Em relação à aplicação móvel que irá apoiar os inspetores, esta permite que inspeções sejam realizadas guardando ambos os tipos de informações relativas à inspeção, informações adquiridas aquando do início da inspeção (onde este é, por norma, realizado num local com acesso à rede) ou informações adquiridas durante o decorrer da inspeção. Permite ainda realizar, em ambos os modos online e offline,

a autenticação do inspetor, a verificação e validação de documentos e licenças de pesca e a agregação e edição da informação sobre a inspeção. Para manter a segurança dos dados armazenados durante todo o processo, o login e a autenticação do inspetor na aplicação deverá ser feita inicialmente online e limitada a um período de tempo específico em caso de se encontrar offline, protegendo assim a informação e prevenindo accessos indesejados ao sistema.

Assim, através da arquitetura baseada em microserviços, foi criado um sistema que permite à DGRM melhorar a disponibilidade e escalabilidade, oferecendo formas mais simples e fáceis de realizar atualizações, modificações e manutenções, sem que o sistema sofra com longos períodos em que se encontra em baixo. Através da aplicação móvel obtemos uma forma mais atual e automatizada de realizar as inspeções às embarcações e de validar e verificar as informações obtidas, por um custo mais baixo que o atual e mais rápida e eficazmente. Ainda assim, existem ainda algumas limitações ao sistema desenvolvido, principalmente nos casos de partilha de informação entre dispositivos móveis de inspetores, nos mecanismos de autenticação utilizados pela DGRM (que não foram implementados devido a questões de privacidade mas que poderão ser implementados no futuro) e finalmente existem limitações no microservicço que fornece previsões de possíveis embarcações a inspecionar. Fornecemos assim discussões sobre os temas e uma base para que estas questões em aberto possam ser desenvolvidas e melhoradas em futuros trabalhos.

Palavras-chave: Microserviços, Aplicação Móvel, Inspecções de Navios, Kubernetes

Contents

List of Figures **xiv**

List of Tables **xvii**

1 Introduction **1**

1.1 Motivation . . . 2
1.2 Goals and Contributions . . . 3
1.3 Organization of the Document . . . 3

2 DGRM - Direção Geral dos Recursos Marítimos **5**

2.1 Current Vessel Inspections . . . 9

3 Literature Review **13**

3.1 Evolution of the Web . . . 13
3.2 Cloud Computing . . . 14
3.3 Architectural Styles . . . 18
3.3.1 Monolithic Architecture . . . 19
3.3.2 Service Oriented Architecture - SOA . . . 20
3.3.3 Microservices . . . 21
3.4 Monolithic vs SOA vs Microservices . . . 22
3.5 Cloud Native Application . . . 23
3.6 Cloud and Mobile Computing . . . 24
3.7 Microservices . . . 26
3.8 Virtualization . . . 27
3.9 Container Orchestration Tools . . . 28
3.9.1 Kubernetes . . . 30
3.10 Security . . . 31
3.10.1 Data Privacy . . . 33
3.10.2 Authentication . . . 33

4 Architecture for crew vessel verification **35**

4.1 User Stories . . . 38

4.2 Requirements 39
4.2.1 Functional Requirements 40
4.2.2 Non-functional Requirements 41
4.3 New Vessel Inspection Mission 42
4.4 Architecture 44
4.4.1 Frontend 47
4.4.2 Backend 54
4.5 Inspection Predictions 67
4.5.1 Predictions Models 68

5 Implementation and Results 71
5.1 Frontent implementation 71
5.2 Backend implementation 72
5.3 Results 74
5.3.1 Mobile application adaptability to the network connectivity . 75
5.3.2 Usability 80
5.3.3 Performance 84

6 Conclusion and Future Work 89
6.1 Limitations 90
6.2 Future Work 91

A Views regarding the frontend mobile application and its architecture 93

B Tables regarding the backend MySQL databases 99

Acronyms 103

Bibliography 111

List of Figures

2.1 DGRM Areas and Sub-areas of operation 7
2.2 DGRM architecture overview diagram (diagram provided by the DGRM in the supporting documentation of the current system) 8

3.1 Main service models of Cloud Computing 17
3.2 Comparison between Monolithic, Service Oriented and Microservices Architecture 23
3.3 Kubernetes Architecture Example 32

4.1 Vessel inspection mission flow 43
4.2 General architecture 44
4.3 Frontend Decomposition View 48
4.4 Frontend Data model 50
4.5 Backend Architecture 57
4.6 Kubernetes elements that compose the microservices databases and their relations 65
4.7 File mysql-deployment.yaml containing the specification of the Database for the Embarcação/Vessel Microservice 66

5.1 File microservice-embarcacao.yaml containing the specification of the Microservice Embarcação/Vessel 74
5.2 Flow of different states during the authentication case 76
5.3 Application screens regarding the online/offline authentication process. Normal authentication Screen, Biometric authentication Screen, Network error in authentication (from left to right) 77
5.4 Application screens regarding the verification and validation of documents/licenses process. Valid license verification, Invalid license verification and Invalid QRCode verification (from left to right) 78
5.5 Application main screen adaptability regarding changes in the network connection. Main screen, with inspection not started and started, and network warning while starting inspection without network connectivity (from left to right) 79

5.6 Application screens adaptability to the network connection regarding the ending of the inspection process. General information screen to end inspection and error message while ending inspection without network connectivity (from left to right) 80

A.1 Frontend Decomposition View Detailed 94
A.2 Frontend Uses View 95
A.3 Frontend Components and Connectors View 96
A.4 Mobile application initiate inspection screen, without inspections started and with one inspection already started (from left to right) 97
A.5 Mobile application select vessel to inspect/predictions screen. Different examples of interactions: the default, presents error when no vessel is selected to inspect and when one vessel is selected (from left to right) 97
A.6 Mobile application final screens of the initiate inspection process. Choose requisitions port screen, Requisitions screen and Add inspector screen (from left to right) 98

List of Tables

4.1 System functional requirements 40
4.2 System non-functional requirements 41
4.3 Summary of the developed microservices details (ports, paths and HTTP methods) 62
4.4 Summary of the developed databases details (DB ports, kubernetes database services and the supported microservice) 67

5.1 Table summarizing the testers feedback regarding the mobile application usability 83
5.2 Table containing the measured authentication screens load times . . . 85
5.3 Table containing the measured licenses verification and validation screens load times 85
5.4 Table containing the measured starting and ending inspection screens load times 85

B.1 Table from the Database that supports the Embarcação Microservice 99
B.2 Table from the Database that supports the PessoaSingular Microservice 100
B.3 Table from the Database that supports the PessoaColetiva Microservice 100
B.4 Table from the Database that supports the Licença Microservice . . . 100
B.5 Table from the Database that supports the Requisições Microservice . 100
B.6 Table from the Database that supports the Inspeção Microservice . . 101
B.7 Table from the Database that supports the Inspeção Microservice, storing information about the inspected vessels 101
B.8 Table from the Database that supports the Inspeção Microservice, storing information about the collective persons 101
B.9 Table from the Database that supports the Inspeção Microservice, storing information about the scanned documents 102
B.10 Table from the Database that supports the Inspeção Microservice, storing information about the crew members and inspectors 102
B.11 Table from the Database that supports the Previsões Microservice . . 102

Chapter 1

Introduction

With the rapid globalization and increasing demand for maritime services over the world, like transportation, fishing and energy, maritime safety and regulation has attracted huge attention in both commercial and marine sectors during the past decade. The International Maritime Organization (IMO) estimates that over 80% of the world's trade is carried by sea and this high volume of maritime activity makes maritime situational awareness, surveillance, inspection and security, important areas of interest in this invaluable sector.

In addition to cargo and transportation vessels, there are also other types of vessels, such as fishing vessels, that need to be monitored and inspected. Inspections are usually aimed at verifying and validating these vessels' fishing licenses, verifying the fishing areas assigned to the vessel, and in certain cases verifying the amount of fish caught, and if the species corresponds with the one licensed and authorized.

Inspections likes the one described are usually carried out with minimal support of the mainland systems, only receiving limited information about the inspection targets. Another aspect that hinders the proper conduct of these inspections is the fact that they may be carried out unexpectedly or on a scheduled basis, and there may be no prior preparation, which creates a situation where there may be no access to data from new vessels to be inspected.

This lack of information can be detrimental to the correct fulfillment of inspections and thus an improvement in this process is required.

To revamp the current system and provide adequate support to inspectors during the inspections, with relevant information about the vessels in need to be inspected, with possible predictions of vessels that could be flagged for inspection in the near future and that are also in the vicinity of the inspectors (based on data collected from previous inspections and vessels trajectories), and with an easier way to request, provide and store that information in a secure location, the Portuguese entity responsible for these inspections, the Direção Geral dos Recursos Marítimos (DGRM), requested a new system to enhance the process.

In response to this request, a mobile application was developed that allows inspectors to store and access all the necessary information for the inspection, in a secure way, either in online or offline modes. To support the application, it was also developed a backend system, based on a microservices architecture[22], that provides the opportunity for the DGRM services to move in the direction of a more scalable and resilient backend, when compared with the current backend that is based on a monolithic architecture. With the increase in the number of services provided by the DGRM a monolithic backend becomes hard to maintain and to rely upon, since its availability can be compromised.

1.1 Motivation

The evolution of cloud technologies, in which the microservices architectures thrive, has promoted the development of applications that can easily be adapted to the deploying environment and are compatible with different types of devices, operating systems and data storage types, thus improving the performance and flexibility of these applications. Developments in these areas and related fields, have improved systems scalability and development processes when compared with monolithic systems but have also created some new obstacles and challenges, such as security, privacy and trust concerns, bandwidth and data transfer, management and synchronization, and data heterogeneity discussed in [46], [52]. Some other, more specific, challenges can also arise from external sources to the systems, such as: weather conditions and connectivity to the network.

Currently these challenges are experienced by the maritime authorities in the validation and verification process of vessels and their crews. The current system, built and developed based on a monolithic architecture, is hard to scale vertically, leaving it vulnerable to the increasing number of vessels needing inspection in the Portuguese maritime territory. Also with the current system, inspectors can't have real-time access to the information needed to carry out the inspections since the adverse weather conditions and the distance to the mainland may affect the connection to the network, leaving them regularly offline and unable to access the authentication services and the necessary information about the vessels and their crews. And finally, another difficulty lies in the case of unannounced inspections where is necessary to have information about vessels and crews that were not scheduled for the inspections.

Satellite communications can be a solution to this problem but are often too expensive and thus used only in cases of emergency, creating the need for a scalable system that supports the inspectors' mission by giving them access to the necessary data while online or offline.

Considering these issues and needs, we worked with the ”Direção Geral dos Recursos Marítimos” (DGRM) to develop a Microservices based architecture and a mobile application to assist in the crews and vessels verification and that in the future can help in starting the transition to service oriented systems from the current, and existing, monolithic system.

1.2 Goals and Contributions

Given the challenges, the main objectives of this project and contributions are:

- The development and implementation of a mobile application that allows the validation and verification of documents/information, in real time, on board of vessels;
- To perform and maintain the user authentication in the application in a secure way, either in online or offline mode;
- And also, to develop a scalable cloud backend architecture and structure that allows access to the required data through an API;

The main challenge to be considered is related with the current architecture and system: there’s the need to improve the scalability of the existing system, since in the future the number of vessels, and also the number of applications to be made available, is likely to increase; the need to use other types of communication, since satellite communications (currently used) can be expensive; and finally how to maintain/perform authentication on a mobile device (in offline mode) during inspections. One last contribution, and a secondary goal, is the research and discussion of possible implementations of vessel trajectory predictions models to aid the inspectors by providing extra information that could be useful in cases of unannounced inspections to new vessels.

1.3 Organization of the Document

This thesis is organized as follows: 1 we have provided a introduction to the work developed and related topics.

In Chapter 2, it is presented the mission and the goals of the DGRM (Direção Geral dos Recursos Marítimos), as well as introducing the main thesis problem.

In Chapter 3 we present a more detailed characterization of topics related to the work, such as Cloud Computing, Monolithic Architectures, Service-Oriented Architectures, Microservices and a comparison between them as well as presenting some

advantages and disadvantages for each one. Next, it is also introduced the state of the art about cloud native applications, mobile computing and cloud computing, microservices, virtualization, container orchestration tools, with special focus on Kubernetes, and authentication methods as well as some of its current challenges. After the problem introduction and the literature review, in Chapter 4 is described the system developed, describing the context and the two main divisions of the project, the Frontend (as a mobile application) and the Backend (as a architecture deployed in Kubernetes), the Chapter 5 focuses on describing the tools and technologies used in the development of the architecture and finally in Chapter 6 we draw the conclusions about our work, discuss some of its limitations and expose the work set for the future.

Chapter 2

DGRM - Direção Geral dos Recursos Marítimos

In this section its given an overview of the DGRM, its goals, missions and also a detailed view regarding the current inspection missions.

The DGRM is a central service under the Portuguese State direct administration, endowed with administrative autonomy to act in the Portuguese maritime space and coastal areas and is governed by principles of accountability, transparency, equity, and quality. Its mission is to develop maritime safety and services, including the maritime-port sector, to implement policies on fishing, aquaculture, manufacturing and related activities, the preservation and knowledge of marine resources, and to ensure the regulation and control of activities developed in these areas.

As the main maritime entity, in the Portuguese territory, the DGRM has many different functions and not only focuses on the control and management of vessels and ports, but also handles marine natural resources and industries related with the sea and coastal areas. To perform its main functions, the DGRM can be divided into five distinct entities, all of them encompassed under its authority:

- **Maritime Administration:** The Maritime Administration is responsible for guaranteeing the quality of the registration of vessels flying the Portuguese flag, ensuring compliance with safety and security requirements in new constructions, vessels in service and port facilities, as well as accrediting maritime training and certification schools.

- **National Fishing Authority:** The National Fishing Authority is responsible for exercising the functions of Fishing Authority in accordance with the provisions of the Union's Control System and the system that aims to prevent, deter and eliminate illegal, unreported and unregulated fishing.

- **National Authority for Sea Traffic Control:** The National Authority for Sea Traffic Control is responsible for ensuring the efficiency and effectiveness

of sea traffic control in the areas of intervention of sea traffic control services, issuing regulatory standards, supervising the operation of control centers, and maintaining the certification of technicians, accreditation of training entities, and recognition of courses.

- **Competent Authority for Maritime Transport and Port Protection:** The Competent Authority for Maritime Transport and Port Protection is responsible for guaranteeing and ensuring the highest possible security for the maritime and port sectors, through the introduction of security measures applicable to port facilities and ships on international voyages and on domestic voyages, as defined by law.
- **National Waste Immersion Authority:** The National Waste Immersion Authority is responsible for the selection and geo-referencing of sites for the immersion of dredged materials in the sea, as well as for the environmental monitoring of these sites, and also for sending the OSPAR Commission the annual report on all immersion operations at sea carried out in Portugal, ensuring the maintenance of good environmental status in the marine environment.

Furthermore, the figure 2.1 represents the different areas and sub-areas where the DGRM can operate. Either being, the already presented, *Maritime Administration*, the *Fisheries area* that mainly focus on the sectors of fishing, control of licenses and

inspections of vessels, the industries and markets related to sea products and also aquaculture and saliculture, where producers have to report their early production of salt and species that they cultivate, or the *Planning and Sustainability area* that has the responsibility of monitoring the environment and coastal areas against waste and pollution, to protect and defend fishing ports mainly against natural occurrences by dredging ports on a regular basis, to handle the maritime spatial planning assuring that it is not misused and also to handle marine protected areas and support projects projects that aim to help preserve biodiversity.

To support the different endeavours and areas mentioned above, the DGRM relies on a system that can be broken down into several layers, which is represented in figure 2.2. Regarding he system architecture in place at DGRM, its architecture is based on a monolithic model and is divided into five main layers:

- The **Presentation Layer (BMAR)**: Layer that implements the first line of interaction between Users and the systems, providing access to information content and applications, according to established profiles. This layer is multi-channel and multi-platform, covering all interaction channels provided by DGRM, including face-to-face, web and mobile.

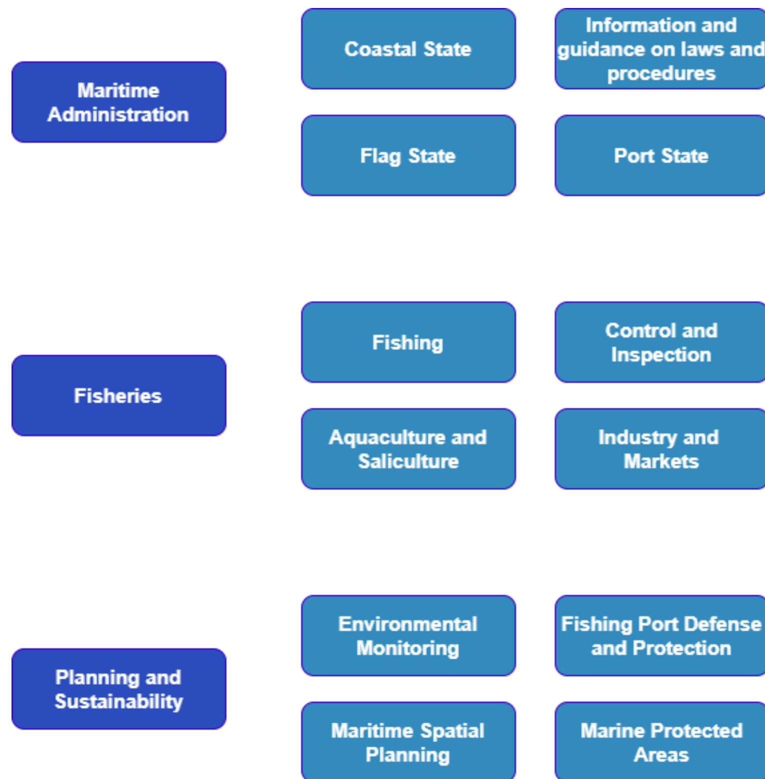


Figure 2.1: DGRM Areas and Sub-areas of operation

- The **Application Layer**: Layer responsible for providing the applications that support the business processes, whether these applications are accessible to the Citizen (front-office) or only to DGRM employees (back-office). Is in this layer that are included the services MONICAP and SIFICAP, that will be explored in the next section 2.1. In this layer are also present interfaces to communicate with external services provided by other governmental and non-governmental entities.
- The **Database Layer**: Layer responsible for storing the entire information repository, from databases to documentary files. This layer supports the operation of the Presentation and Application Layers.
- The **Infrastructure Layer**: The layer that contains the components that support the operation of all application elements at all layers. On premises (physical infrastructure), infrastructure as a service, or hybrid approaches are considered and can be implemented.
- And finally, the **Security, Monitoring and System Administration Layer**: That encompasses the mechanisms and services transversal to the various layers of the Architecture intended to guarantee its correct operation and the continuity of the business processes supported by it.

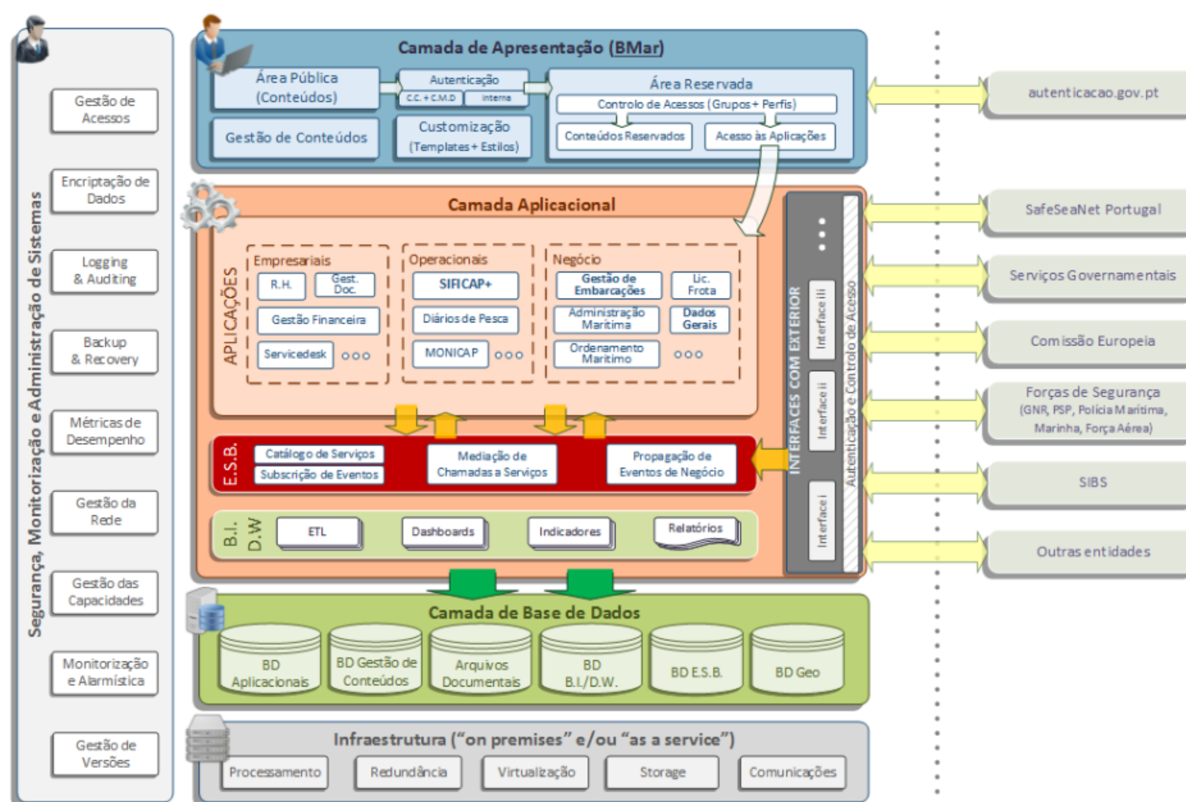


Figure 2.2: DGRM architecture overview diagram (diagram provided by the DGRM in the supporting documentation of the current system)

2.1 Current Vessel Inspections

As mentioned before, one of the many roles of the DGRM is to monitor vessels navigating in the Portuguese coastal zone and in certain cases perform inspections on those vessels. These inspections can be requested by many different reasons, the most common is due to Illegal, Unreported and Unregulated fishing practices that constitute one of the most serious threats to the sustainable exploitation of living aquatic resources and undermines the international efforts to promote better ocean governance while also representing a major threat to marine biodiversity.

Regarding the monitoring of vessels, the DGRM has a main system, SIFICAP, and a sub-system, MONICAP, that assist in these tasks.

- **SIFICAP** is a Monitoring, Control and Surveillance system that integrates subsystems from several entities that participate in it with the purpose of contributing to a better defense and conservation of the resources related to fishing. Its main goals are to:

  - ensure the functional articulation of the entities, aiming to establish, in a timely manner, the conjugation of the various operational means, with a view to a more rational and effective intervention capacity;
  - endow the entities with an integrated system of information and support for the surveillance, supervision and control of fishing activity that allows the flow, in a timely manner, of information of interest to the activity of each of the entities;
  - ensure the processing of the collected data and make the information available to the entities;
  - and provide statistical elements and decision support;

  As for the authorities and entities that are involved, participate and contribute for the proper functioning of the system, are the DGRM, the Portuguese navy, the Portuguese Air Force, the GNR (Guarda Nacional Republicana) and both autonomous regions of the Açores and Madeira, thus encompassing the three domains where the system can be used to act upon, on land, sea and air.

- **MONICAP** is a system with global coverage for the Continuous Monitoring of Fishing Activities integrated in terrestrial (coordination) and aerial components. This system can also be defined as a Vessel Monitoring System (VMS)

and, in most cases where it is implemented and used, comprises a mobile device installed on board of fishing vessels, commonly known as the Blue Box or Continuous Monitoring Equipment, which receives global positioning data and communicates with the CCVP/FMC-PT (Centro de Controlo e Vigilância da Pesca/Fishing Monitoring Centre - Portugal) through the Inmarsat satellite network. It can provide various data, with a programmable periodicity, or immediately upon request from the CCVP (polling), such as: **Location (Latitude and Longitude), Date and Time, Course, Speed** and **equipment management information (alarms, power, battery charge, antenna signal)** that allow, for example, to verify the exercise of fishing activity and operations: without a fishing license or authorization, use of unauthorized fishing gear, in closed or temporarily closed areas and during closed seasons.

With the data acquired from the monitoring, inspections of vessels and their crews can be requested. This inspection requests can be made if there are blatant violations of the law or if there is a suspicion of violations or infractions committed by the crew onboard of the vessels.

Currently, the inspection process begins in the mainland, where available Inspectors receive a request for inspection. The inspectors then coordinate with other relevant entities and authorities via the SIFICAP system. Next, the inspectors travel to the vessel to be inspected and gather data related to possible infractions that the vessel and its crew could be committing. This data focus on examinations of the onboard working systems, such as VMSs (Vessel Monitoring Systems), DPE (Diário de Pesca Eletrónico - Electronic Fishing Logbook) or in paper, the zone and port where the vessel is registered, the type of vessel, the dimensions of different parts of the vessel, the engine power, the minimum and maximum capacity, the fishing gear used and most important, the vessel's license(s). The licenses describe the type of species and quantity that can be caught, the license validity date, the area where its valid and other details and nuances on how the species should be caught (for example, the allowed size of the nets).

The licenses play a major role on the vessel inspection, and in the last years the DGRM has invested in modernizing these documents by developing a new solution for issuing and managing electronic certificates for electronic document signature and authenticity and traceability control. The request, analysis procedure and issuing certificates are totally electronic, in accordance with the International Maritime Organization (IMO) guidelines. With this modernization of these documents to the digital space, inspection and manning agencies can verify the expiry date of the certificates at `www.portugueseflagcontrol.pt`.

After the gathering of information about the vessel and the verification and validation of the licenses, the inspectors create a inspection report detailing the

steps taken, the information acquired, the status of the fishing licenses and, in the case of any irregularities, the evidence which support the claims.

Finally, back in the mainland, when all the information is gathered, including all the information of the different inspectors, the inspection report is sent to the inspection subjects. This stage is necessary to ensure that all the obtained data is correct and to give the opportunity to the inspection subjects to correct it if its not accurate. This process its not immediate and can take several days to conclude, since the information should be sent via email and the inspections subjects can add comments to the report if they believe something is wrong.

After the review by all the participants, the final report is created and the inspection process is concluded.

With this modernization of the process of issuing and managing licenses, the DGRM wants to keep improving and modernizing its infrastructure continuing with the vessel inspection processes. Since currently this process is mainly carried out without technological support can be prone to errors and repetition of information already held by the DGRM. With the introduction of a more technological paradigm and different system architectures these processes could be improved and streamlined in a more dynamic way.

Is then, the main goal of this project, to improve these inspection processes with the development of a mobile application that supports the inspectors in their mission and a new backend architecture based on microservices.

Chapter 3

Literature Review

In this section we start by giving an overview of the cloud computing history and its evolution. Other technologies closely related with this field will also be discussed and how these technologies have benefited and were able to grow from the advancements and developments in cloud computing. Additionally, it will also be provided a background for Cloud Architectures and a comparison between the benefits and disadvantages of 3 different types of architectures. It will also presents the state-of-the-art related to cloud computing and some of the technologies used in this area. The state-of-the-art on microservices and authentication (in the cloud) are also presented. One of the big uses of cloud computing relates to the area of IoT, and how this last one has introduced a nearly infinite number of endpoints to the networks. This trend has made it more challenging to consolidate data and processing in a single data center, giving rise to other cloud paradigms, cloud native applications and mobile cloud computing.

3.1 Evolution of the Web

When speak about the evolution and development of any technology or field is always important to understand its beginnings, foundations and the path that brought them to where they are now, but as we are going to focus on technologies, fields and topics that depend on and were built/developed with the web in mind, it can be a difficult task to make a separation between technologies and the medium in which they were built. So there is another very important topic we should keep in mind, the web itself.

Since its beginnings, in 1989 when Tim Burners-Lee initially made the proposition to create a global hypertext space in which any network-accessible information could be referred to by a single "Universal Document Identifier" (UDI), the main goal of the web was to build a "common information space in which we communicate by sharing information" [65].

From there, it evolved into Web 2.0, towards a technological period more focused on the move to ”the internet as a platform”, its rules and the communities and the interactions between users and services by expanding forms of cooperative production and online information sharing. This shift, combined with the early 2000s internet adoption through computers and mobile devices, allowed the web to grow and explode in popularity. As the web saw more users come online, companies built centralized platforms that would let users publish content/information that other users could consume (e.g. Flickr, YouTube, Wikipedia, Blogger, MySpace).

According to O’Reilly [66], is hard to precisely delimit the boundaries of Web 2.0, being presented as a ”gravitational core” for a set of practices and principles. These core principles were defined as: *The Web As Platform*, *Harnessing Collective Intelligence*, *Data is the Next Intel Inside*, *End of the Software Release Cycle*, *Lightweight Programming Models*, *Software Above the Level of a Single Device* and *Rich User Experiences.*

Finally we reach the most recent state of the web, Web 3.0. In [2] is mentioned that the idea of a semantic web is a old one, dating back to 2006, by John Markof, and even earlier, by Tim Berners-Lee. Semantics web basic idea is to characterize structured data and connect them in more effective discovery, integration, automation and reusable ways, improving data management, support accessibility of mobile internet, among others. We can thus essentially view Web 3.0 as Semantic Web technologies integrated into, or powering, large-scale Web applications.

3.2 Cloud Computing

With the popularization and growth of the web, other fields began to emerge. Cloud Computing was one of them and can be mapped back to older real-time systems which have been used long before cloud computing has come into existence.

In the 1970s, one of the first implementations of symmetric multiprocessing and virtualization was utilized by mainframes dedicated to organizations, researchers and scholars. Mainframes are high-performance computers with large amounts of memory and processors that process billions of simple calculations and transactions in real time, and began to raise concerns regarding its high cost and sparsely intermittent computational needs. To solve this problem, a solution was created that allowed users and external organizations to ”rent” computing resources at much lower costs, helping to improve the efficiency of these expensive systems, reducing idle periods and allowing for greater returns on investment for organizations. Technologies with similar philosophies have been developed trough the years - from Shared and Dedicated Web Hosting [51, 69] to Virtual Private Server (VPS) Hosting [12] and Grid/Utility Computing, all contributed to the the current definition

of Cloud Computing.

By the late 1990s, with the standardization of the internet and its protocols, companies like Salesfores started distributing and hosting customer relationship management software over the internet on a subscription basis, pioneering the fields of hardware virtualization and and cloud computing as we know it today, paving the way for other companies that nowadays hold the monopoly in this area such as: Amazon, Microsoft, Google, IBM and others.

In [7] cloud computing is referred to as encompassing both the applications, the services, made available over the Internet as well as the physical structures, the hardware, that are in the data centers that provide these services. It can also be defined as a model that allows on-demand access, via the internet, to a shared pool of configurable computing resources like networks, servers, storage space, applications, and different services while providing different levels of abstraction. These distinct levels of abstraction grant cloud computing technologies the possibility to be implemented in multiple different styles of architectures, with different services and different deployment models [27].

Regarding the deployment models, in [8] the *Public Cloud Model* is described as being a model based on the pay-as-you-go system and that it is usually available to the general public. In this model, the customer is mainly responsible for what will be sent to the cloud, be it a backup, an application or some files, while the cloud provider is mainly concerned with the maintenance, security and management of all resources. Some public cloud examples that are available today are: Amazon's Elastic cloud, Microsoft's Azure platform, Google's AppEngine and Salesforce. On the other end of the spectrum exists the *Private Cloud Model*, this model is usually dedicated to the needs and goals of a single organization, referring to the internal data centers of a business/organization even though it may be managed or hosted by a third party. "A private cloud has the potential to give the organization greater control over the infrastructure, computational resources, and cloud consumers than can a public cloud" [27]. The occasions that involve a composition of the two models mentioned above, are the *Hybrid Cloud Model*.

Another important aspect and one of the key strengths to consider about cloud computing, are its principal service models [16]:

- **Software as a Service (SaaS)**: SaaS is cloud-hosted, ready-to-use application software. Users pay a monthly or annual fee to use a complete application from within a web browser, desktop client or mobile app. The application and all of the infrastructure required to deliver it - servers, storage, networking, middleware, application software, data storage - are hosted and managed by the cloud provider, managing all the upgrades and patches to the software ensuring a specified level of availability, performance and security. This type

of service is one of the most common since anyone who uses a computers or mobile devices to access emails, social media or cloud file storage solutions (such as Dropbox or Google Drive) is already making use of it. In this type of service consumers typically can interact with the application through a user interface like a Web browser or particular applications.

- **Platform as a Service (PaaS)**: PaaS provides a cloud-based platform for developing, running, managing applications. The cloud services provider hosts, manages and maintains all the hardware and software included in the platform - servers (for development, testing and deployment), operating system (OS) software, storage, networking, databases, middleware, frameworks, development tools - as well as related services for security, operating system and software upgrades, backups and more. Developers are only required to manage the deployed applications and settings of the environment hosting them, being granted access to the layers of Application and Runtime environment, where they can collaborate on the application lifecycle development, including integration, deployment and testing, making use of libraries and programming languages supported by the cloud service provider and other available tools.
- **Infrastructure as a Service (IaaS)**: IaaS is on-demand access to cloud-hosted computing infrastructure - servers, storage capacity and networking resources - that customers can use, configure and provision. The main difference between this and on-premises hardware, is that the cloud service provider hosts, manages and maintains the hardware and computing resources in its own data centers. Customers, in some cases, can choose between virtual machines (VMs) hosted on shared physical hardware or bare metal servers on dedicated physical hardware, and can also configure and operate the services resources via a graphical dashboard, or programmatically through application programming interfaces (APIs).

The figure 3.1 presents the main layers and levels of control of each main service model discussed. There are also other minor service models that are worth mention, such as: *API as a Service (APIaaS)* that is used to manage its own custom APIs and allow applications to connect to 3rd party APIs like Google map or voice search API, *Analytics as a Service (AssS)* provides analytics software over the cloud on the subscription-based model and can be used for Business, Data and also Predictive Analytics, *Data as a Service (DaaS)* is a service model that provides pre-aggregated and pre-calculated data and others like *Database as a Service (DBaaS*, *Function as a Service (FaaS)* and even *Security as a Service (SECaaS)*.

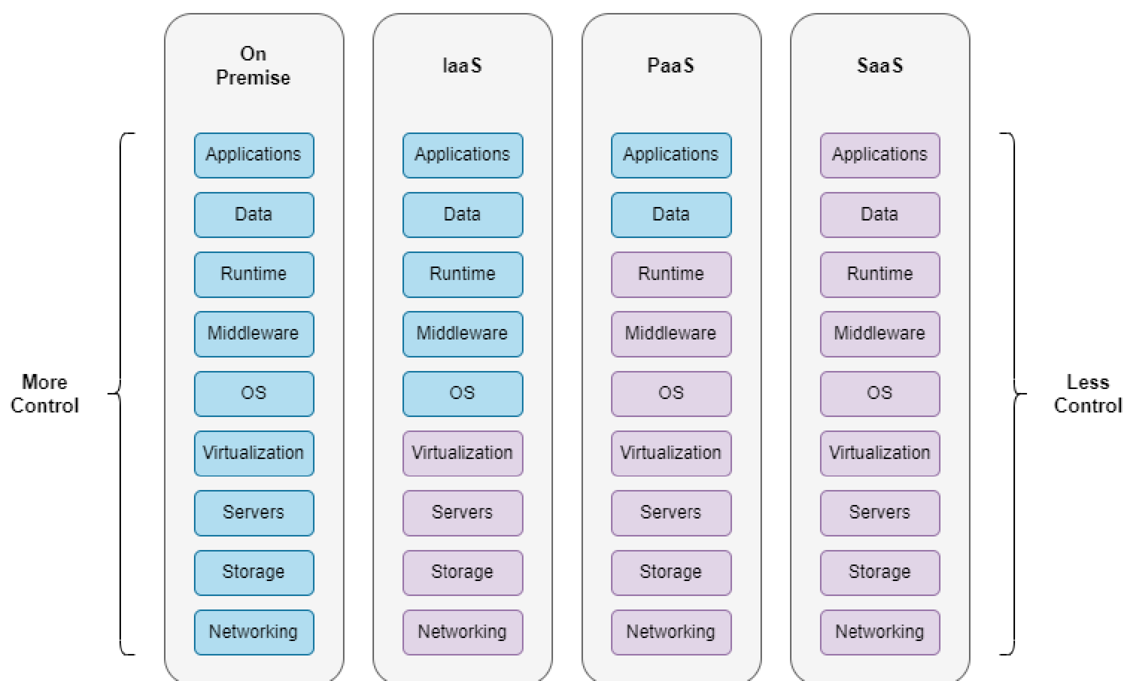


Figure 3.1: Main service models of Cloud Computing

With all these different levels of abstraction and distinct services, many questions about the security of these systems have been presented and discussed over the years. In [27] the upsides and downsides of security and privacy are discussed. The upsides mentioned are the areas where organizations can improve their security, such as: staff specialization, platform strength, resource availability, backup and recovery, mobile endpoints and data concentration.

Some of these advances and advantages greatly benefit certain architectural styles more than other, such as microservices architectures and consequently microservices based applications. For example, they make it possible to restrict access to specific resources and control these accesses giving different levels of permissions and information based on the service connecting to them. Another way to restrict accesses without resorting to authentication and permissions is by hiding specific services from other services and external entities, making use of network isolation. It is also made easier to maintain updated replicas of data automatically, specify events that can save resources by only executing at certain hours or be triggered by other events or tasks (and thus saving extra resources that otherwise would be just unavailable) and modify, by increasing or decreasing, the number of instances running during times where the system reaches peak loads or base loads of traffic.

In contrast, the areas of concern listed are: the system complexity, shared multi-tenant environment, internet facing services and loss of control. Are also mentioned some security and privacy issues, like: governance, compliance, trust, architecture,

identity and access management, software isolation, data protection, availability and incident response. These issues are also discussed later on in [46]. Other concern with these technologies becoming more widespread, is the lack of standards between cloud providers. Since each cloud provider develops their own technologies, and in doing so provide different ways to achieve the same goals, it can be a difficult decision to switch from one provider to another. Reverting decisions like this can be costly and hard to develop, making vendor portability and lack of standardization one major and delicate issue, leading to what is usually called as ”vendor lock-in”.

3.3 Architectural Styles

Another important area that has contributed to the improvement of cloud computing, and in part to the improvement of any system related to the fields of software engineering, is the Software Architecture.

According to [25] software architecture can be described as the notion that the architecture of a system can represent its structure by making use of one or more views. Views are software architecture descriptions of certain top-level design decisions that address specific system concerns, such as, the composition of the system and its parts, the interactions between the system components and the their properties. They can be divided in: *Module Views*, where the focus is on the systems Data Model and the key elements of the software are mapped to modules and subsystems; *Components & Connectors*, where the focus are the components (usually units of runtime interaction and data stores), the connections between them and the flow of these interactions, that may represent complex forms of communication; and also *Allocation views*, that focus on the relationship between the software elements and the elements in one or more external environments in which the software is created and executed (deployment, install and implementation views).

With the advancements in architecture description languages and tools and the emergence of architectural standards over the years, it became easier to identify and apply techniques and patterns to the system, increasing that way, its Reliability, Scalability, Efficiency, Usability, Maintainability, Functionality, Portability while reducing its Complexity (it being one of the most important factors since the systems have become more and more complex).

Then, the creation of a system’s architecture is one the most important steps while developing software, since software architectures focus on structural choices that are costly to change once implemented and can provide a bridge between the code and system requirements. Some of their main characteristics are that they can lead to better understanding about the system for the stakeholders, enable the reusability of components and reduce the systems complexity by separating

points of view associated with the various stakeholder concerns. There are multiple architectural styles, like: Blackboard, Client-server, Event-driven, Pipes and filter, Layered and even specific styles for Big Data and Grid Computing, as presented in [57]. In this section we focus on three specific architectures styles: Monolithic, Service Oriented and Microservices.

3.3.1 Monolithic Architecture

In monolithic architectures all the functionality is packaged into one single application containing every component within the same environment making it hard for it's modules to be executed independently or reused [15]. With this type of architecture being tightly-coupled, applications developed under its guidance can have benefits such as: *being simple to develop and deploy*, since deploying a monolithic architecture is as simple running just one application, *and being simple to scale*, since to scale the application we can run multiple copies and redirect traffic with the help of a load balancer [56]. But the use of this architecture style can also have some drawbacks, such as mentioned in [54]:

- Development can be slow and hard due to the fact that these applications are complex and difficult to understand and modify.
- Continuous deployment is difficult, due to the need to rebuild and deploy the entire application when any change is made.
- Scaling its components can be difficult since it can only scale horizontally, not vertically, and its components aren't detachable from the main application.

Application with the Monolithic style are predominantly, Enterprise Applications where a lot of persistent and complex data is handled while following the required business logic. The Monolithic style architectures, commonly follow an approach of at least 3-tier layer, as seen in the 3.2:

- a **Presentation layer (or also called UI/GUI layer)**, where the system displays the information or services to be accessed and where the system users (clients and admins) can interact with those functionalities;
- a **Business logic layer**, where the Domain Model is represented and the business logic and rules are defined. In some cases, the business layer can be combined with the persistence layer into a single layer, especially when the persistence logic that handles tables, graphs or other types of information (e.g.: SQL or other query languages) is embedded within the business layer

components. In the cases where the business layer and persistence layer are separated, the system consists of a 4-tier layer architecture instead of only 3;

- and a **Database layer**, where the persisted data is organized into tables and the relationship between the data and those tables is created.

Martin Fowler, in [41], defends that developers should start by developing systems as Monoliths and explore the complexity before committing their applications to undefined and inconsistent boundaries, that may come with other architectural styles. If the complexity rises, then it will valuable to split the system into less complex services.

stating that we should start with a Monolith and explore the complexity of the system before diving in inconsistent and undefined boundaries. When the complexity rises, then the separation of the system into services will be valuable.

3.3.2 Service Oriented Architecture - SOA

Service Oriented Architecture (SOA) is an architectural style in which the various application components, that make up the system, provide services to other components via a communications protocol, typically over a network. Each service in an SOA embodies the code and data required to execute a complete, discrete business function and can be invoked with reduced or no knowledge of how the service is implemented underneath, reducing the dependencies between applications and creating a contrast between Service Oriented Architectures and Monolithic Architectures, by presenting loose coupled interfaces instead of tightly coupled. Other design principles are: service abstraction, reusability, autonomy, statelessness, discoverability and interoperability. SOA can also be seen as an approach for business-to-business intercommunication, as mentioned in [15], and according to [63] it tried to solve the complexity problem in Monolithic architectures by dividing the applications into groups of business applications offering services. This communication between different business applications is usually delegated to an Enterprise Service Bus (ESB) that performs integrations between the multiple applications. The communication between services can also be supported by other methods:

- **SOAP-Based Web Services** that focus on the communication between services when they're web services, having different styles for their different applications (e.g. RPC style and Document-literal style) [13];

- **REST**, or Representational State Transfer, which avoids the complexity and processing overhead of the Web services protocols by using bare http. Having four basic operations, POST, GET, PUT and DELETE, the operations in a resource URI would correspond to create, retrieve, update and delete (CRUD)

operations used commonly in information systems. It also provides a uniform interface to users that can be tested quickly and without the need of expensive tools. But when compared with other protocols, SOAP provides a more specific requirements with less flexibility but more standards that offer built-in compliances that can make it preferable in enterprise scenarios.

- **Messaging Solutions**, that focus primarily on exchanging asynchronous messages between between distributed applications in a sender-receiver or publish-subscribe way. The messaging system (e.g. Microsoft MSMQ, SonicMQ, Oracle AQ) provides configurable message queues, where applications can connect and send/recieve messages. These messages are coordinated by the messaging systems and sent asynchronously. These solutions are also often called Event-Driven Architectures (EDA). The main challenges of these types of systems are, the complexity that asynchronous messages introduce to the system, the performance cost to wrap the information in the message packets and the poor interoperability of systems, since some or are not available on all platforms. On the other hand, they offer great reliability, promote loose coupling and provide high scalability.

Finally, even though SOA is commonly implemented using Web services, services can use other implementation strategies [40]. In contrast with previous architectures styles, SOA grants some benefits such as: providing a scalable paradigm that is better equipped to develop systems that can evolve and be managed more easily; providing a foundation for agile and adaptable businesses with multiple development teams; and is designed to support very high workloads and a huge number of users; [40]. However it also has some drawbacks, as mentioned in [70] and [63], for-instance: the high initial cost; its time consuming and complex to implement correctly and without issues; and since it is an older concept, compared with Microservices, it was not designed for the size of the cloud, leading to possible bottlenecks in the ESBs and providing a single point of failure.

3.3.3 Microservices

Microservices architectures are a recent software architectural style, having gained popularity after J. Lewis and M. [22] described it in detail and some of its characteristics. It is defined as being an approach to the development of applications as a group of small services, where each service is deployed independently and the communication is done via lightweight mechanisms, often HTTP resource API. Usually it also has access to a dedicated memory persistence tool, such as databases. According to [20], ”A microservice is a cohesive, independent process interacting

via messages." and should be independent, by having the possibility to being deployed, updated and redeployed in isolation (tipically with the help of virtualization tools and/or containters technologies, following industry proven DevOps practices [11, 31]) and without compromising the application's ecosystem's integrity. The scalability, portability, updatability, and availability of the microservices are then some of its biggest benefits along with agility, resilience and maintainability [21]. It also presents some disadvantages, such as: the need to deal with distributed system and its increasing complexity; inter-service communication mechanisms must be implemented; managing a system comprised of many different services can become very complex; and increased memory consumption, since there are multiple isolated service instances it can have a high overhead, as mentioned in [56].

Microservices are one of the most recent and trending topics in the field of software development, contributing to a large number of fields, cloud computing and cloud architectures included. When compared with systems built 20 or 30 years ago, the complexity is still present but in a different scale and while monolithic systems are still useful for enterprise applications, microservices can provide services at a much bigger scale. The cases of success in making the changes and transitioning to a microservices architecture, are many small and big companies that have a focus on the potential for growth in the market they are engaged in. Companies such: Amazon [59], Netflix [42], LinkedIn [30] and many others.

3.4 Monolithic vs SOA vs Microservices

The relationship between monolithic architecture and service oriented architecture (SOA) is marked by a great contrast since a monolithic architecture is a concept where software is developed as a single unit and SOA focuses on developing software that has multiple services which interact with each other, making a more complex network of communication but having the benefit of being more loosely coupled, improving reusability of all those independent services, the maintainability, reliability and development. Some other cons of the SOA are that, due to its complexity is harder to deploy than a monolithic architecture, that is usually deployed as a single solid entity, and also it has a worse performance than monolithic based software, as in these later systems there are very minimal or sometimes no call for API's and modules are close to each other.

On the other hand, the relationship between SOA and microservices is a topic on which there is seldom consensus, making it hard to understand if microservices concepts and technologies do or do not constitute a new architectural style different from SOA, due to the similarities that these two concepts have. Zimmermann in [72]

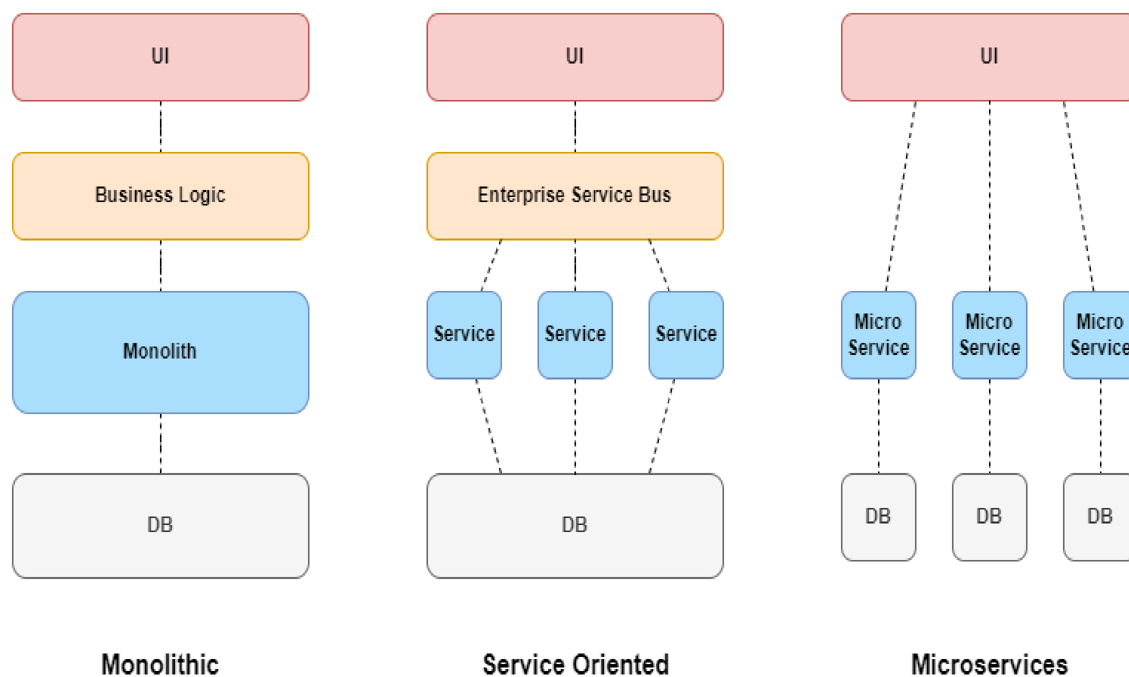


Figure 3.2: Comparison between Monolithic, Service Oriented and Microservices Architecture

reviewed this debate and concluded that microservices concepts and technologies are not a new architectural style "but rather qualify as SOA implemented and services realized in one particular way with state-of-the-art software engineering practices.".

This position is derived from a literature reviewed that makes evident the differences between them, one is a architectural style with a enterprise scope and the other a engineering process with a application scope.

3.5 Cloud Native Application

Cloud native is an approach to building and running applications that exploits the advantages of the cloud computing delivery model.

Cloud Native Applications (CNA) are then applications that take advantage of ideas like the culture of *DevOps*, where building, testing, and releasing software happens rapidly, frequently, and more consistently, *Microservices* and *API Driven applications*, *Service Discovery*, *Horizontal Scaling* and *Containerization or Virtualization technologies*. So these applications can be described as the three paradigms:

- **The organization of the components of the system**

  Components of CNAs are often realized as containers in CNAs. These self-contained deployment units are operated on platforms like Kubernetes or

Apache Mesos, and can encapsulate deployment unit heterogeneity. These platforms can reduce vendor lock-in effects by being deployed across different cloud service providers. For these heterogeneous components to be deployed, some requirements need to be met such as: the need to be *self-contained deployment units*, have *scalable and simple communication means* and must be *loosely coupled by events or by data.*

- **The properties of the components**

  The CNA design seems to be highly correlated and influenced deeply by pattern based approaches, like microservice architectures, gaining some of the same properties such as horizontal scalability, elasticity, resiliency, replaceability and upgradeability leading to a system with High Cohesion, Low Coupling, autonomous services and independent components.

- **And the relations between the components and the environment/user**

  As mentioned before, API Driven applications and Microservices are some of the main concepts that influence CNAs. So the communication and relations between the components of these types of applications tend to make use of the same mechanisms, like REST-based APIs, providing pragmatic means of communication relying mainly on the already existing internet infrastructure and well defined and widespread standards. The provided services shall be loosely coupled by events or by data, realized via messaging solutions (like AQMP standart) or Scalable storage solutions (like NoSQL databases).

  These mechanisms also have another advantage, since the communications are made via API calls, the users also have a simple interface to interact with.

In [37] is discussed the research implications of identified CNA principles, architectures, methods and properties identified in the items mentioned above.

3.6 Cloud and Mobile Computing

In [28] was studied in detail the fog computing architecture and its levels and presented a survey of the various computing paradigms and features. The paper also presents a discussion about various fog system algorithms and fog computing research challenges.

In [19] it is also discussed and presented the characteristics, challenges, the necessity for the use and development of fog computing as well as a architecture.

So, both papers discuss future and present challenges to a new cloud paradigm in development and its architecture.

In the study [9] various performance and metrics for fog, edge and cloud computing are discussed in an extensive way, highlighting particular issues to give an understanding of the future directions in performance evaluation of orchestration techniques in cloud computing and edge computing. Another important and already mentioned topics are: Edge computing and Mobile cloud computing.

In [36] the fundamental concepts of cloud and edge computing are studied. It is also classified and categorized the state-of-the-art in Edge computing, mentioning Cloudlets, Fog cloud and Mobile Edge computing, according to services such as real-time applications, security, resource management, and data analytics. Edge computing key requirements are also mentioned and some research challenges are also identified, as well as some limitations.

[5] examines recent mobile cloud computing architectures, compared the traditional cloud computing and mobile cloud computing models and also compared 26 surveys, since 2010, about mobile cloud computing. These comparisons were made based on the focus, components of the proposed architecture for mobile cloud computing, the contributions they make, the analysis techniques used to determine research challenges for mobile cloud computing and describing these challenges. Finally the researchers also proposed a generic architecture to assess recent mobile cloud architectures and identified research challenges needing investigations, including: security, privacy, bandwidth and data transfer, data management and synchronization, energy efficiency and heterogeneity. The state-of-the-art in cloud computing also focuses on other challenges, mainly related with security challenges and virtualization.

In [71] and [3], both present various state of the art technologies, discussing topics like Architectural Design of Data Centre, Distributed file system over clouds and Distributed application framework over clouds, and also focus on some research issues and challenges such as: Virtual Machine Migration, Information Security, Novel cloud architectures and others. Other surveys, like [58], present various security challenges and vulnerabilities, attacks and threats that hamper the adoption of cloud computing. The researchers also proposed a 3–tier security architecture, to help enhance cloud security, consisting of: an application level, a cloud-service middle level, that makes use of sniffing tools - Heluna - and a protocol and standard called Cloud Trust Protocol (CTP)) and an infrastructure level.

3.7 Microservices

Another technology closely related and that greatly benefited from Virtualization and Containerization is Microservices, where containers are considered the standardized way for microservice deployment.

So, [31] presents several microservices tools and categories, such as: Container engine, Service discovery, Monitoring, Container Orchestration, Fault tolerance, Continuous delivery and others. They also present technological and architectural perspectives and future challenges in the field. Finally some articles and their contributions are mentioned, for instance: In [55] is presented a methodology for designing microservice architectures based on domain-driven design (DDD) and model-driven development (MDD) and discussing the challenges and strategies to cope with them;

In [15] is reported the experience of migrating a mission-critical monolithic banking application to a microservice architecture, mentioning how it can improve scalability and also other benefits of migration to microservices, including simplified integration, higher cohesion, and lower coupling. [62] identified common bad practices in microservice development.

Another important topic regarding Microservices is the high availability that this architecture style can provide, but since they can be described as distributed systems, Microservices also are subjected to the CAP Theorem (Consistency, Availability and Partition Tolerance) presented in [26]. Dividing the network where the Microservices are deployed can be a useful, not only regarding security concerns, but also in improving the availability of the system.

To improve and optimize the Microservices, and thus enhancing the availability of the system, [47] provides a conceptual methodology to partition a microservice based on domain engineering technique and the domain-driven design pattern. To confirm this methodology, is demonstrated its use on the weather information dissemination domain as a confirmatory case study. Is then presented how the split in the weather information dissemination system sub-domain is achieved, by splitting it in different microservices of optimal size.

One pattern that also helps improve scalability and availability, and that became more affordable with the development of cloud computing solutions and scaling of database systems, is the Sharding pattern discussed in [18]. In the paper are discussed database distribution models: Replication, Sharding and Partitioning Strategies, focusing in a sharding technique known as hash partitioning with the objective of cataloging in the format of a Database Scalability Pattern the best practice, consisting in sharding the data among the nodes of a database cluster using the hash partitioning technique to nicely balance the load between the database servers. Another objective of this research is to help developers identifying when to adopt the pattern instead of other sharding techniques, since it efficiently provides read and

write scalability improving the performance of a database cluster but it does not solve all database scalability problems.

In a different implementation of partitioning microservices, [45] presents an architecture and implementation for automatic network slicing for microservices. The concept of “Network slicing” enables the possibility to provide an optimized logical infrastructure for each service and in this case is used to construct multiple isolated logical infrastructures, slices, on a single physical infrastructure where each slice accesses appropriate virtual network functions, a logical topology, isolated logical computational resources, and isolated logical network resources for the service to be provided. It is also discussed the issues encountered during the implementation and the knowledge gained can be used to create more general automatic network slicing for other service development methods, thus providing slices more quickly and cheaply in comparison with the current slices that can be expensive, since to construct a slice that offers the performance needed by a service, the service providers must design a slice optimized for their service and fully understand a service, what can be costly and inefficient.

3.8 Virtualization

While the use of virtualization technologies has increased dramatically in the past few years, virtualization is not specific to cloud computing or just one field of computer science. Virtualization can be described as *”a technology that combines or divides computing resources to present one or many operating environments using methodologies like hardware and software partitioning or aggregation, partial or complete machine simulation, emulation, time-sharing, and many others”* [17]. Then, a virtualization layer (also called Virtual Machine Monitor), can support the infrastructure by using lower-level resources to build multiple VMs that can be isolated and independent from each other.

Virtualization can exist at different levels, from the *Instruction Set Architecture Level* where is implemented by emulating an instruction set architecture completely in software; the *Hardware Abstraction Layer* which is the most commonly used on x86 popular platforms, making them more efficient, viable and practical; at the *OS level* the virtualization layer above the OS produces a partition per virtual machine that is a replica of the operating environment on the physical system, consisting of the OS, user-level libraries, a file system, system specific data structures, and other applications and environmental settings; to the *Programming Language level* where the Java Virtual Machine (JVM) was a pioneer, by creating a virtual machine at the application-level that could behave like a machine to a set of applications, like any other machine. Some examples are the JVM, Microsoft .NET CLI and Parrot;

In practical scenarios, there are countless useful reasons that can benefit from the use of virtualization technologies, such as: *Sandboxing* if there is a need to create secure and isolated environments for running untrusted applications; *Multiple execution environments* when there is a need to create multiple execution environments, improving a service Quality of Service; *Virtual hardware* and *Multiple simultaneous OS*, among others.

Regarding the virtualization of resources the study [67] discusses different virtualization technologies, identifying the most relevant as being: Hardware virtualization, or Hypervisors, and System-level virtualization, or Containers. Some existing solutions are also presented in the form of the most known container-based systems, such as: OpenVZ, LXC (LinuX Containers) and Linux-VServer. It finished by giving an overview about Docker, a container-based orchestrator, and stating challenges and issues.

In [53] is concluded that research on container technologies is still in a formative stage and needs more experimental evaluation, since most of the reviewed literature is still a mix of technology reviews, solutions and use case architectures and not all of them are implementations with concrete data and conclusions. Is also highlighted that there is a imbalance of contributions with a higher number of use cases than technology solutions. Some benefits of containerisation and cluster-based orchestration are also mentioned, like: adjustable cluster sizes for the deployment of containers, easier cluster maintenance, quicker deployment, development, testing and monitoring, and when combined with the interoperability of successful container technologies, allows the management of highly distributed topologies of smaller virtualised devices beyond centralised clouds. Finally, is emphasized that future research should focus on methodological and tool support, like Containerization technologies and Container Orchestration Tools, instead of only focusing on virtualization performance and isolation.

3.9 Container Orchestration Tools

As mentioned in the previous section, the development of Container Orchestration Tools can have an enormous impact on the field of cloud computing. A huge challenge cloud providers face is to efficiently provide services with the growth of their offerings. To deliver on their services and make the best use of their systems, cloud providers resorted to virtualizations techniques, as mentioned in the section above.

Initially, VMs were the main technology for this purpose, but with the introduction of Containers it became a not so trivial choice to make. Although VMs and containers are both virtualization techniques, they focus on solving different problems. **Containers** have a focus on the PaaS model, by delivering portable soft-

ware while aiming for high interoperability and still making use of OS virtualization principles. On the other hand, **VMs** have their focus on hardware management, allocation and virtualization, having more properties in common with the IaaS model.

In general, containers are more efficient at holding self-contained, packaged, ready-to-deploy parts of applications and, if necessary, middleware and business logic (in binaries and libraries) to run applications. They usually have:

- A smaller overhead, when compared with VMs, by making use of a lightweight portable runtime;
- A capability to develop, test, and deploy applications to a large number of machines/servers;
- And also have the ability to interconnect independent containers;

making them faster and more suited to deploy portable, interoperable applications that need a lightweight distribution of packaged applications.

Another great feature of container technology is its flexibility, a container can be started in a few seconds, allowing containers to be started and stopped as needed, (scaling up, at a time of peak demand and scaling down when not needed). In addition, if a container crashes, it can be restarted quickly so it can get back to the task and can easily support the rapid deployment strategies of the DevOps philosophy. For these reasons, it quickly became associated with Microservices and their development and deployment.

With the separation of larger applications into small services/microservices, the complexity of the system shifted from its structure and implementation to the communication between its services and their orchestration. Also, with so many software running as containers, managing these containers has now become a requirement, being arduous to borderline impossible to perform the management and orchestration manually. To attend to this need, specialized software and container orchestrations tools were created, being the most commonly used: **Kubernetes**, which is an open-source, out-of-the-box container orchestration tool, with scheduler and resource manager for deploying highly available containers more efficiently. Its architecture is presented in the next section; **Openshift**, which is built on top of Kubernetes; **Hasicorp Nomad**, is an orchestration platform from Hashicorp that supports containers and shares a similar philosophy of Kubernetes in managing applications at scale; **Docker Swarm**, that consists of a tool for the management of a cluster of Docker containers, but is still maturing in terms of functionalities compared with other tools presented on this list; **Apache Mesos**, is another cluster management tool, following the same model of master/slave as Kubernetes; And other cloud-based managed container orchestration tools, such as Google Container

Engine (GKE), AWS Elastic Kubernetes Service (EKS) and AWS EC2 Container Service (ECS).

In [32] is studied the performance and made a comparison between four of the most used Container Orchestration Tools/Engines: Kubernetes, Docker Swarm, Apache Mesos and Catlle. The study focused on four metrics to compare the engines (*cluster provisioning time, Provisioning time of applications with different complexity with local image and Docker registry, Provisioning time of a web application with a high number of replicas using local images and Docker registry* and *Failover Time*). Its concluded that Kubernetes has a better performance in the deployment of applications with greater complexity while the others engines are superior and more efficient tools for simpler deployments.

3.9.1 Kubernetes

As mentioned in the last section, Kubernetes is a a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. Kubernetes provides several management services, including: *Service discovery and load balancing*, *Storage orchestration*, *Automated rollouts and rollbacks*, *Automatic bin packing*, *Self-healing* and *Secret and configuration management*, but to fully understand it and its architecture (presented in the figure 3.3) is necessary to have a basic knowledge about its core components.

- **Pod**: Pods are the smallest deployable units of computing that can be created and managed in Kubernetes. It can represent a group of one or more containers with shared storage and network resources, and a specification for how to run the containers;

- **Deployment**: A Deployment is an object that allows the management of a set of Pods, by providing them declarative updates. A desired state can be defined and the Deployment Manager will enforce it over the set of managed Pods, by increasing or decreasing the number of replicas, rolling out updates or restarting the Pod in case of errors or crashes;

- **Service**: A Service is an abstract way to expose an application running on a set of Pods as a network service and also a policy by which to access those Pods. It makes sure that the network traffic can be directed to the current set of Pods for the workload, by providing an entrypoint that is usually defined by ports and protocols;

- **Volume**: At its core, a Volume is an abstraction for a directory system which is accessible to the containers in a Pod, aiming to solve problems like data

persistence and sharing files between containers running in a Pod;

- **PersistentVolume and PersistentVolumeClaim**: Unlike regular Volumes, which are transient in nature, PVs are persistent, supporting stateful application use cases. A PV is a resource object in a Kubernetes cluster which continues to exist even after the pods using it have been destroyed. PVs must be requested through PersistentVolumeClaims (PVCs), which are requests for storage by mounting a PV meeting certain requirements on a pod. This allows developers to dynamically request storage resources, without being aware of the implementation of underlying storage devices.
- **ConfigMap**: A ConfigMap is an object used to store non-confidential data in key-value pairs, such as: environment variables and command-line arguments, allowing the decouple of environment configuration from the container images, making the application easily portable.
- **Secret**: A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Secrets are in this way, similar to ConfigMaps, with the main purpose to hold confidential data and eliminating the need to include confidential data in your application code.

All these objects can be run/created in a Kubernetes cluster and are part of the Kubernetes environment, they can be queried and have their states managed by resorting to the K8s API, by either directly using arguments on the command line or using yaml configuration files.

One of the key challenges about Kubernetes is the managing of StatefulSet (in situations of failure, recreation and restart). In [1] the focus was to investigate the current support for stateful microservices, in Kubernetes, and identify the related problems. To combat this challenge, its proposed a solution to enhance the Kubernetes tool with a State Controller, allowing for state replication and resulting in an improvement of 55% (and even up to 99% in some specific cases) regarding the recovery time of a stateful microservice.

3.10 Security

Security is a topic that can never be left out of any system, architecture or device due to its importance in keeping information private and secure. Even more, it has become increasingly more important due to the explosive growth of mobile computing, the use of mobile devices and their performance.

In [23], the paper focuses on mobile devices and data protection, presenting types of weaknesses and threats that are usually identified on this type of device. The

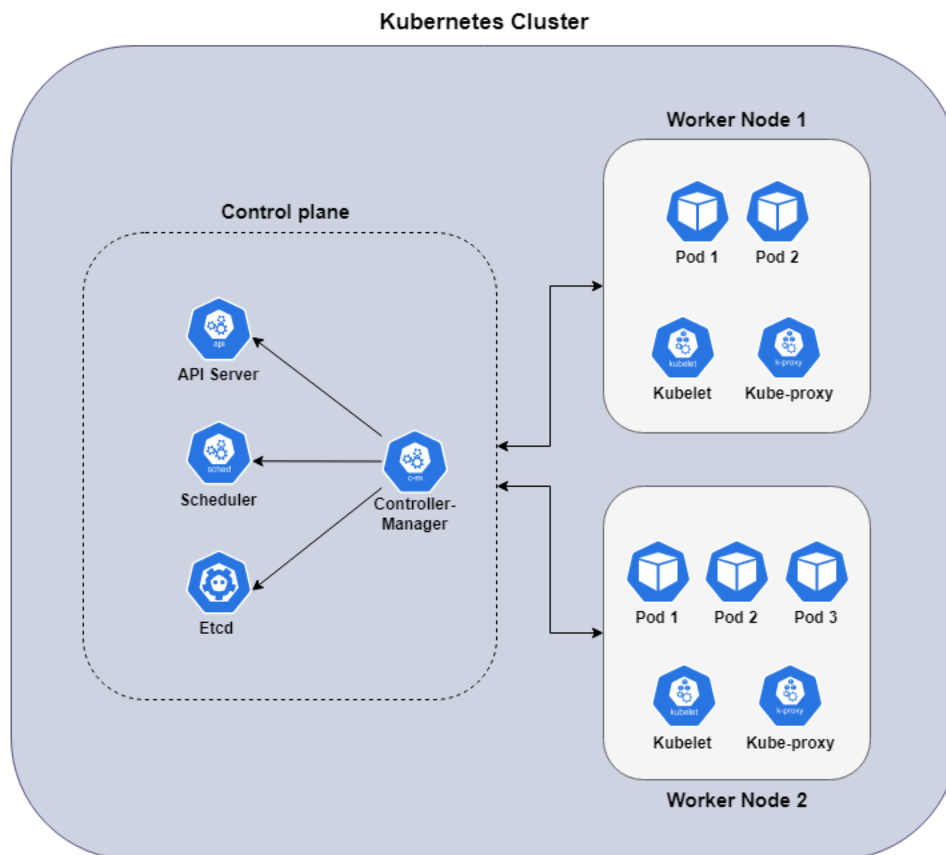


Figure 3.3: Kubernetes Architecture Example

threats are presented in categories, such as: *Malware*, *Phishing and social engineering*, *Direct attack by hackers*, *Data communications interception and spoofing*, *Loss and theft of devices*, *Malicious insider actions* and *User policy violations.* It is also presented several defences mechanisms that can be implemented to counter these threats, from the use of personal firewalls, or programs that help detect and protect from virus or spyware, to encryption, VPNs, Backup and recovery methods and also education and training. Finally are presented some discussion on these defences and concluded that fortunately the needed security technologies are available and the main challenge is assessing priorities and correctly selecting the most important measures to defend against the specific attacks that the devices are more vulnerable.

3.10.1 Data Privacy

In [43] the security of user data in mobile devices based on a secure cloud approach is presented and a solution is provided by encrypting the necessary information locally, with the AES 256 algorithm, and storing the encrypted information in the cloud. This approach has several advantages but some weaker parts of the process where the communication is concerned. The study also discusses several joint efforts of mobile device manufacturers and operating system developers that have incorporated encryptions on the devices to be utilized by the users [29], while presenting different mobile manufacturers solutions, including: Safe for Huawei p9 and Later Models, Samsung Android Mobile Device Security, Google cloud storage, Microsoft OneDrive, Signal, CoverMe, Secret Box and Crypt4All.

Other ways of securing data in mobile devices, can be methods that encrypt, overwrite or flush data in specific cases where the unauthorized user tries to access or modify it. In [48] transient authentication in mobile devices, more specifically wearable devices and proximity-based hardware tokens, is explored and the possibilities of encrypting, overwriting, and/or flushing data are mentioned as precaution mechanisms when the user departs from its device, granting the security and confidentiality of the stored information.

Finally, in [39] is discussed what data is, different encryption and decryption techniques, Data compression and various compression techniques to securely transmit information without loss.

3.10.2 Authentication

In [4] firstly, conventional authentication approaches are reviewed and explored, like: Secret knowledge-based approach (PIN, passwords and passphrases, cognitive knowledge question, patterns and graphical passwords), Token-based approach, Biometrics, Multi-layer and -factor authentication. It also examines the current use of

authentication technologies offered by service providers and device manufacturers (HSBC, NatWest, Lloyds, SAMBA, Windows 8.1 Laptop/PC, Android (Samsung Galaxy S5 and above), iPhone 5S and above and Google Authenticator). Some featured authentication frameworks, namely single sign-on and federated identity, are discussed in terms of the benefits they offer as well as their shortcomings and finally are considered some Continuous and transparent authentication systems. The last survey mentioned in this section [6] focus on authentication on mobile cloud computing, beginning by exploring the differences between authentication requirements and principles in MCC and cloud computing, impacting in positive and negative ways, such as: resource limitations, mobile device sensors, mobility, network heterogeneity and types of communication (wired or wireless). Then the state-of-the-art of authentication approaches in MCC is presented, being divided in Cloud-side (Multi-factors, MDA and Cloud-based Biometric) and User-side (FDZ, QR code-based, ScDiCi, TrustCube, SMCBA, NameAuth).

Chapter 4

Architecture for crew vessel verification

In this chapter we will introduce the mobile application and the supporting backend developed, based on cloud technologies, to improve the verification and validation process of documents regarding vessels and vessel crews, whether offshore or near the coast. We will focus on the overall architecture and the needed requirements, the frontend application (a mobile app developed using Flutter), the backend systems (Microservices and Databases), and some of the challenges encountered during the development of the system.

As we stated before, cloud technologies and cloud computing have recently gain popularity due to a variety of innovations in the field that allowed them to grow and start to compete with more standard, tested and reliable technologies and methodologies. The evolution of these technologies and related fields have allowed a new, and different, perspective on developing applications and systems and try to tackle some of the shortcomings of other approaches, for example monolithic approaches. Applications built with the support of these new technologies, methodologies and architectures focus on improving the scalability of the systems and increasing the adaptability and compatibility with different types of devices, operating systems and data storage types, thus improving performance and flexibility.

Still, they are not a perfect solution that can solve all problems and be mindlessly implemented in every situation, having some challenges and hurdles to overcome, such as security, privacy and trust concerns, bandwidth and data transfer, management, synchronization and heterogeneity limitations, as discussed in [46], [52]. There are also other challenges that can arise from external sources to the systems, such as: weather conditions and connectivity to the network.

Currently some of these challenges are experienced by the Direção Geral de Recursos Marítimos and Portuguese maritime authorities in the validation and verification process of vessels and their crews. The DGRM's mission is to develop safety

and maritime services, including the maritime-port sector, the implementation of policies on fishing, aquaculture, manufacturing and related activities, the preservation and knowledge of marine resources, as well as to ensure the regulation and control of activities developed in these areas. It is also responsible for carrying out vessels and seafarers inspections, regarding their safety, security, certifications and licenses [1].

Regarding the inspectors and the entities carrying out this type of vessel inspection missions, they can be undertaken by a wide range of entities, such as: the DGRM itself, the Portuguese Navy, the Portuguese Air Force, the Republican National Guard, the Autonomous Region of the Azores (Inspeção Regional das Pescas) and the Autonomous Region of Madeira (DRP/Inspeção Regional de Pescas da Madeira). These inspections can also be carried out jointly or autonomously (jointly meaning that different entities can collaborate during the inspection; autonomously meaning that only a single entity is responsible for the inspection) and in an unannounced or scheduled manner.

To improve the current process of inspections, of vessels and their crews, and the system that supports those missions, a new system was requested to tackle two main problems with the current architecture: *The lack of data aggregation about inspections and all its components* (data about inspectors, the vessel utilized by the inspector, crew members and the inspected vessel) and *The high cost of the satellite communications*, used in the communications related to the verification and validation of the documents and information obtained in the inspections. This type of communication is widely used in systems associated with navigation of ships and air crafts, due to its nature it can provide a large coverage over many geographical areas, including near and far away from the coastal line, but as they are too expensive, they end up being an unattractive option that is mainly used in emergency situations.

Thus, the development of the new architecture, based on microservices, contrasts with the current architecture, based on a monolithic approach. The new architecture focused on mobile application, as a frontend, and a backend cloud structure that provides the system with better scalability, high availability, and that is based on Microservices, allowing access to the necessary data through different APIs. So the main goals of this architecture are:

- **The development of a mobile application (frontend);**
  - This mobile application should support the inspectors during the vessels inspections by:

[1] `https://www.dgrm.mm.gov.pt`

1. Collecting and saving information, in a secure way, about the vessel, the crew and the inspectors in charge of the inspections. This information can be collected in the beginning of the inspection and cross checked with the information obtained during the inspection or just collected during the inspection and checked on a later stage;

2. Supporting the validation and verification of, physical or digital, identification documents that can be carried out on board of vessels or on the coast;

3. Maintaining/performing user authentication, of the inspectors, on the mobile device. Since the vessel, to be inspected, may be in areas far from the coast, where access to the network may be limited or non-existent, it is expected that it will not be possible to have real-time access to the authentication and document validation/verification platform. Therefore two different types of authentication are considered:

   * **Online Authentication**: Where the user can communicate with authentication services, in real time, and can verify and validate his credentials. This type of access is usually made through land terminals or terminals that are within 20/40 miles of the coast.

   * **Offline Authentication**: Where the user does not have immediate access to the authentication services. Therefore, it will be necessary to synchronize online and offline authentication, forcing the user to perform online authentication initially and then use those credentials to perform offline authentication. These credentials can have an associated validity period, preventing possible attacks where they are reused.

- **The development and deployment of a scalable cloud backend architecture;**

  - The developed backend should support the inspections and the mobile application by providing them with the necessary data to carry out their missions. The backend is then split in two parts:

    Microservices: Each of the different microservices will be accessible via an API, providing the users with information regarding the vessels, the

vessels licenses, the crew members, and other data needed to support the inspections.

Databases: Each microservice will be supported by a distinct database, to improve security and data isolation.

4.1 User Stories

To achieve the goals mentioned in the previous section, to structure the development of the overall system and to articulate how a piece of work will deliver a particular value back to the customer/end user, in this section are presented some of the main User Stories who were defined for the scope of this project.

Initially, there were defined some main Use Cases, such as: *Device authentication (online/offline), Document validation/verification on board of vessel, Synchronized sharing data between mobile devices* and *Obtain data from the crew members and vessel*, but since the system being presented in the next chapter is based on microservices, it's development should adhere to agile development methodologies. With that goal in mind, the main Use Cases were converted into the User Stories presented below.

As an Inspector, I want to:

- **Perform authentication on a device, in order to use the application and its functionalities;**

  The authentication on the mobile devices must be first done in a online mode, so it can get access to the authentication platform and retrieve the necessary information that enables the offline authentication mechanisms. After the initial online login the user is allowed to authenticate locally (offline) during seven days (this time period was provided to us by the DGRM team as being the maximum time an inspection can last). After that amount of time, the authentication credentials are wiped and the user is required to re-authenticate online.

- **Initiate the inspection process in order to obtain the necessary data for the process;**

  In the beginning of the inspection process, the inspectors need to choose the vessel(s) to be inspected and request all the necessary information related with the vessel(s) to be inspected, license(s), crew(s) and vessel used by the inspectors.

- **Access the inspection report, in order to have access to its information;**

The inspector, once the inspection has started, should have offline and online access to the information requested in the beginning of inspection.

- **Add additional information acquired during the inspection, in order to complement the inspection report;**

  Additional information can be acquired and stored during the inspection regarding the vessel, crew members, inspectors and also photos and videos of the vessel.

- **Edit information gathered in the report, in order to build a report with correct information;**

  The information gathered during the inspection and the initial information should can suffer alterations, such as being outdated, and should be able to be edited during the inspections.

- **Validate physical documents in order to verify their authenticity;**

  This validation should be able to be done online or offline. Since the DGRM uses QRCodes to scan physical documents, it can be used to verify the validity of the phishing licenses related with the vessel.

- **See an overview of all document validations performed during the inspection, in order to simplify the inspection process;**

  In order to simplify the validation and verification process, the inspectors should have access to a scan history where are displayed all the valid and valid scans made during the inspection.

- **Share/Receive information obtained during the inspection with other inspectors in order to standardize reports and increase Redundancy and Data Integrity;**

  Inspectors should be able to share the information, gathered individually during the inspection, and share it with other inspectors of the inspection team. This will allow each inspector to obtain information and have it stored and confirmed by other inspectors, promoting the redundancy of data and its integrity.

4.2 Requirements

Regarding the capabilities of the system, according with the necessities and prerequisites presented by the DGRM, were gathered some functional and non-functional

requirements that proved to be useful in the early stages of the development of the cloud architecture and the overall system.

Concerning the Functional Requirements, they are presented in the next section 4.2.1, and specify what the system must do in response to the different interactions with the users.

The Non-Functional Requirements, are introduced in the section 4.2.2 and were divided in two distinct groups depending on if they were backend or frontend requirements.

4.2.1 Functional Requirements

Functional Requirements	
Number	Requirements
FR1	Enable the retrieval of data from crew members and vessel
FR2	Perform the validation of data and documents on board the vessels
FR3	Enable the update of data from the vessel crew members
FR4	Allow the synchronization and sharing of data/information between mobile devices during vessel inspection operations
FR5	Register occurrences for possible administrative offense proceedings

Table 4.1: System functional requirements

4.2.2 Non-functional Requirements

Non-functional Requirements	
Backend	
Requirement	Definition
Scalability	The solution will serve more than 5000 vessels (or more than 8000, including the autonomous regions of the Açores and Madeira)
Availability	The solution shall implement load-balancing, failover mechanisms and make use of multiple replicas and orchestration systems (like Kubernetes)
Interoperability	The solution should make use of open standards and resort to containerization and virtualization technologies (like Docker and VMs)
Platform dependency and Portability	The solution must have a low dependency on the platform on which it is being deployed, making it easier for being deployed in different environments
Security	The solution should use strong and proven authentication methods, multiple authorization levels, secure communications, input validation mechanisms API request limitations and others
Usability	The solution must offer scripts that allow the user to deploy it in a fast and simple way
Data access	The accesses/requests to data must be done through an REST API
Frontend	
Requirement	Definition
Redundancy and Data Integrity	The solution shall provide a mechanism for the different mobile devices, used by the inspectors, to communicate locally and share the collected information
Security	The local communication of the devices must be secured using technologies that allow data encryption and avoid well-known and well-studied attacks, like Man-in-the-middle attacks and others
Application	The solution must be accessible via a mobile application and must maintain authentication on the device during the time of the inspections (inspections can last from 24 hours to several days)
Activity Log	The solution should be abble to regularly record/log the actions and activities of inspectors in a log file

Table 4.2: System non-functional requirements

4.3 New Vessel Inspection Mission

The vessel inspection mission, where is made the verification and validations of documents, is one of the most important aspects of this project since it is where the most challenges and difficulties arise, compared with other stages of the process of inspection. So, in the figure 4.1 is presented a flowchart that encompasses the new mission flow, making use of the developed architecture and mobile application, and the main interactions between the intervenes, being them: the *Inspection Subject*, the *Inspector*, the *Mobile device* used by the Inspectors and the *Database*.

The process begins in the mainland, where the Inspectors have stable and real-time access to the network, and therefore to the system. The inspectors start by initiating the inspection, requesting to the backend, via the microservices APIs, the necessary information about the target (information about the vessel to be inspected, the licenses and the crew) and requesting a vessel that will be used by the inspectors themselves. This data is then stored temporarily in the mobile device of the inspector and can be accessed and used later.

The next step of the inspection occurs when the inspectors are already on board of the vessel. The inspectors travel to the vessel and begin the inspection of the crew, captain, and the vessel itself. During this inspection mission is when the inspectors will carry out the process of verification and validation of the data provided by the subjects of the inspection, and the process can be carried out in two distinct ways, online or offline, depending on the distance the vessel is from the shore and whether it is connected to the network / able to perform the verification in real time in the system. The online process connects to the available services and verifies the obtained data with the current data in the database, while the offline process only verifies the obtained data with the stored information in the mobile device.

In the third stage, if the obtained data is invalid it will be presented a warning to the inspectors. If the obtained data is valid the inspection will continue.

Finally, when all the information is gathered, including all the information of the different inspectors, a report is created and sent to backend systems. Later, this report will be sent to the inspection subjects to be approved, or amended. This stage is necessary to ensure that all the obtained data is correct and to give the opportunity to the inspection subjects to correct it if its not accurate. This process its not immediate and can take several days to conclude, since the information should be sent via email (or by other available means) and the inspections subjects can add comments to the report if they believe something is wrong.

After the review by all the participants, the final report is created and the inspection process is concluded.

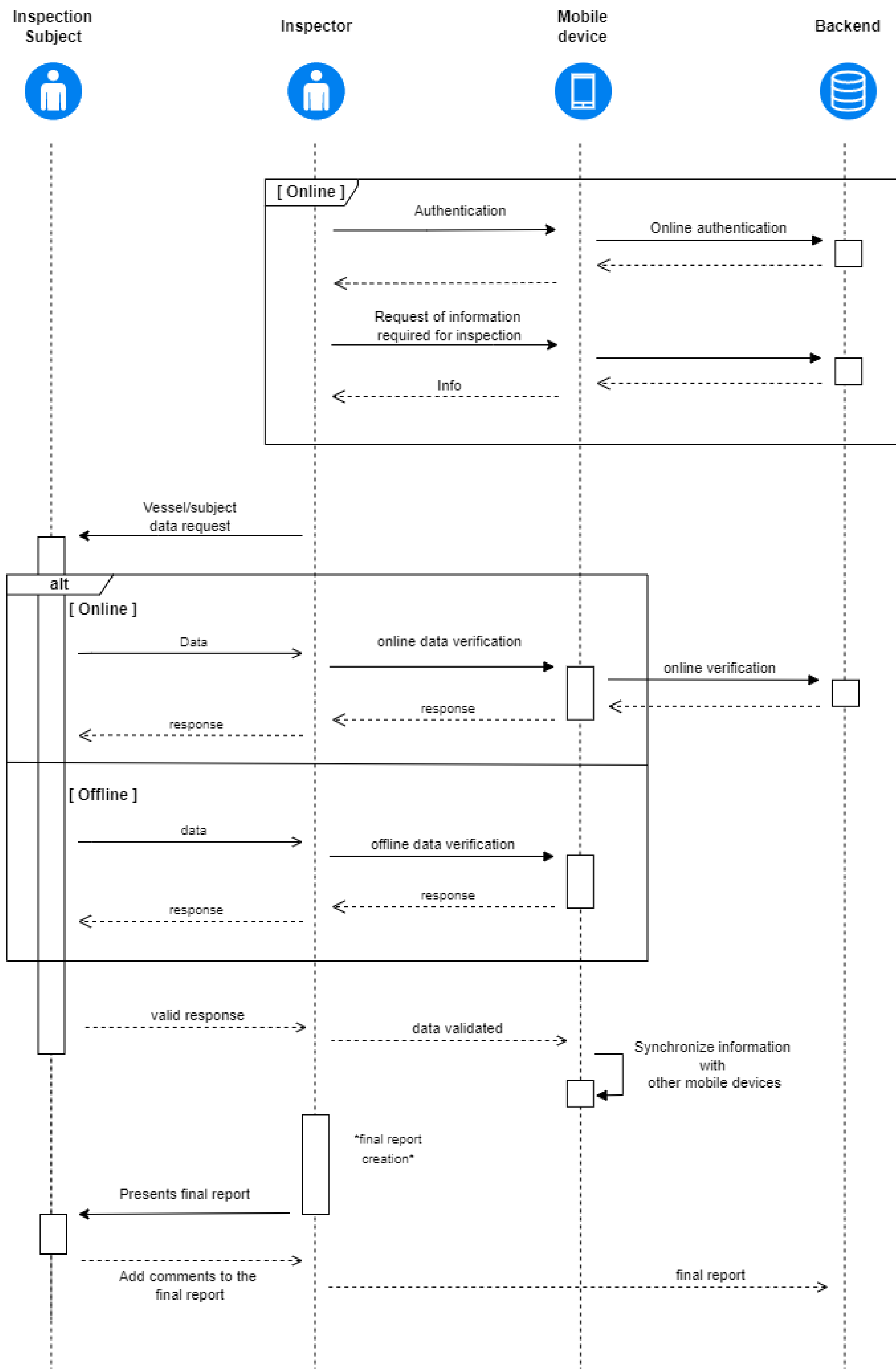


Figure 4.1: Vessel inspection mission flow

4.4 Architecture

The developed cloud architecture is presented in figure 4.2 at the conceptual level, divided into several components and layers. In this overview the architecture can be divided in two distinct sections: the Frontend and Backend that will be discussed later in this section.

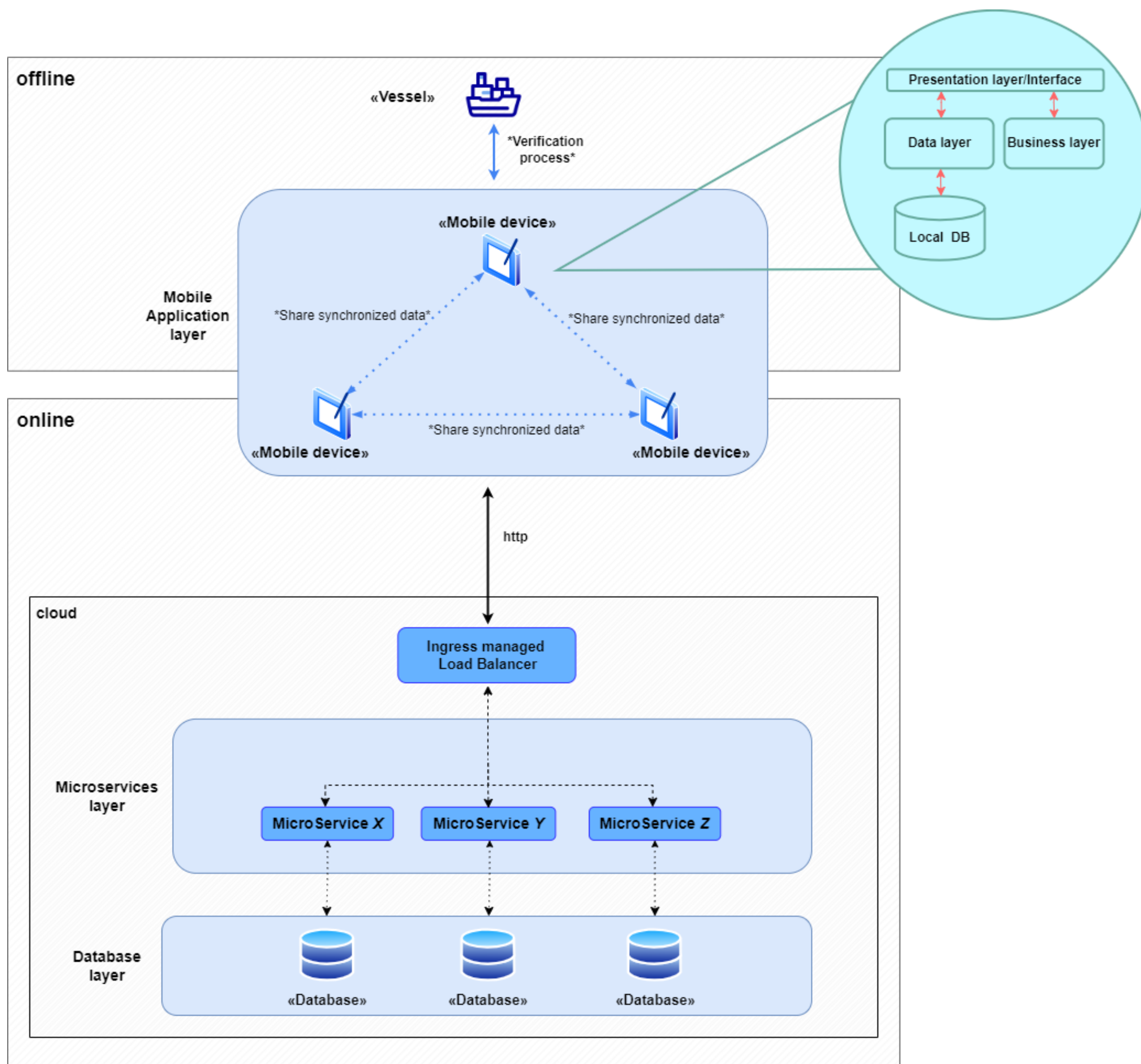


Figure 4.2: General architecture

The architecture was developed with the goal of providing a different, more up-to-date paradigm for DGRM's systems. The existing monolithic architecture, although reliable and meeting the proposed requirements at the beginning of its development, does not meet the current requirements that have been updated over

time.

In the last few years, DGRM has been trying to modernize its infrastructure, software and processes, now focusing on simplifying its backend architecture to enable a more practical, continuous and manageable development

Although the current monolith is divided into layers, this brings advantages such as the separation of concerns, which means that each layer can focus solely on its role, making it Maintainable, Testable, easy to asign separate roles, update and enhance separate layers. These are good qualities, but when the scope of the project gets too large, the maintenance of the code and the development of new services or modules start to depend more and more on each other, increasing coupling.

So, starting with the creation of the new generation of digital certificates and licenses, the DGRM wants to focus more on usability and simplicity of access to information, reliability and redundancy of data, availability of its services, granting business continuity, periodic updates and development of new services/capabilities and also the scalability of their systems.

Following the requirements raised for the new architecture the solution must be easily deployable, contributing to this the use of containerization and virtualization technologies. These technologies also contribute to a low platform dependency and facilitate the portability of the system. They also contribute to make it easier to scale services horizontally, improving the scalability and reliability of the system.

These requirements leave us with two main architectural styles, microservices and event-driven.

Event driven is a good approach since it provides good scalability (by having low coupling of services), point in time recovery if the events are backed by a queue or maintaining some kind of history, it is possible to replay events, or even go back in time and recover state and also provides a low degree of service dependency leading to asynchronicity. This however, can have its downsides, since inconsistencies can occur more often and is not typical for this type of architecture to support ACID (Atomic, Consistency, Isolation, Durability) transactions, so handling of duplications, or out of sequence events can make service code more complicated, and harder to test and debug all situations.

Microservices also has its downsides, associated with its complexity, difficulty to manage large numbers of microservices, network latency and load balancing. But, with self-contained and independent manageable microservices, the low cost and dynamic scaling (compared with monolithic architectures), high availability, easy testing and a faster release and development cycle, it proved to be the best suited approach.

Regarding the side of the architecture that focuses on the mobile application and mobile devices, it is a simple Client side that provides data redundancy thus

improving the reliability and security of the data.

Now focusing on the more specific factors of the developed architecture, the Frontend side of the architecture, represented as the **Mobile Application layer**, was developed as a mobile application, running on mobile devices used by the inspectors, that help organize and access necessary information. It is also represented the main three interacting elements during the inspection process, *the inspectors* that carry out inspection missions, *the vessel being inspected* that is the target of the inspectors and *the mobile application/mobile devices*.

Regarding the *Mobile application* developed, it can be broken down in four main different layers, each one handling distinct aspects of the application's operation: *Presentation layer/Interface layer*, *Data layer*, *Business layer* and *Local Database*.

- The **Presentation layer** presents the app content and triggers events that modify the application state, changing information, views and screens present in the application.

- The **Data layer** handles all the persistence of data in the device, communicating with the local database that is present in the mobile device and saving the necessary objects and data, via get, insert, delete and update functions. This layer is also in charge of handling all the network connections and communication with the backend, via API requests, and other devices, sending and receiving data.

- The **Business layer** focus on handling the business logic of the application, from handling the creation of inspections and controlling the different aspects of gathering data to the scanning of identification documents. It also aggregates all the different information that can be obtained in these types of inspections, from information about the vessels, crew members, licenses, inspectors and also the videos and images recorded during the inspection.

- Lastly, the **Local Database layer** that consists of a database, hosted locally in the mobile device, giving the ability to store data whether in online or offline areas and keeps the data stored even if the app is closed or the mobile device runs out of power or stops working for other reasons. This local database is based on SQLite, adapted for the a mobile environment, organizing data into tables and maintaining relationships between them.

These layers are examined and presented in more detail in the section 4.4.1.

The Backend side of the architecture, is represented as the grouping of the **Load Balancer**, the **Microservices layer** and the **Database layer** running on the cloud environment and infrastructure of the DGRM. The backend focuses on the

microservices architecture, where each service is deployed in a Kubernetes cluster, from a docker image containing the application code, libraries, tools and dependencies needed to make an application run.

- The **Load Balancer layer**, managed by an Ingress Controller that exposes the system to the outside by ”sitting” in front of the microservices and the databases, redirecting the http requests and traffic, from the frontend mobile application to the desired microservice.

- The **Microservices layer** consists of all the developed and necessary microservices to support the mobile application. These microservices functionalities are exposed via APIs, granting the mobile application (or other applications/services) the DGRMs information about vessels, crew members, inspectors, licenses and more.

- The **Database layer** was developed to simulate a database structure, similar to the one currently deployed in the DGRM infrastructure, but with some key differences to better support the microservices architecture and improve security. Then, the new database structure is divided in various, and separate, smaller databases that provide information to a single service.

The figure presenting the architecture, 4.2, also provides a distinction between the states of connectivity, Offline and Online. The *Online zone* represents the locations of the coastal region where the users can communicate with the DGRM services, in real time, and can verify and validate the identification documents and licenses. This type of access is usually made through land terminals or mobile terminals that are within 20/40 miles of the coast. The *Offline zone* represents the zones where the users don’t have immediate access to the DGRM services and where is necessary to handle information and business logic locally.

4.4.1 Frontend

As mentioned in a previous section, the main focus of the development of the mobile application is to assist and improve the process of inspections of vessels in Portuguese coastal areas by automating some parts of these inspections, that currently are mainly done manually (in the cases where the inspection is done far away from the coast and the internet access is weak or non-existent) and with support from expensive means of communication, satellite communication. So, the focal aspects of the application is the collecting and storing all inspection-related information, by aggregating and saving it in the backend systems (or temporarily in the mobile device), and also the verification and validation of the vessel identification documents, on board of these vessels, by scanning the documents and checking if they are up to date and valid.

There are also other important aspects, such as the authentication of the inspectors and the collection and storing of relevant information in cases of occurrence of misdemeanors, for possible future legal action (the development of the architecture didn't focus on this last aspect since it involves access to confidential information and knowledge of the juridic system, being out of the scope of this project).

Mobile Application

The figure 4.3 gives an overview about the different layers of the application (a more detailed decomposition view is presented in the Appendix A, A.1), where we can have a general view about the three main layers: *Presentation*, *Business* and *Data layer* (the Database layer is not presented in this Decomposition view of the system since it is managed by a external plugin, not developed by us, and represents a local dependency).

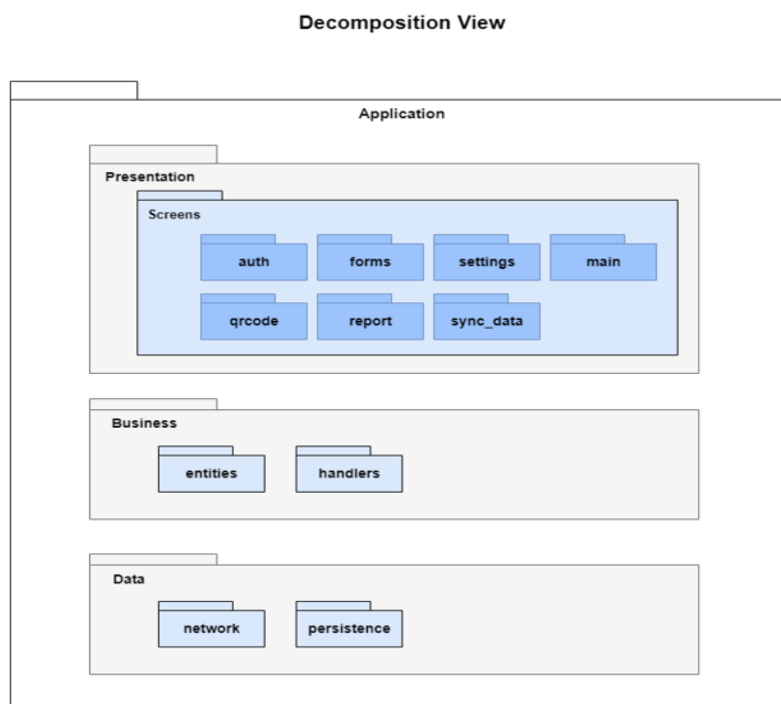


Figure 4.3: Frontend Decomposition View

- **Presentation layer**, in contrast to the other layers that focus on data and business logic, focuses on user experience and the visual appeal of the information being displayed to the user. In other words, it translates the business logic to the user and simplifies its understanding by hiding the system complexity, giving only the necessary information to the user and guiding him through the

various screens and views of the application in a logical, concise and effective order.

This is the layer with the higher number of modules, since the screens are mainly grouped by cases and their relation/order that they are viewed. The main modules are the **Auth module**, that handles the screens relating to the login, the **Qrcode module**, that handles the screens of the qrcode scanner and the list of scans, the **Sync_data module**, the focus on the screens where inspectors synchronize information, the **Forms module**, that handles all the forms and input of additional information and finally the **Report module**, handling the main page, the different views of data about vessels, inspectors, crew members and the inspection, and also providing access to the screens where you begin and end the inspections.

- **Data layer** is a layer that has its focus on the information and data acquired during the inspection. Its one of the most important parts of the frontend application since it handles all the information and its flow. In controlling the inspection data there are two main cases we can address, *the communication between the mobile device and the backend*, or other devices, by requesting/sending information via APIs and the retrieval/persistence of information in the database, local to the mobile device. The modules presented in this layer, reflect this, therefore we have the **Network module** that handles all the external communication and sharing/requesting of data, mainly with the developed backend system and architecure (via the microservices APIs) but also with other nearby devices (via Bluetooth).

  The next module, that also focus on data, is the **Persistence module**. This module differs from the Network module since if has its focus on the persistence of data, instead of focusing on the information sharing. It handles all the flow of information from/to the mobile device local database and also the creation of the tables and schemas vital for the inner workings of the application logic, by grouping the get/insert/delete/update methods for the different entities, like the inspectors, vessels, scans, licenses, etc.

- **Business layer** is the layer that handles the business logic of the application, making the bridge between the views and presentation of the application and the Data layer, where the data is stored in the device or send/requested via the network module. Its also divided in two modules. The first module is the **Entities module**, that contains all the individual entities needed to group and organize information about the inspections. These entities and their relations can be seen in more detail in 4.4. that represents the entities Data Model.

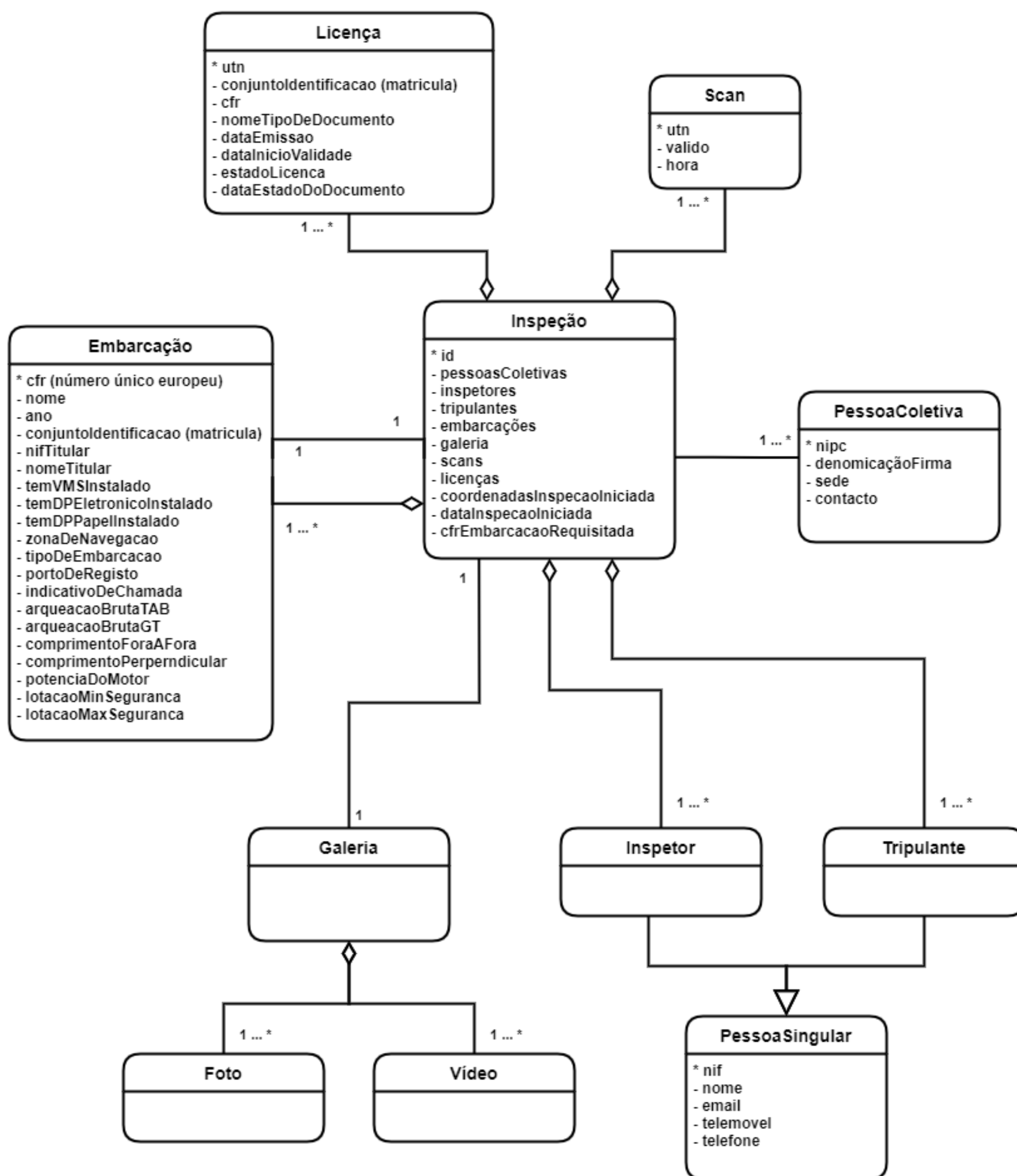


Figure 4.4: Frontend Data model

- **Inspeção**: The main entity that encompasses the information obtained during the inspection and that is stored, in the backend, when the inspection is over. It joins all the information about the vessel, the crew, inspectors, the gallery, scans, licenses, starting time of the inspection and the coordinates;
- **Pessoa Singular**: Represents a unique individual person, containing the NIF (a unique identification number), the name, email, the phone and cellphone numbers;
- **Inspetor**: Represents the inspectors who are carrying out the inspection and extends the PessoaSingular entity;
- **Tripulante**: Represents the members of the vessels crew, who is being inspected, and also extends the PessoaSingular entity;
- **Pessoa Coletiva**: It describes a unique collective person, containing the NIPC (identification number), the denomination, the headquarters/address and contact number, either phone or cellphone;
- **Embarcação**: The entity includes all the data relative to the vessel, such as: the CFR (Community Fleet Register), the name, the identification set, the NIF and the name of the person in charge of the vessel and other information about the port, engine power, dimensions of the vessel, capacity and monitoring devices onboard;
- **Licenca**: It encompasses all information about fishing and other types of licenses. Each license has a UTN (Unique tracking number) used by the DGRM to identify individual licenses, the identification set and CFR of the associated vessel, the type of license and its status (e.g.: valid, canceled, expired), and the expiration date;
- **Scan**: This entity represents the license scans made during the inspection, storing the UTN of the scanned license, its status and the time when the scan was made;
- **Galeria**: And finally we have the entity similar to a gallery application, that holds the information about all the photos (**Foto** entity) and videos (**Video** entity) taken during the inspection, that can later be used in legal proceedings.

The second module that makes part of the Business layer, is the **Handlers module**, containing the components that handle the logic of the different cases supported by the application. the handlers focus on four main instances:
- The *authorization* aspect of the application, where it is handled the core of the authorization process to get into the application and have access to the information. The authentication algorithm is represented in the form of pseudo-code in 1. Firstly the flag, that checks if the user needs to perform the online authentication, is verified. If the flag is true, it means the user already

has the credentials stored in the device and can perform the offline authentication. During the offline authentication, is checked if the user has activated authentication via biometrics (finger print) or just the regular credentials, username/password. If the flag is false, it means the user did not already performed the online authentication or it has been more than 7 days since the last online authentication. In both options, the user must authenticate online, after the credentials being verified by the DGRM authentication service their hash is securely stored in the mobile device and the flag is reset/set for 7 days.

The inspectors credentials are never stored in plain text, are always encrypted before being stored in a secure location by using the package *FlutterSecureStorage*[2]. According to the package, the passwords are stored differently depending on the operating system. On IOS uses Keychain services and on Android makes use of the KeyStore. The encryption is handled by the package *Crypt*[3], that can provide secure one-way string hashing for salted passwords using the Unix crypt format, utilizing the SHA-256 algorithm in the encryption since it is considered one of the strongest hash functions in use today.

Algorithm 1 Mobile application - User Authentication

```
if local credentials are up to date then
    perform offline authentication (biometric auth or username/password auth)
    if credentials are validated by the DGRM server then
        perform online authentication
        set local credentials and update flag
    else
        Error
    end if
else
    perform online authentication
    if credentials are validated by the DGRM server then
        perform online authentication
        set local credentials and update flag
    else
        Error
    end if
end if
```

- The *scan* aspect, that focus on handling the plugins and libraries that allow us to make the scan of QRCodes that accompany the documents/licenses and that provide the UTN of the document, the number it will use to identify and

[2] https://pub.dev/packages/flutter_secure_storage
[3] https://pub.dev/packages/crypt

check the the validity of the documents;
- The *report* aspect, where it is handled all the main information related to the inspection, routing to different views where the information can be seen, created and edited. It also controls the different stages and the flow of the inspection;
- And the last one, that handles the *synchronization* of data with other devices;

In figures A.2,A.3, are presented two different views of the system that shows how the modules in the different layers use the modules and dependencies and how the components in each of the layers interact with other components (internal and external).

- Lastly we have the **Local Database layer**. For dealing with local storage of data in Flutter, exist three main approaches: *storing simple key-value data on the disk (with SharedPreferences)*, *storing highly structured tables (with SQFLite)* and *storing large amounts of data in local files.* For this project we discarded the key-value pairs strategy since it provides a small size storage location, not suitable for the complex data used in the business logic of the application and because it didn't provided a secure storage, saving the plain data without encryption (this approach was only used for storing non sensitive application variables that should persist between sessions). We also discarded the local files approach, since it didn't provided the most efficient way to retrieve/store complex data, even though it offered the possibility to have full control over the encryption of the data. So, we chose the SQFLite storage strategy since it provides support for structured and complex information and relations, provides an abstraction for the handling/storing of data and also provides a secure way of storing that data.

  This layer supports the *Data layer* by providing a local database deployed in the mobile device, making it more secure, since it reduces the communications with external devices that can be intercepted. Having a local and smaller database is also useful to reduce communications with the backend lowering the costs and giving the possibility for the frontend application to have a more interactive, but complex, business logic. Then, the local database is supported by the flutter plugin, SQFLite [4], that offers a self-contained, high-reliability, embedded, SQL database engine.

  The patterns that define this layer can be defined as the *Singleton pattern* and the *Repository pattern.* Since the mobile application is a more simple application that mainly stores information and handles a small portion of the

[4] `https://pub.dev/packages/sqflite`

entire system logic, so it makes sense to have a single database per mobile device. The Repository pattern focus on dividing the logic from the data access, giving the responsibility for the data access to the Storage class.

4.4.2 Backend

In this section, is presented another important part of the development of this project, the backend. It gives a description of the entire backend system and its architecture, based on cloud methodologies, that will support the frontend and the inspectors by allowing them to have access to the necessary data for a accurate inspection.

As the main requirements of the backend system are *Scalability*, *Availability*, *Interoperability*, *Low Platform dependency*, *Usability* and more, the current monolithic system deployed by the DGRM is not able to meet all of the requirements. Thus was requested a solution, an architecture based on the cloud, that complied with the mentioned requirements and to help start the process of transitioning between architectural styles, bringing the DGRM into a new paradigm.

The architecture is based on the Microservices Architectural style, deploying simple microservices that focus on distinct information and support the business logic capabilities. To develop this architecture, were followed a few different design pattern like the *Database per Service Pattern*, where each service has its own smaller database with only the specific information to attend the data requests. Since different services have different data storage requirements and must be loosely coupled, this is a solid approach provided that the databases be private and only be accessed by their associated microservice. Another design pattern utilized, in the development the architecture, was the *Decomposition by Entities*, more particularly decomposition by the data and data relations of each entity. This pattern helps us split the current monolithic database structure into several smaller, entity-focused databases that help and support the associated microservices.

In figure 4.5 is shown the main backend architecture, organizing the system in three main layers, the **Ingress Controller layer**, the **Microservices Deployment layer** and the **Database Deployment layer**.

- The first layer is the **Ingress Controller layer** where the traffic enters the backend system and that consists of a Ingress Controller acting as a Load Balancer for the rest of the system services. Within Kubernetes, an Ingress resource is a collection of routing rules that control how the services managed by the Kubernetes cluster are accessed by external users, usually via HTTP and HTTPS requests.

  In the documentation, the main goal of the Ingress Controller is defined as:

"To expose HTTP and HTTPS routes from outside the cluster to services within the cluster" [5], but it can also be configured to provide other types of services, including *SSL termination*, *name-based virtual hosting*, *authentication*, *support for multiple protocols*, *resilience* (eg.: timeouts and rate limiting) and also *different types of routing*, such as routing based on http request headers, http method, or other specific requests.

It is important to mention that in the Kubernetes environment, the Ingress and Load Balancer are two distinct objects and even though they both have the function to expose services to the outside of the cluster, they do it in different ways. The **Kubernetes Load Balancer** is a type of service and the default way to expose other services to the internet. Each service exposed by a Kubernetes Load Balancer will be assigned its own IP address (rather than sharing them like Ingress) and require a dedicated load balancer, having the need to deploy multiple load balancers, one for each service we want to expose. Another aspect to consider is that the Kubernetes Load Balancer service operates at the L4 level (Transport layer), meaning it will direct all the traffic to the service and support a range of traffic types, including TCP, UDP and also gRPC.

In comparison, the **Kubernetes Ingress** is a collection of rules that route traffic to the services, not a service. It sits in front of all the services we want to expose and acts as the entry point for the entire cluster of pods, allowing multiple and different services to be exposed using a single IP address, contrasting with the Load Balancer. Another difference between these two objects is that the services exposed by the Kubernetes Ingress, operate at the L7 level (Application layer) utilizing HTTP as the main protocol.

- The second layer is the **Microservices Deployment layer** and focuses on the Microservices developed, each one tackling one aspect of the frontend necessities, whether it is the need to access information of the backend or to store information gathered during the inspections. For this project were created seven individual Microservices: *Embarcação*, *PessoaSingular*, *PessoaColetiva*, *Licenca*, *Requisição*, *Inspeção* and *Previsão*, each one dedicated to a specific subsection of the available data and the relations between the data. In 4.4.2 the Microservices are described in more detail and are also presented their communication endpoints, like ports and paths.

- And lastly, the **Database Deployment layer** that deals with everything related to the databases that feed information to the Microservices, from the

[5] https://kubernetes.io/docs/concepts/services-networking/ingress/

service that contains the database to its persistent storage. For this project were created seven distinct databases, each one dedicated to a specific microservice and that shall only be accessed by it. In 4.4.2 these databases are described in more detailed.

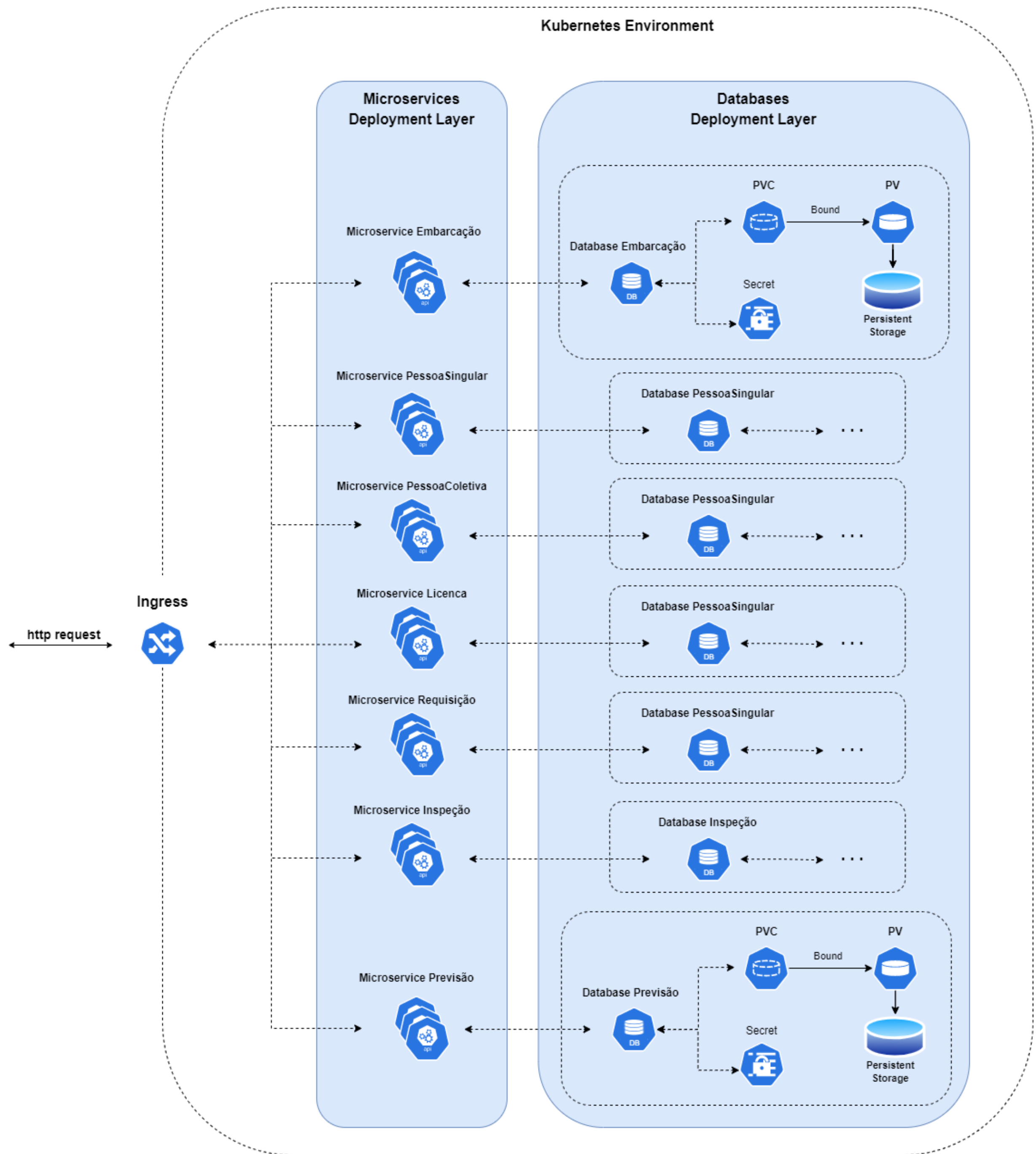


Figure 4.5: Backend Architecture

Microservices

The microservices, that focus on providing RESTful APIs that the consumers can interact with, can be accessed via HTTP requests through the main endpoints **/embarcacao/**, **/pessoasingular/**, **/pessoacoletiva/**, **/licenca/**, **/requisicoes/**, **/inspecao/** and **/previsoes/**. Below are described in more detail each of the developed microservices.

- **Microservice Embarcação/Vessel**: This microservice focuses of delivering information related with the vessel to be inspected by exposing two endpoints, accepting GET requests to either */embarcacao/{cfr}* or */embarcacao/list/[cfrs]* paths. The first endpoint recieves a CFR (Community Fleet Register number) as a parameter and returns a information about the vessel, if the CFR is valid and if it exists in the backend. In the case of a valid CFR the microservice responds with vessel information such as: **name of the vessel, year of construction, CFR, identification set, the NIF and the name of the person in charge of the vessel** and other information about the **port, engine power, dimensions of the vessel, capacity and monitoring devices onboard**. If not, it returns a response with error messages describing the problem (400 - Bad Request, 401 - Unauthorized, 404 - Not Found).

  The second endpoint differs from the latter in that it allows the users to get data about multiple vessels, by passing a list of CFRs as a parameter, instead of only about one vessel. This allows the user to choose the most convenient way to request the information and to avoid load on the server with multiple requests with information about a single vessel.

  Lastly, the microservice is exposed, for the Kubernetes cluster, in the port 5001 and is supported by a instance of a MySQL database, independent from the other microservices and databases.

- **Microservice Pessoa Singular/Individual person**: This microservice concentrates on providing information about the different individual persons involved in the inspection, crew members and also the inspectors that carry out the inspections. There are available four endpoints that can be used to, in the same way as the previous microservice, request a single or multiple inspectors and crew members: */pessoasingular/inspetor/{nif}*, */pessoasingular/inspetor/list/[nifs]*, */pessoasingular/tripulante/{nif}*, */pessoasingular/tripulante/list/[nifs]*

(GET methods). The endpoints receive a unique identification number, NIF (Número de Identificação Fiscal), and return information about the inspectors or crew members if the NIF is valid. This information consists of **the NIF, name, email, the phone** and **cellphone numbers**. If the NIF is invalid or other error occurs, the error messages mentioned previously will be returned.

The microservice is exposed in the port 5002, to the cluster, and is also supported by a distinct instance of a MySQL database, isolated from the other servers and databases.

- **Microservice Pessoa Coletiva/Collective person**: The third microservice is similar to the previous but focus on the information of Collective Persons (eg.: enterprises, companies or other type of associations) related to the vessel or the crew members. There are available two endpoints: */pessoacoletiva/{nipc}* and */pessoacoletiva/list/[nipcs]* (GET methods) and receive one or more parameters, depending on the usage, NIPC that is a unique identification number for collectives. The same error messages, as the previous microservice, will be returned if the NIPC is invalid, but if the NIPC is valid it will return information such as: **the NIPC, the denomination, the headquarters/address and contact number, either phone or cellphone**.

  It is exposed in the port 5003 and as the rest of the microservices is supported by a isolated instance of a MySQL database.

- **Microservice Licença/License**: This microservice provides information about the fishing licenses, associated with the vessels, and also provides a way to verify licenses based on the UTN (Unique Tracking Number) that identifies the licenses in the backend. The first endpoint, */licenca/list/[cfrs]* (GET method), receives a list of CFRs (Community Fleet Register number) as a parameter and returns a list of Licenses related to the vessels being inspected, if the CFRs exist in the backend and are valid. The information contained in the License is: **CFR, identification set, UTN (Unique Tracking Number), type of document, dates of emission and validity** and also the **state of the license (VALID, INVALID or others**.

  The second endpoint */licenca/verificarLicenca/{utn}* (GET method) focuses focuses on verifying the validity of the license, returning an error message if the license is not valid or other error occurs and returning a response with 200 status code. As a parameter it accepts a different variable, UTN (Unique Tracking Number) that identifies the License in the systems backend.

The microservice is exposed in the port 5004 and, like the other microservices, it also has a separate instance of a MySQL database to support it and maintain the information isolated.

- **Microservice Requisições/Requisitions**: This microservice focuses on providing assistance in the requisition of the vessels used by the inspectors. It provides three endpoints that aid the inspectors in knowing what vessels are available in some determined port (eg.: Porto de Lisboa, Viana Do Castelo, Cascais) and also aid in making the requisition and release the vessel once the inspection is finished. The first endpoint */requisicoes/list/{registrationPort}* (GET method), accepts as a parameter the port in which some vessels may be registered, if the port is invalid or it doesn't exist it will return a error message and will return a list of information about the available vessels that can be used to initiate and end the requisitions, such as: the **CFR, identification set** and the **registration port**.

  The second and third endpoints, */requisicoes/{cfr}* and */requisicoes/terminarRequisicao/{cfr}* (PUT methods) respectively, will accept the CFR of a vessel as a parameter and lock that vessel so that it isn't available for other inspections until the current inspection is finished and the vessel released. The third endpoint will release the vessel that was previously requested.

  This microservice follows the same design structure as the other, having an isolated MySQL database supporting it and is exposed, for the Kubernetes cluster, in the port 5005.

- **Microservice Inspeção/Inspection**: This microservice provides a different type of interaction since it only has POST methods. It focuses on the storing of all the information gathered during the inspection, like the **information about the crew members, inspectors, collective persons, inspected vessels, documentation scans** and other data like the **inspection ID, coordinates, the date and also the photos/videos taken**. The two endpoints have similar goals, to store information, but while the first endpoint */inspecao/inspecao* (POST method) stores all the gathered information in text format, the second endpoint */inspecao/upload/{inspectionID}* (POST method) is dedicated to receiving the information in other formats (pdf, png, jpg, jpeg, gif, mp4), like photos and videos.

  The microservice is exposed in the port 5006 and follows the same structure as the previous microservices, being supported by a dedicated MySQL database.

- **Microservice Previsões/Predictions**: This last microservice focuses on giving the user possible predictions regarding the next subject of inspection, having a simple model that checks if a vessel is at a distance less then 100km from the coordinates passed as the parameters, LAT and LON (representing the current latitude and longitude of the inspector). Other approaches can be followed and a more complex predictive model can be developed, focusing also on the distance to the closer ports, the amount of times it has been inspected in the last X days and other metrics that can be useful for the topic.

  Is the most simple, when compared with the numbers of endpoints of the other microservices, since it only has a single endpoint */previsoes/previsoes/{lat,lon}* (GET method). As already mentioned, it accepts as parameters the current latitude and longitude of the inspector and returns a list with predictions for future inspections.

  Following the trend from the previous microservices, it is exposed in the port 5007 and is also supported by a dedicated MySQL database. One important fact regarding the database is that the values used in the files that populate this database don't have a basis in real positioning of the vessels in real time, since some of that data is not public and was not shared with us by the DGRM. That said, the vessels can be monitored in real time by the DGRM and eventually this microservice can be integrated with the their tracking service.

In the table below, 4.3, are mapped the endpoints for the communication with the different developed microservices in a more concise and simple manner. Are presented the ports for each of the services, starting in the 5001 for the Microservice Embarcação/Vessel and ending in the 5007 for the Microservice Previsões/Predictions (these ports will not be accessed by external users, are used only by the Ingress Controller for the routing of traffic). Next, are presented the different paths to the microservices and the http methods for the specific path.

Database

During the development of databases, in the context of microservices, new challenges can arise when compared with the development of the usual single database that serves monolithic architectures. Since microservices are built to be smaller, independent and specialized applications, is common to emerge the need to access data from multiple microservices and to have inter-communication between services.

One of the main problems regarding this approach is that is very challenging to maintain ACID (Atomicity, Consistency, Isolation and Durability) properties in distributed transactions involving multiple databases.

Microservice	Port	Path	HTTP Method
Embarcação	5001	/embarcacao/{cfr} /embarcacao/list/[cfrs]	GET GET
PessoaSingular	5002	/pessoasingular/tripulante/{nif} /pessoasingular/tripulante/list/[nifs] /pessoasingular/inspetor/{nif} /pessoasingular/inspetor/list/[nifs]	GET GET GET GET
PessoaColetiva	5003	/pessoacoletiva/{nipc} /pessoacoletiva/list/[nipcs]	GET GET
Licença	5004	/licenca/verificarLicenca/{utn} /licenca/list/[utns]	GET GET
Requisições	5005	/requisicoes/list/{registrationPort} /requisicoes/{cfr} /requisicoes/terminarRequisicao/{cfr}	GET PUT PUT
Inspeção	5006	/inspecao/inspecao /inspecao/upload/{inspectionID}	POST POST
Previsões	5007	/previsoes/previsoes/{lat,lon}	GET

Table 4.3: Summary of the developed microservices details (ports, paths and HTTP methods)

To solve this problem several patterns emerged, providing solutions for how to handle data storage in a microservice architecture, including: *The Database-per-Service Pattern*, *The Database Cluster Pattern* and the *The Shared Database Server Pattern*.

- The **Shared Database Server Pattern** proposes storing the data on a single shared database. This is probably the closest microservice pattern to a monolithic implementation and therefore the main advantages, like: the simplicity of the migration from monolithic applications, since existing schemas can be reused without any changes, and promotes data consistency, since the existing code base will not suffer significant modifications. The main advantages of similar system also apply, since it is hard to scale and does not promote data isolation.

- The **Database Cluster Pattern** is similar to the Shared Database Pattern but it proposes storing the data on a database cluster, instead of storing it in a shared database. From the microservices perspective, both patterns are identical since in both cases the database is accessed in the same way, the only difference is internally in the adopted database. This approach aims to improve scalability and also preserve data consistency, where microservices have a sub-set of database tables that can be accessed only from a single microservice; in other cases, each microservice may have a private database schema. Some of the main disadvantages include the increased complexity due to cluster architecture and increased risk of failure, due to the introduction of another component and the distributed mechanism.

- The **Database-per-Service Pattern** is a different pattern, where each microservice accesses its private database. Since there are dedicated databases for each service, it becomes easier to improve scalability by scaling only the necessary services and supporting databases. It also allows for a improvement in independent development where teams can work independently on a service, without influencing the work of others in case of DB schema changes and data security. This is the easiest approach for implementing microservices-based systems, being used often in migrations from monolithic architectures, but is also has some disadvantages, such as: the need to split data, creating more points of failure that need to be coordinated, and also a decrease in data consistency, since some data will need to be stored in different places making.

When choosing the most appropriate pattern to the developed backend system databases were taken in consideration the advantages and disadvantages of each pattern presented, the low scalability power of the the *Shared Database pattern* makes it a poor choice, since one of the main characteristics of the the microservices architectures is its scalability. The *Database Cluster pattern* provides improvements in this specific problem but in doing so, creates a distributed system of high complexity that is prone to failures. The last pattern presented, the *Database-per-Service pattern*, offers a scalable but simple solution that also allows developers to take advantage of another of the main characteristics of microservices architectures, independent development of services. The main disadvantage of the approach being the difficulty of maintaining the data consistent, but even if it is a disadvantage it is still something that can be seen as a positive aspect because it provides data isolation, making the microservices more secure. Thus, the developed backend databases where developed according to the *Database-per-Service Pattern*.

Regarding the system backend that supports the current monolithic architecture, it is deployed in the DGRM servers, where are also deployed other applications related with the field and that in the future may come to communicate with the system being developed. Curently it is deploying an Oracle Database, Oracle 11.2 for all the deployed applications in the DGRM environment. While initially would be simpler to access the data from this already existing database, according to M. Fowler [22] Microservices implementations tend to prefer for each different service to manage its own database, an approach called "Polyglot Persistence", that translates in the pattern mentioned above "Database-per-Service". These database systems can be entirely different databases or just different instances of the current database technologies, and even though they can be costly to implement initially, having the need to split the data and structure it in a new format, they also offer benefits like the ability to scale horizontally by running multiple instances over a cluster.

Since the DGRM database has confidential information, we did not have full access to the stored information and existing schemas in their system. Instead, were provided two files with information that we could use to separate and re-create the information access. The first file consists of information about the vessel and the second file contains the main information related to the vessel-related licenses. From that information were created seven distinct databases, where each one supports a different microservice.

These databases, like the microservices, were designed to be deployed as Kubernetes containers making the whole backend system deployable in the Kubernetes environment. Later, the DGRM can deploy these databases in their own machines or create their own environment based on the one developed in this project.

In Kubernetes databases and data persistence have been a highly discussed topic, since containers were created to be immutable, meaning that when a container shuts down or is re-deployed all data created during its lifetime is lost, contrasting with the main goal of databases. Kubernetes is inherently an ephemeral system and persistent storage by definition must survive, so to solve this problem Kubernetes provides a convenient persistent storage mechanism for containers based on the concept of a *Persistent Volume* (PV).In order to assure the persistence of all the data in the database a PV is created so that Kubernetes API can reserve the required space in the cluster for the data. Besides that, are also created: a *Persistent Volume Claim* (PVC) that looks for a existing PV that meets the criteria defined in the user's PVC, and if there is one, it matches the PVC to the PV, in a process called binding; a *Secret* that stores the confidential information such as passwords and other needed variables that should not be directly used in the Deployment, to increase security; and a *Service and Deployment* to deploy the database service, without exposing it what makes it impossible to connect to from the outside.

The interactions between these components are presented in the figure 4.6 below.

A sample of the kubernetes objects used for the database deployment can be seen in the figure 4.7, starting with the PV and PVC configurations. It requests 2Gb of space from the cluster and the 'ReadWriteOne' access policy defines that the volume can be mounted as read-write by a single node and the hostPath is defined, pointing for a directory inside the kubernetes cluster where the data will be stored and persisted.

Another object present, is a Secret, where is stored all a small amount of sensitive data such as: the username and password for the root user, the name of the database and other variables necessary to initialize the BD. Such information might otherwise be put in a Pod specification or in a container image, but using a Secret means that

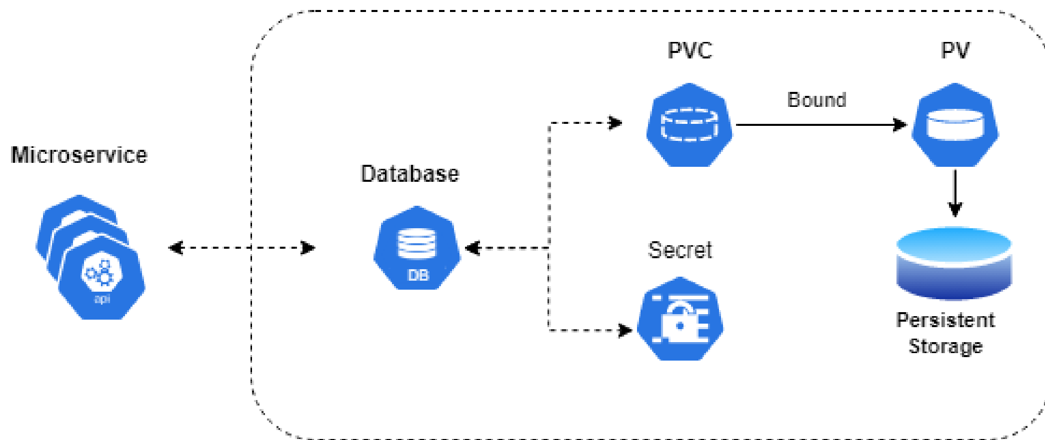


Figure 4.6: Kubernetes elements that compose the microservices databases and their relations

you don't need to include confidential data in your application code.

And finally, the Service, that exposes each database in the port 3306 and is accessed by its name, and the Deployment, that uses the variables stored in the Secret, connects the correct PVC to the database and uses a Recreate strategy to define how to create, upgrade, or downgrade the application.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysql-db-emb-secret
type: kubernetes.io/basic-auth
stringData:
  username: root
  password: test1234Embarcacoes
  MYSQL_DATABASE: local-db-emb
  MYSQL_USER: dbUser1
  MYSQL_PASSWORD: dbUserPassword1
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mysql-db-emb
spec:
  ports:
  - port: 3306
  selector:
    app: mysql-db-emb
```

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-db-emb-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/disks/db_emb"
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-db-emb-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql-db-emb
spec:
  selector:
    matchLabels:
      app: mysql-db-emb
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql-db-emb
    spec:
      containers:
      - image: mysql:5.7
        name: mysql-db-emb
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-db-emb-secret
              key: password
        - name: MYSQL_DATABASE
          valueFrom:
            secretKeyRef:
              name: mysql-db-emb-secret
              key: MYSQL_DATABASE
        - name: MYSQL_USER
          valueFrom:
            secretKeyRef:
              name: mysql-db-emb-secret
              key: MYSQL_USER
        - name: MYSQL_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-db-emb-secret
              key: MYSQL_PASSWORD
        ports:
        - containerPort: 3306
          name: mysql-db-emb
        volumeMounts:
        - name: mysql-db-emb-persistent-storage
          subPath: data
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-db-emb-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-db-emb-pv-claim
```

Figure 4.7: File mysql-deployment.yaml containing the specification of the Database for the Embarcação/Vessel Microservice

During the development of databases, the essential data for the microservices was divided into 11 different tables (B.1, B.2, B.3, B.4, B.5, B.6, B.7, B.8, B.9, B.10, B.11). The tables were populated with the information provided by the DGRM, and each database focus on storing only the information related with the microservices, that being, **the vessels**, **crew members/inspectors**, **collective persons**, **licenses**, **requisitions**, **predictions** and the **inspection as a whole**.

Originally, the development of the databases services was focused on maintaining the consistency with the current databases of the DGRM and be deployed in Oracle Databases. This would facilitate a possible future transition, since the database systems would be the same. The communication with the databases would be done via a open-source module, *CX_Oracle* [6], that enables access to Oracle Database, conforms to the Python database API specification and is also compatible with Oracle Databases 11.2, the database version currently running in the DGRM machines. To create the Oracle Database 11.2 services in the Kubernetes environment, the Oracle team has made available resources that provide developers with the possibility to build an Oracle Database Docker image [7]. The sample container build files aim to facilitate installation, configuration, and environment setup for DevOps users, making development and testing of new applications easier to integrate with existing systems, but several problems arose when trying to create multiple instances of the Oracle databases on different ports, other than the default.

A solution for this problem was to change the ports, described in the files used to build the Oracle Database image, but in the building process the ports returned

[6] `https://cx-oracle.readthedocs.io/en/latest/`

[7] `https://github.com/oracle/docker-images/tree/main/OracleDatabase/SingleInstance/dockerfiles/11.2.0.2`

to the default values and the listener couldn't find the requested service in the connection descriptor.

To circumvent this issue, we decided to change to another Relational Database Management System, MySQL. Compared to MySQL, Oracle SQL is a closed source, meaning that exists less transparency about the product and less options and documentation to guide developers in creating products that are more customizable and flexible. In contrast, MySQL is open-source, has the benefit to support some types that Oracle SQL does not, it also supports more operating systems and can be modified, customized and altered for individual environments based on different requirements of the microservices. Lastly, both support SQL language, which facilitated the migration process from the Oracle SQL to the MySQL Database.

A summarized version of the information regarding the kubernetes services databases names, the ports to access the databases and the microservices names associated with them are shown in the table 4.4.

Microservice	DB service name	DB Port
Embarcação	mysql-db-emb	3306
PessoaSingular	mysql-db-user	3306
PessoaColetiva	mysql-db-pc	3306
Licença	mysql-db-lic	3306
Requisições	mysql-db-req	3306
Inspeção	mysql-db-insp	3306
Previsões	mysql-db-prev	3306

Table 4.4: Summary of the developed databases details (DB ports, kubernetes database services and the supported microservice)

4.5 Inspection Predictions

As discussed in the last section, the microservice focused on providing predictions of possible new inspection targets is a simple service but an important one, a service that with some development and research can be improved and refined to better support one of the most important steps in the process of beginning a vessel inspection.

One of the problems to solve, regarding the vessel inspections, is knowing the correct amount of information to provide the inspectors with and still give them the additional data to act and not be restricted, in uncommon occasions.

For security reasons, only the necessary information to the correct operation of the inspections should be stored in the inspectors mobile device and since this data is acquired in the initial stages of the inspection when there is access to the network, and thus to the backend, all the needed information is available to the

inspectors. The problem arises when the inspectors, already in the middle of a vessel inspection, have the need to inspect other vessels that either, were not in the initial search area having moved to range in the meantime or that were in the initial search area but were not deemed as suspects initially. This poses the main problem, since the nautical space changes over time and there is a limit on how many days an inspection can last (maximum, 7 days) it is necessary to have, at least, the licenses information of the potential inspection subjects.

To improve the system and give the inspectors the option to follow through with a second unscheduled inspection to a new vessel, it will be necessary to predict which vessels will be within range of the inspectors in the future and get their licenses information, as well as other information deemed necessary.

4.5.1 Predictions Models

Currently, to monitor vessels location (and as a part of the MONICAP system mentioned in chapter 2), the DGRM makes use of the Vessel Traffic Service (VTS). VTS is a vessel traffic monitoring system that employs radar technology, VHF radiotelephony, and the Automatic Identification System (AIS) to track vessel movements and ensure safe navigation in restricted areas. The AIS is a system used globally due to its reliability, it uses transceivers on ships that transmit data via two methods: *Terrestrial-based AIS (T-AIS)* used when the vessel is in the range of 70/100 Km and *Satellite-based AIS (S-AIS)* when the vessel is in a range superior to 100 Km. Since it is a widely used system several studies and research, specifically on the prediction of vessel paths, were conducted with the AIS system as its basis for vessel data acquisition.

Then, some of the main research regarding vessel path predicting will be presented and discussed to hopefully provide a foundation to future development of this problem and consequently improve the predictions microservice.

In [64] is developed a model for vessel trajectory prediction based on a Long Short-Term Memory (LSTM) neural network, used to observe the first 10 min of the ship's state to predict the location of the ship 10 min later, thus predicting the next location of the ship every 20 min. The developed neural network is composed of two LSTM layers and was trained, with AIS information regarding the Tianjin port (in China), by predicting the probability distribution. In the experimental comparisons, performed against other vessel prediction models based on the Extended Kalman filter (EKF) [35, 34, 33] approach and a Back Propagation Neural Network (BPNN) [10] approach, it achieves a better performance in the field of long-term ship position prediction even though the stability of the prediction needs to be further improved, due to the substantial influence of the initial value of the neural network on the predictions.

A related approach was developed in [60], by introducing a ship trajectory prediction framework based on a Recurrent Neural Network, using more specifically a Gate Recurrent Unit (GRU) model, taking into account vessel navigation errors. The model was also tested and trained with real data, from the Zhangzhou port (in China). In a comparative experiment, conducted against the LSTM approach mentioned before which is considered as a valuable solution, the results showed that the GRU model had a good prediction accuracy (up to 96% and 98% in some cases) similar to that of LSTM but it had the advantage of being computationally more efficient, thus being more suitable for the requirements of immediate and early warnings of maritime navigation and can also provide appropriate decisions for intelligent vessel navigation systems and VTS. LSTM is similar to GRU, but is less effective in long-distance trajectory predictions.

Although the models mentioned above have good values of accuracy, they all focus on a single trajectory learning, meaning that the data used to develop the models is associated with one target vessel and thus cannot be used to predict other vessel's future positions. This is a major disadvantage, but there are few models that focus on combating this problem. In [68] was proposed a new approach, focusing on predicting various vessels paths with different geographical context and for different types of vessels. In the paper, is proposed a comprehensive framework for massive real-world historical AIS data learning. When quantitatively compared with other approaches, such as: Gaussian Process Regression (GPR), Least Square Support Vector Machine (LS-SVM), Multilayer Perceptron (MLP) and Gaussian Mixture Models (GMM) achieved better results in one-shot learning (train only once and no successive update required) and was also able to predict paths for multiple types of vessels, what other models had difficulties doing. (The algorithms for comparison were chosen as the baseline because, according with research presented by the paper, they are popular for path prediction and able to achieve state of the art results). The experimental results have demonstrated its effectiveness for handling the diversity, divergence, and imbalance in trajectory data, nevertheless there are still some limitations present, since it requires big amount of historical data to train the models and thus may limit its application to the scenarios where only very limited data are available and it has many user parameters which need to be manually tuned in order to obtain good prediction results, making it difficult option to implement in production in its present state.

In short, the LSTM and the GRU models are strong candidates to research, improvement and integration in the predictions microservice, but have a major drawback since they lack the generality of being applicable to any vessel and it can be hard to gather the amount of necessary information for training and testing these models. On the other hand, they can have high accuracy scores and make reliable

predictions, at the cost of possible extra computational burden for the system. The last model, is more generic and can be applied to many different types of vessels, but required even more information and needs to be manually tuned in order to obtain good prediction results. Future research can compare the different models presented in this section, and other promising methods, and provide an answer to the problem.

Chapter 5

Implementation and Results

In this chapter are described the tools and technologies used in the development of both the backend architecture and the frontend mobile application, presented in the previous chapter. It also introduces the testing of the project mobile application performance and usability.

5.1 Frontent implementation

The development of mobile applications can be a complex and difficult field to fully comprehend, since in the last decade grew massively in popularity and many different tools were created to support this increasing need for a better performance and also better appearance, since this type of application must be, not only capable of performing the tasks that the users expect of them, but also to be appealing and fairly easy to use. Nowadays there is a lot of variety in mobile development tools, from the most used and popular, such as *Android Studio*, *XCode*, *Flutter* and *React Native*, to the more specific ones, such as *Unity* that is more focused on mobile game development, and with all this diversity it can be difficult to pick the right one for our specific needs.

Android Studio, which is a powerful and sophisticated development environment, designed with the specific purpose of developing, testing, and packaging Android applications - allows the developers to program mobile applications, based on the Android OS, in different programming languages, like Java, Kotlin and more recently, Dart. It was built by Google and became one of the most used IDEs for the development of mobile applications by having a growing community of third-party plugins that provide a large array of valuable functions, that help increase the speed up build times, debug a project, and many more.

XCode is Apple's main IDE for developing any type of application and software for macOS, iOS, iPadOS, watchOS, and tvOS. It supports source code for the programming languages: C, C++, Objective-C, Objective-C++, Java, AppleScript,

Python, Ruby, ResEdit (Rez) and also Swift, one of the the most used programming languages for the development of software in the Apple's ecosystem and a replacement for Apple's earlier programming language, Objective-C.

The main problem with the mentioned tools is the fact that they are plataform specific. Android Studio only allows developers to create applications for Android devices and XCode is more focused on the IOS development, since it is only available for macOS devices.

To combat this lack of portability between operating systems, Google developed Flutter in 2017. *Flutter* allows developers to create a multi-platform native mobile application capable of running on both Android and IOS devices, with only one codebase and using only one programming language (Dart). Flutter, as well as creating applications for IOS and Android, it can also be used in the development of applications for windows, macOS and Linux.

It also enables developers to make instant changes in the applications, with the hot reload feature that make any changes the developers make in the code appear instantly in the app (reducing the number of times that the application needs to be initialized during development), it has a high performance making for a better and smother user experience and with a single code base, its easier and less expensive to deploy applications, making the development process faster and more efficient.

The disadvantages focus on the size of the apps, that tend to have a bigger size in comparison with other development tools and in handling its main problem, the state management problem. State management problems are one of the most crucial and challenging topics [14], having several answers for different and specific cases, as mentioned in the Flutter documentation about the topic [1].

5.2 Backend implementation

Regarding the aforementioned microservices, they were developed using Python. Since it is a dynamic and interpreted language with great support, both for creating network connections and writing REST services it is a popular language as a backend language in web applications, particularly in small and contained ones, just like microservices we wanted to develop. It also has a vast community and ecosystem of third party libraries that improve the productivity by simplifying the development.

One of the most used and well supported framework to develop web applications with *Python*, and the one chosen to develop this project, is *Flask*. Flask can be defined as a micro framework, making the core functionality simple but extensible, providing a basic foundation of web framework but allowing more plug-ins to be

[1] `https://docs.flutter.dev/development/data-and-backend/state-mgmt/options`

added, improving and extending the feature set while maintaining the application manageable and without unnecessary features. Some of the main features of flask are: *Integrated supports for unit testing*, *Support for secure cookies*, *Extensive documentation*, *Google app engine compatibility*, *Restful request dispatching* and *Unicode based*.

Another important feature of Flask is the compatibility and ease of integration with the *OpenAPI Specification*. OpenAPI Specification (formerly known as Swagger Specification) is an API description format for REST APIs that provides standards and helps with the design, build, document and consume RESTful Web services easily by allowing developers to describe the entire of API structure, including the Available endpoints (/embarcacao) and operations (GET /embarcacao/{cfr}), the operation parameters, the expected structure of the input/output, error messages, and other information as licenses, terms of use and contact information. This specification is usually written in a YAML or JSON format and by reading the API structure, OpenAPI can automatically build beautiful and interactive API documentation. It also automatically generate client libraries for your API in many languages and explore other possibilities like automated testing.

Since the microservices focus on providing RESTful APIs, OpenAPI is a good tool to provide a clean, understandable and easy to use. These pages documenting the application can be accessed by the users by using the a special path in each microservice: */micro_service_main_path/ui*.

The connection with the databases are achieved with a *MySQL Connector Driver*, developed and provided by the MySQL team making the communications as standard as possible.

The microservices themselves are deployed in the Kubernetes environment using a .yaml file that contains the specification for each Kubernetes Service and Deployment objects that represent the microservice. In Figure 5.1 is displayed the initial configuration of both the Deployment and the Service objects that represent the microservice dedicated to handle all the information about the vessel data. In this specification it is possible to see most important parts of the microservice, from the type of the Service (NodePort) and its ports/protocols to the strategy and number of initial replicas used in the Deployment.

One other important aspect is the reference to the image used to build the application. The referenced image was created using Docker and the Dockerfile present in the folder of the microservice and can be built locally or pulled from a image repository compatible with the system.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: embarcacao-service
  namespace: default
  labels:
    app: embarcacao
spec:
  selector:
    app: embarcacao
  type: NodePort
  ports:
  - name: http
    protocol: TCP
    port: 5001
    targetPort: 5000
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: embarcacao-deployment
  namespace: default
  labels:
    app: embarcacao
spec:
  strategy:
    type: Recreate
  replicas: 2
  selector:
    matchLabels:
      app: embarcacao
  template:
    metadata:
      labels:
        app: embarcacao
    spec:
      containers:
      - name: embarcacao
        image: pedrosobreira/embarcacao:v1.0
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 5000
```

Figure 5.1: File microservice-embarcacao.yaml containing the specification of the Microservice Embarcação/Vessel

5.3 Results

With the developed systems described in previous sections and chapters, the performed tests and their results are presented and discussed.

The main tool used to obtain and measure the results regarding the mobile application performance was the *Dart DevTools*[2], a suite of performance and debugging tools for Dart and Flutter. Dart DevTools allows developers to: Inspect the UI layout and state of a Flutter app, Diagnose UI jank performance issues in a Flutter app, CPU profiling for a Flutter or Dart app, Network profiling for a Flutter app, Source-level debugging of a Flutter or Dart app, Debug memory issues in a Flutter or Dart command-line app, View general log and diagnostics information about a running Flutter or Dart command-line app and Analyze code and app size.

Flutter has three types of launching modes: Development, Debug and Profile. The performance and all times were measured using Flutter's *Profile Mode*[3], that compiles and launches the mobile applications almost identically to release mode,

[2]https://docs.flutter.dev/development/tools/devtools/overview
[3]https://docs.flutter.dev/perf/ui-performance

but with just enough additional functionality to allow debugging performance problems, creating an environment similar to the one where users will interact with the mobile application.

During the measurements, it was also considered the use of three main threads of a Flutter application:

- **Platform thread**: where the main computations of the application are performed.
- **UI thread**: The thread where the application creates and displays a scene. The UI thread creates a layer tree, a lightweight object containing device-agnostic painting commands, and sends the layer tree to the raster thread to be rendered on the device.
- **Raster thread**: This thread takes the layer tree and displays it by communicating with the GPU. This thread was previously known as the "GPU thread" because it rasterizes for the GPU, but it runs on the device's CPU.

Finally, all the tests conducted on the mobile application were carried out on the same test device, a Redmi Note 7 Pro with the following specs: 6GB of RAM, a Snapdragon 675 Octa-core 2.02GHz CPU, and 10QKQ1.190915.002 android version.

5.3.1 Mobile application adaptability to the network connectivity

One of the main objectives of the performed tests, was to focus on the mobile application adaptability to the loss of network connectivity, since this is a case that can occur regularly during maritime inspections as was already mentioned. These tests focus on the main cases of *authentication*, *documents verification and validation*, *starting an inspection* and also *ending an inspection*, since other cases, such as: *accessing the inspection report* and *adding and/or editing the inspection report information* are not restricted by the network connectivity and are all handled locally, not depending on the information of the backend microservices.

Authentication case

In the first case, user authentication in the mobile device, are presented two distinct paths that the application can take, the *online path*, where the authentication is achieved by communicating with DGRM services, and the *offline path*, where the authentication is achieved locally, in the mobile device itself. In each case, online or offline, are considered three main situations:

- **the first time the application is launched**;

- **the case where the authentication token is valid** - less than 7 days have elapsed from the last time a online authentication was performed;
- **the case where the authentication token limit has expired** - 7, or more, days have elapsed from the last time a online authentication was performed;

In the diagram 5.2 are represented the flows of the application from the different cases of the authentication process, regarding the network connectivity during the performed tests. These cases were tested by performing the authentication on the device with both the network connectivity turned ON and OFF, and by switching between those states to encounter edge cases where the connectivity is temporarily lost.

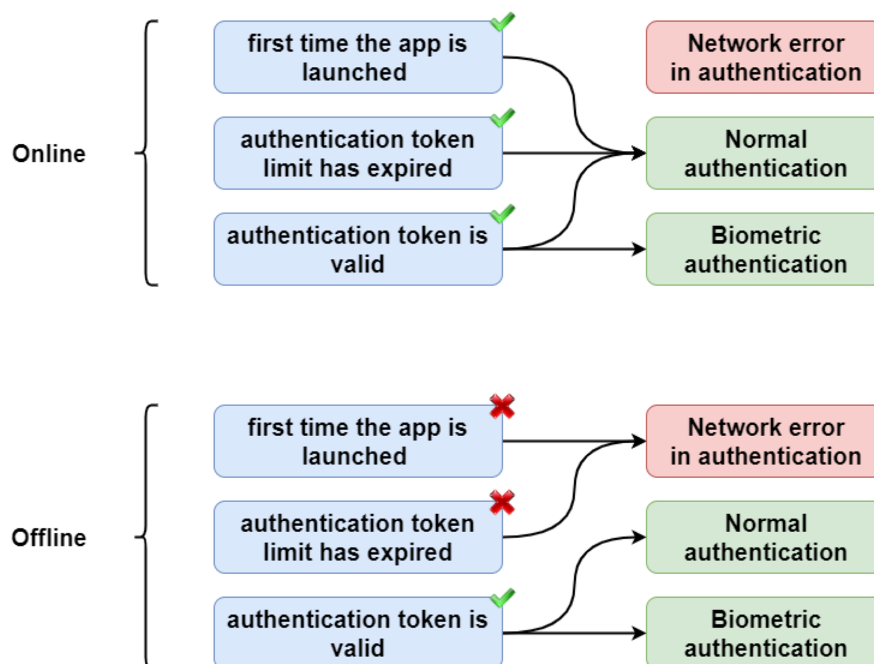


Figure 5.2: Flow of different states during the authentication case

Regarding the *online path*, the application always tries to route the user to a successful authentication. In the first time the app is launched, the user is automatically routed to the default authentication method, using login and password credentials. In both cases, where the authentication token limit has expired and authentication token is valid, the authentication is also always available, depending only if the user has activated the Biometric authentication, in the app settings, or if the token needs to be refreshed, where the user is requested to provide the default authentication credentials.

The figure 5.3 depicts the three different screens that the user can be presented with, in the authentication case.

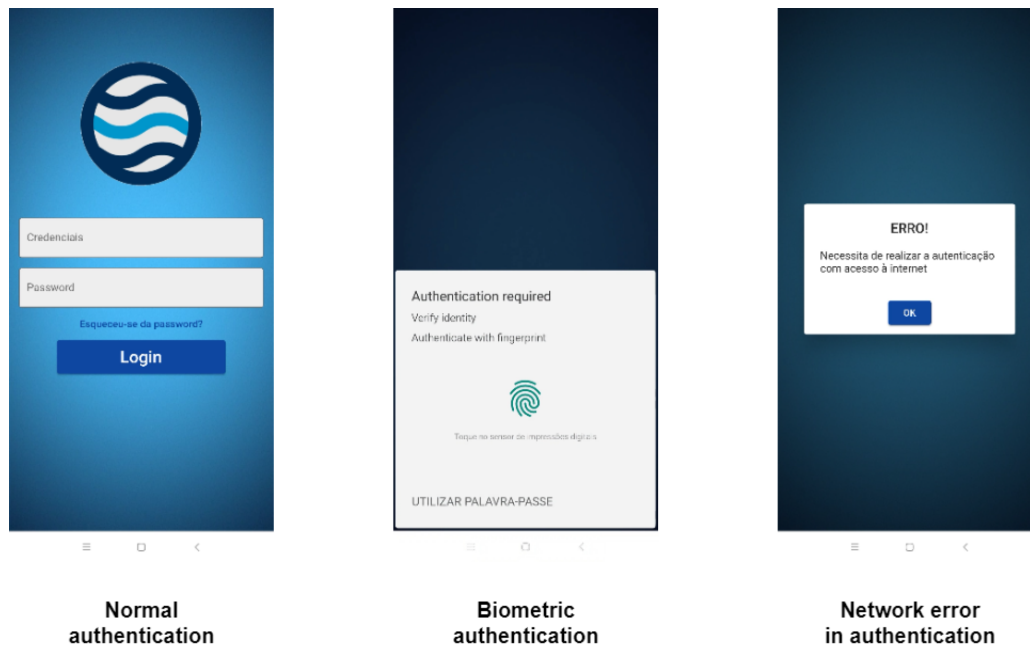


Figure 5.3: Application screens regarding the online/offline authentication process. Normal authentication Screen, Biometric authentication Screen, Network error in authentication (from left to right)

Verification and Validation of documents

The second case focuses on the verification and validation of documents, via the mobile application. These documents data is loaded into the device when the inspection starts and help the inspectors by providing them with information about the type of species and quantity that can be caught, the license validity date, the area where it is valid and other details. There are also two distinct paths, *online path*, where the documents are verified and validated by using the developed microservice Licença, and the *offline path*, where the documents are verified and validated locally, in the device.

In this case, the network connectivity only influences the variability of the licenses that can be verified and validated. With network connectivity, the application can validate any license available in the system, but without network connectivity the application can only validate licenses that have been previously uploaded to the device. Thus, simulating a case where two types of licenses, "not valid" and "valid", are validated via QRCode during a planned inspection, no adaptation problems to network unavailability were identified.

The figure 5.4 presents the screens regarding the scans of valid QRCodes, with valid and invalid documents, and the scan of invalid QRCodes, that will provide a

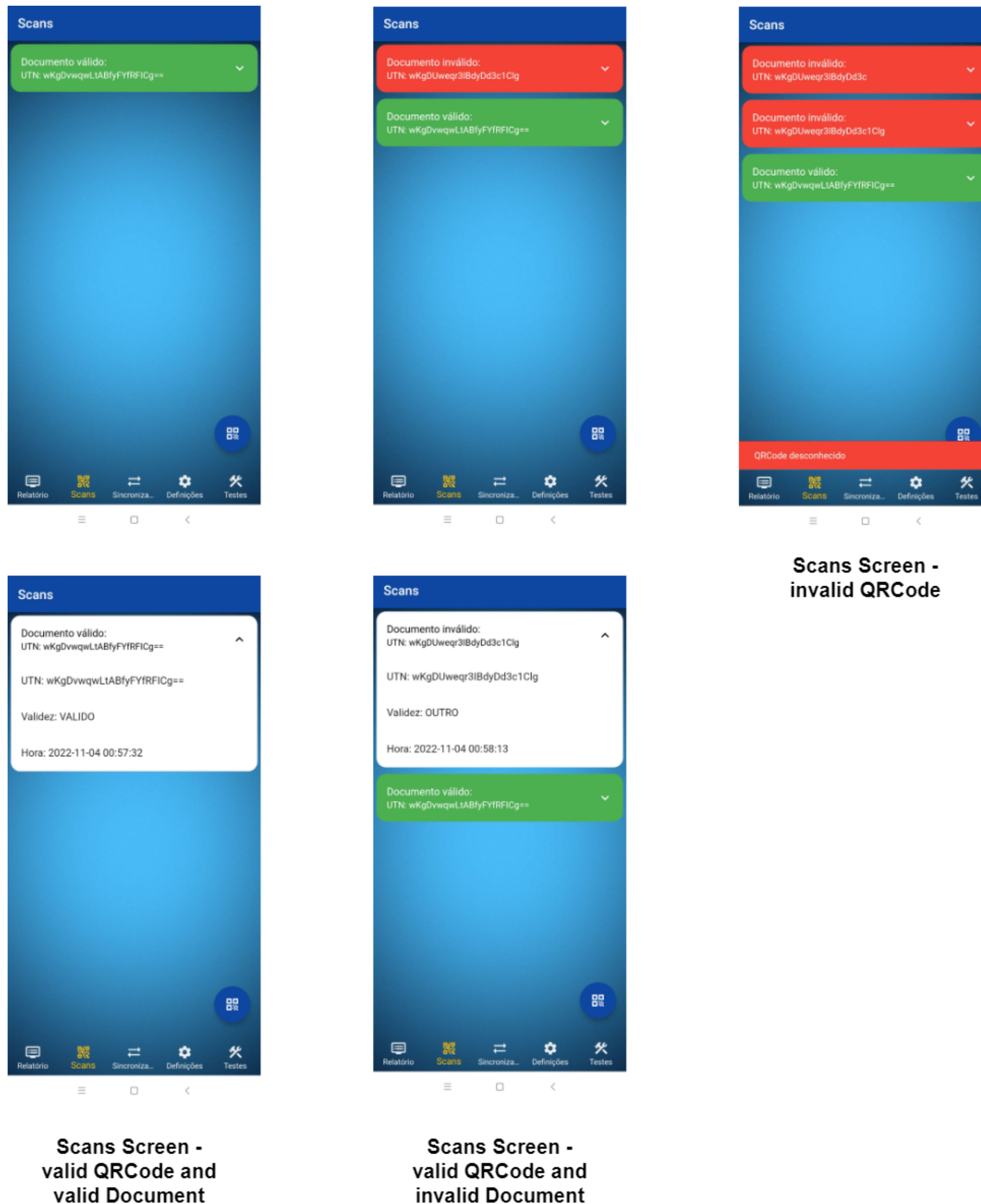


Figure 5.4: Application screens regarding the verification and validation of documents/licenses process. Valid license verification, Invalid license verification and Invalid QRCode verification (from left to right)

invalid response and an error message. The validation and verification screens are identical in both cases, with or without network connectivity.

Starting/Ending Inspection

The third, and last, case focuses on two actions: *the start of an inspection*, by retrieving the required information, such as: vessel and crew information, licenses, requisition of a vessel for the inspectors and previsions, from the DGRM backend systems, and also *the end of an inspection* where the data acquired during the inspection is stored in the DGRM backend systems.

This case is also affected by the network connectivity, being one of the most network dependent cases where the lack of connectivity can be detrimental for the proper use of the application and the continuation of the inspection process. Regarding the *online path*, the start of the inspection should enable the inspector to choose which vessel to inspect and make a vessel requisition to the backend; the end of the inspection should allow the inspector to finish the inspection by storing the information in the backend. In the *offline path*, the inspectors should not be allowed to start, or to end, the inspection by presenting a error message with the correct information, since they dont have access to the backend to store the information acquired during the inspection.

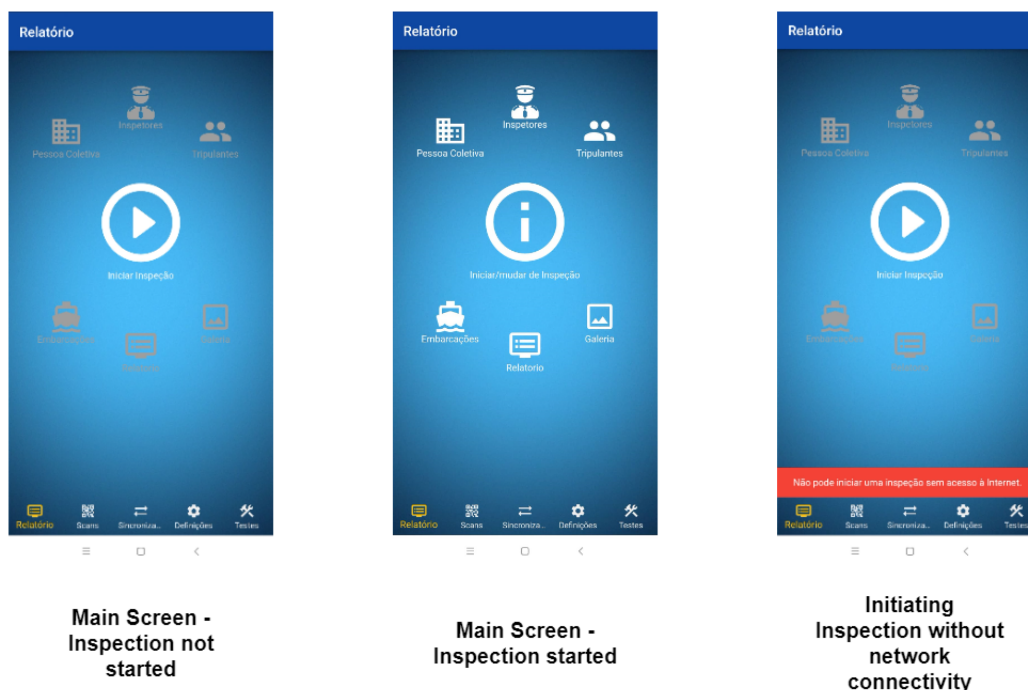


Figure 5.5: Application main screen adaptability regarding changes in the network connection. Main screen, with inspection not started and started, and network warning while starting inspection without network connectivity (from left to right)

The figure 5.5 shows the main screen of the mobile application before and after

the inspection was started, with network connectivity, and also the screen without network connectivity that presents a error message, indicating that the user needs a network connection to start the inspection process. In the Appendix A, the figures A.4, A.5 and A.6 give a more detailed overview of the inspection starting process.

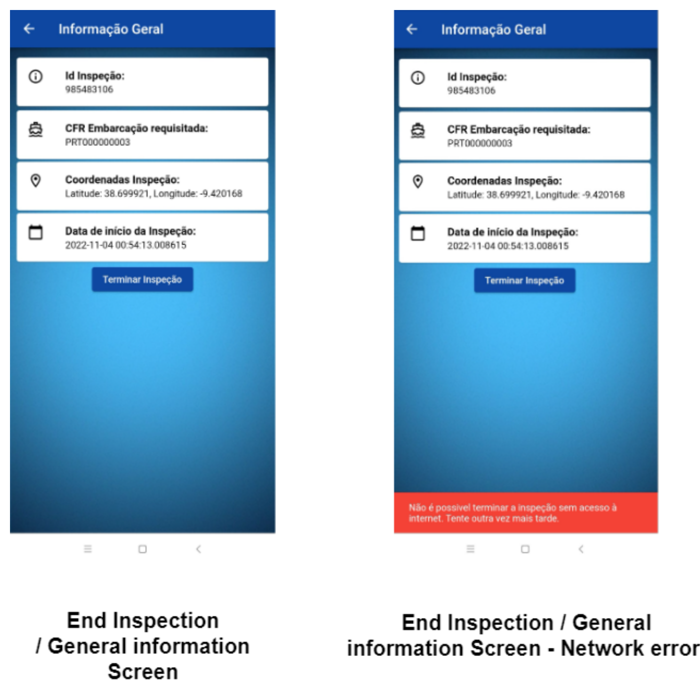


Figure 5.6: Application screens adaptability to the network connection regarding the ending of the inspection process. General information screen to end inspection and error message while ending inspection without network connectivity (from left to right)

The figure 5.6 shows the a general info screen, where the inspection can be ended. In the screens is also presented the adaptability of the inspection ending process, and the impossibility of ending the inspection without having a network connection. This limitation is indicated in the presented error message of the screen on the right.

With these three cases, where the most of the communications with the backend are performed, we present a mobile application capable of adapting its functionalities and capabilities to the state of the connection with the network, while warning the users that certain cases require a network connection and making it impossible to make network requests to the backend.

5.3.2 Usability

This section focuses on the mobile application usability, and the tests conducted to obtain relevant information and feedback relating to that aspect of the developed system.

Usability can be a difficult subject to test and study since it is more prone to subjective opinions, when compared to other tests and results obtainable by Unit tests, Integration tests, Functional tests and Performance tests. These types of tests are more focused on the experience of the user while using the application and the interactions that allow the users to reach their goals and complete the needed tasks. With that in mind, the UI (User Interface) can not be confusing, demanding, or cause stress to the users. Instead, the users tasks and workflows during the application usage should be fluid, intuitive and effortless, squashing the doubts that users could have by providing simple choice paths and by adhering to identifiable and more universally recognisable symbols and signs.

To guide the development and creation of more accessible, user-friendly, and intuitive products and UIs, a set of heuristics was proposed by Jakob Nielsen in [49]. Heuristics do not determine specific usability rules. Instead they are general principles, created through observations, that can be used and followed to help create a better user experience.

Therefore, the proposed Nielsen's heuristics, and the heuristics used to evaluate the mobile application usability, are:

- **H1. Visibility of system status**: – The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.
- **H2. Match between system and the real world**: – The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.
- **H3. User control and freedom**: Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.
- **H4. Consistency and standards**: Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.
- **H5. Error prevention**: Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.
- **H6. Recognition rather than recall**: The user should not have to remember the information to easily use the application. Instead the user should

be able to recognize common and identifiable symbols, terms, images, objects and actions.

- **H7. Flexibility and efficiency of use**: Accelerators, unseen by the novice user, may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.
- **H8. Aesthetic and minimalist design**: Only the relevant information should be displayed, increasing its visibility and creating a better point of focus for the user;
- **H9. Help users recognize, diagnose, and recover from errors**: Error messages should be presented in a language that the user will easily understand, with only the necessary information for the user to understand and handle the situation;
- **H10. Help and documentation**: Even though it is preferable that the system can be used without documentation, it some cases it may be necessary to provide easy to search and concrete documentation and information about the system and the user's task;

These heuristics serve as guidelines for the testers to provide feedback regarding various metrics, like: *ease to input data, screens response/load times, simplicity and ease of use, ui flexibility, memory load, operability, feedback and user guidance, consistency and fault tolerance.*

With these heuristics and metrics in mind, were conducted several tests with fourteen (14) users, where they were asked to perform the different tasks where the majority of the communications with the backend is done: ***Task 1*** - *authentication*, ***Task 2 -*** *documents verification and validation* and ***Task 3 and 4 -*** *starting and ending an inspection, respectively.* Users were also asked to complete tasks that do not require communication with the backend and that can be done either in online or offline mode, such as: ***Task 5 -*** *accessing the inspection general information*, ***Task 6 -*** *navigating to the vessels information screen and editing the information of a vessel under inspection* and ***Task 7 -*** *taking photos and videos for possible future evidences.* The application evaluations were conducted without outside intervention, only in the cases where it was needed to connect and disconnect the device to the network, thus presenting the users with different situations and creating a more comprehensive evaluation. The Table 5.1 below presents a summary and a compilation of the feedback obtained during the evaluations.

Feedback regarding the mobile application usability		
Task	**Feedback**	
Task 1	**Problem**	It is not possible to access ”forgot password” screen
	Description	The button ”forgot password” exists in the UI, but it is not functional
	Heuristic	H1. Visibility of system status
	Correction	Selecting the ”forgot password” button should present a different page where the user can request a new password
Task 3	**Problem**	Buttons positions are not consistent
	Description	The position of the buttons on the ”predictions/map screen” and the ”choose requisition port screen” are not consistent, one being in the bottom and the other in the top of the screen
	Heuristic	H4. Consistency and standards
	Correction	Button from the ”choose requisition port screen” should be in the bottom of the screen
Task 3	**Problem**	In the scrren ”choose requisition port”, users do not know all the ports available
	Description	During the start inspection process, when choosing a requisition port, the users do not have information about which ports are available
	Heuristic	H6. Recognition rather than recall and H7. Flexibility and efficiency of use
	Correction	The users should be presented with a list of the available ports, so they can choose faster and without having to remember the available options
Task 4	**Problem**	Lack of confirmation when ending an inspection
	Description	When the user ends an inspection, it is not presented with a confirmation box to confirm an important decision
	Heuristic	H5. Error prevention
	Correction	When the user tries to end an inspectio, it should be asked to confirm the decision, preventing misinputs
Task 6	**Problem**	Edit vessel button is dificult to find
	Description	While editing a vessel information, the edit button is dificult to find since the screen contains more information and is necessary to scroll to the bottom of the card
	Heuristic	H8. Aesthetic and minimalist design
	Correction	The edit and remove buttons of the entities should be placed on the top of the information card to facilitate its use
Task 7	**Problem**	The filter, to switch between photos and videos, is not intuitive/standardized
	Description	When changing between the photos and videos taken, the filter button is hard to comprehend without clicking on it
	Heuristic	H4. Consistency and standards
	Correction	The filter button icon should be changed and the default presentation of the photos and videos should be combined, initially presenting all the information to the user

Table 5.1: Table summarizing the testers feedback regarding the mobile application usability

The problems encountered during the mobile application usability evaluation focused on the **Tasks 1, 3, 4, 6 and 7**, since they were the most challenging tasks that involved multiple steps or had errors spotted by the users. The major problems found, focused on UI inconsistencies or lack of development on specific and smaller functionalities that could improve the user experience and save time to the user by preventing errors.

The ***Task 2 -*** *documents verification and validation* is not represented in the table since the users concluded that the correct UI elements were used, the steps to conclude the task were clear and easy to complete and that the correct feedback was given in the different situations of validating a document via QRCode. The ***Task 5 -*** *accessing the inspection general information* did not have feedback since it was a simple task and the users did not have difficulties completing it.

5.3.3 Performance

In this section, we finally focus on the third aspect of the tests, the performance of the application. The tested cases were again the cases mentioned in section 5.3.1, *authentication, verification and validation of documents, starting an inspection and also ending an inspection* since they are the most important cases, provide the most functionalities to the application and are the ones that are most likely to fail or take the most time, as they perform the communication with the backend of the system.

Therefore, we focus on the measuring the time it takes for each screen to load and become ready for the next user input. The measurements were performed 20 times for each screen and then an average was calculated from those values.

The first table, 5.2, shows information about the *authentication* screens and their load time. The data is divided in the four cases where the authentication can be done successfully, such as: using biometric or username/password while there is network connectivity and using biometric or username/password while there is no connection to the network. The load times were obtained using the tool described earlier, *Dart DevTools*, and focus on three different measurements: UI thread time (ms), Raster thread time (ms) and the Real time (ms), which is the closest approximation to the real experience of the user.

It is possible to observe similarities between the authentications using biometrics, since they both use the same authentication process just differentiating on a network connectivity check, which may be negligible but still observed through the differences in the *Raster Thread time*. The other types of authentication, username/password, also have similar values but in this case this similarity is due to the fact that the online authentication, achieved by communicating with the DGRM

Measured load times of authentication screens				
Network connectivity	**Yes**		**No**	
Type of Authentication	**Biometric**	**Username/password**	**Biometric**	**Username/password**
UI thread time (ms)	57.6	69.4*	57.1	64.2
Raster thread time (ms)	198.5	246.0*	207.9	265.8
Real time (ms)	60.9	76.0*	65.1	71.5

Table 5.2: Table containing the measured authentication screens load times

services, is not fully implemented due to security reasons, and thus is simulated locally saving time in network communications.

In table 5.3 was measured the time it took for the application to perform the validation of the licenses and update the UI elements. When comparing the different values between the measurements performed with network connectivity and without, the first tends to be a few milliseconds slower overall, since it verifies the acquired information in the backend systems increasing the communication time when compared with the local verification.

Measured load times of Licenses verification and validation screens				
Network connectivity	**Yes**		**No**	
License state	**Not valid**	**Valid**	**Not valid**	**Valid**
UI thread time (ms)	92.7	112.3	87.3	80.5
Raster thread time (ms)	532.8	645.9	477.6	493.6
Real time (ms)	119.2	162.4	104.2	123.4

Table 5.3: Table containing the measured licenses verification and validation screens load times

Lastly it is presented the Table 5.4 that provides the load times regarding the screens of the start and ending processes of an inspection.

Measured load times of starting and ending an inspection screens						
	Screens			**UI thread time (ms)**	**Raster thread time (ms)**	**Real time (ms)**
Start insp.	**main screen**	→	**init. insp. screen**	56.9	353.8	66.3
	init. insp. screen	→	**map vessels screen**	211.9	965.7	242.4
	map vessels screen	→	**choose req. screen**	141.3	575.3	191.3
	choose req. screen	→	**reqs. list screen**	162.4	672.9	156.8
	reqs. list screen	→	**inspector screen**	89.6	434.6	100.2
	inspector screen	→	**main screen**	136.5	514.2	157.3
			total	798,6	3 516,5	914,3

Finish insp.	**Screens**			**UI thread time (ms)**	**Raster thread time (ms)**	**Real time (ms)**
	end insp. screen	→	**main screen**	164.6	607.9	193.4

Table 5.4: Table containing the measured starting and ending inspection screens load times

Being a multi step process, the start of the inspection is composed of several screens with different information and that perform specific tasks needed to obtain the information to correctly start the inspection. Firstly a screen to initiate the inspection will be displayed A.4, in the next screen is shown a map with possible vessels to be inspected, where the user should pick at least one A.5, in the next step a screen asking the port of the inspector is displayed, where the user should input a port and choose an available requisition A.6, finally the user can add the the information of the inspector that will carry out the inspection.

The indicated flow, is represented in the table by the transitions from the screens on the left to the screens on the right, and their loading times. The slower screens transitions can be identified as the ones where the fetching of lists of information to the backend is done, such as loading the the map with the vessels predictions and loading the list of requisitions available. The Raster thread also has a more intensive load during these screens. The other screens, save information locally or make simple requests to the backend, having a smaller impact in comparison.

In the total, as the Raster thread has a value much higher than the UI thread, it leads to this case being the slower of the other studied cases while still presenting a loading screen values similar and in the same range as the ones presented in tables 5.2 and 5.3.

In contrast, the ending of an inspection is a much simpler process, only requiring a single screen. In this case the time is influenced by the amount of information gathered during the inspection, especially large files, such as recorded photos and videos, that may correspond to a large percentage of the overall collected data.

One final metric that was used to characterize the application's performance, is the average Frames Per Second (FPS). As one of the most simple and direct factors to be measured, has been widely used as a standard unit to describe the swiftness and smoothness of an application. Since it is a general metric that can be easily measured in most applications, it is usually used to compare the performance of two different application. In the *Flutter Documentation*[4], it is stated that "Flutter aims to provide 60 frames per second (FPS) performance." and to achieve this performance, it is required for each frame to be rendered approximately every 16ms.

In comparison with this goal, the values presented in the previous tables (real time values) did not achieve this performance, and thus also did not achieve the aimed 60 FPS. But since not all the the load times are as elevated as these values, with the presented values representing a small percentage of the overall frames, **the average FPS measured** from the 20 test runs was **52.6 FPS**, with the lowest measurement being 49 FPS and the highest being 55 FPS. This gives the user an

[4] `https://docs.flutter.dev/perf/ui-performance`

experience where most of the use of the application is fluid, but with frame drops leading to stutters in the transition between some screens, mainly screens regarding the process of starting the inspection where are measured the highest load screen values due to communications with the backend systems.

These results indicate that the application performance is not as expected, but some operations can't always be avoided or improved to render frames ate 16ms. Still, there are actions that can be taken to improve the performance, by:

- Minimizing layout passes caused by intrinsic operations
- Avoiding using constructors with a List of concrete children if most of the children are not visible on the screen to avoid the construction cost.
- Controlling build() cost, avoiding repetitive and costly work in build() methods since build() can be invoked frequently when ancestor widgets rebuild.

According with the presented metrics, 16ms of load time, the measured values of the different cases are higher and do not comply with that metric.

Considering that, there are still positive points to make. In [50], Ch. 5.5 Feedback, are discussed three important limits regarding response times, that focus on the 0.1s, 1s and 10s as reference marks to give the user a sensation of instant reaction, keep the user's flow of thought uninterrupted and keep the user's attention focused, respectively. With most of the measured values, of the real time variable, contained in the interval between 50ms and 150ms we can consider that the application experience will maintain the user engaged and focused on the current task, with the downside of not always allowing for a seemingly instantaneous responsive UI and creating a little stuttering in the screen load process.

Some exceptions that fall outside of the mentioned interval are: the screen transitions *init. insp. screen → map vessels screen* and *map vessels screen → choose req. screen*, regarding the start inspection process, and the screen transition *end insp. screen → main screen*, regarding the finish inspection process, and with following values: 242.4ms, 191.3ms and 193.4ms, still provide a user experience that keeps the user's flow of thought uninterrupted but do not provide a instant response that the user can perceive. With that in mind and with these exceptions values in the lower end of the 0.1s and 1s interval, the load times do not adhere to the metric presented by the Flutter team, 16ms, but still provide a adequate user experience that can be improved in the future.

Chapter 6

Conclusion and Future Work

In this project, the aim was to develop an architecture based microservice with the support of cloud computing that could be easily deployed and integrated in the DGRM private environment. The main goal of this cloud architecture is to assist inspectors during the inspections of vessels and vessel crews, that verify and validate fishing licenses and permits, by giving them access to relevant information about the details of the inspection subjects and the inspection itself.

To achieve that goal, the objectives of the overall architecture were: the creation of a scalable cloud backend architecture solution that supported the interactions between the users and the system through various APIs. This was achieved with the creation of seven microservices that expose REST APIs, that provide information about vessels, individual and collective persons, vessel licences, vessel requisitions, inspections and predictions fs(presented in the section 4.4.2), and their supporting, independent and dedicated databases that store only the relevant and necessary data for that specific microservice, increasing the data isolation and providing a more secure system. A secondary objective was also the research and discussion of the of possible implementations of vessel trajectory predictions models to aid unannounced inspections to new vessels, different from the main target of the initial inspection;

All the developed microservices and supporting databases are deployable in a Kubernetes environment, lowering the platform dependency and increasing the portability making them easy to deploy in most of public cloud providers infrastructures or even in a private cloud.

Other objective was the development of a frontend mobile application that allows the validation and verification of documents/information, in real time on board of vessels, and that can communicate with the backend system to retrieve the necessary information to the inspection. This was achieved with the creation of the mobile application, using the open source framework *Flutter* that allows for the development of multi-platform applications from a single codebase. The libraries/packages

used in the development of the application are all compatible with both Android and IOS, what also contributes to the compatibility with both Operating Systems. Currently the mobile application allows inspectors to: start inspections by choosing the vessel to be inspected and the vessel to be used in the transportation of the inspectors; have access to all the gathered information, during the inspection, and the information received from the backend, in the beginning of the inspection; scan documents/licenses using QRCode to verify if they are valid; edit or add new information about the vessel and crew members; save photos and videos related to the inspection for future evidence, in case its necessary; and change between multiple inspections;

Also regarding the mobile application, and as discussed in the Section 5.3, the application adaptability to the network and usability provide the user with ease of use and a good context about the status of the application, the connectivity to the network and the errors/successes of the user actions concerning these metrics on most of the tested cases, with the process of starting an inspection being one of the cases where most of the users difficulties occurred and thus being the case where the most changes and improvements can be made. Concerning the application performance, it does not achieve the goal of granting a 60 FPS experience, since the delays caused by the backend communications are higher than the average 16ms to load a frame. When dealing with network communications this type of delay is common, since the state of the network connection and the state of the backend system can contribute to the increase of this values. Nonetheless, the measured average of 52.6 FPS, offers a smooth and swift experience, mainly suffering with slowness in the specific case of starting an inspection.

6.1 Limitations

There are also some limitations, regarding the frontend application. Since we did not have full access to the DGRM backend, the user authentication service used to verify the authentication of users in the system was not available to us. So regarding the login and user authentication in the DGRM system, it was not implemented and were instead used a mock username and password ("testeU" and "testeP"). Despite this limitation, once the simulation of the authentication in th DGRM servers was made, it was possible to authenticate the user locally during a pre-determined time (7 days). When this time had passed, the user would need to re-authenticate with access to the authentication services.

Other limitation regarding the mobile application, is the fact that the synchronization of gathered data, during the inspection, between the inspectors is not com-

pletely implemented. When working with the plugin *flutter_nearby_connections* [1], that supports peer-to-peer connectivity and discovery of nearby devices for Android and IOS, it suffered from a high degree of inconsistency when searching for devices. In rare instances it would correctly detect and connect to other devices, but in the majority of the cases it would not detect other devices even if they were side by side and in the most advantageous possible position. This lack of consistency may be related to version differences between the devices and the different strategies used, but it was not possible to confirm this theory. Due to this unpredictability, this feature has some limitations and is not fully implemented.

The last limitation, concerns the developed databases of the backend system and was already mentioned in the previous section. In the development of the databases supporting the microservices, more specifically in the development of Oracle Databases 11.2 in the Kubernetes environment, the resources provided by the Oracle team, via Github, had some limitations while creating and running multiple instances on different ports, since it would return to the default ports even after the necessary changes were made. This limitations may arise since the Oracle databases are primarily built to be commercial and paid, and so the versions made available may not have some functionalities or be locked. Another reason may be due to the fact that the version used by the DGRM databases, version 11.2, is an older version that is still supported by Oracle but may not have the functionalities or the correct support to be deployed in a Kubernetes environment. To tackle this issue, we change to a more customizable option that gave more control over the specifications, MySQL.

6.2 Future Work

In respect to the implementation of this project and future work that can be done to improve the system, there are areas that can be built upon and continue to develop to create a better system, such as making the integration with the DGRM service(s) for user authentication so that the system can use the this information in a way that improves the usability of the mobile application.

Another part of the system that can be improved, is the microservice that handles the predictions of the vessels to be inspected. Currently it has a simple logic that mainly provides predictions based on the location of the inspectors. In the future, and with access to the DGRM services, VTS and AIS, that track the coastal area and vessels in real-time, it can be developed a more complex logic resorting to artificial intelligence predictive models. In the section 4.5.1 are presented and

[1] `https://pub.dev/packages/flutter_nearby_connections`

discussed some models that provide solid results on the prediction of vessel paths, based on the information provided by the AIS system, also used by DGRM. This approaches would require more and different data than the one that was provided for the development of this project and should also focus on gathering more relevant information and studying old inspection cases, creating patterns that could be used in the future.

Regarding the local data synchronization, between inspectors, this is other part of the system that can be enhanced. Currently, it is difficult to locally detect other devices (without resorting to a connection to the network) and so it can be improved and a more user friendly UI can be designed. This task entails researching questions such as: what are the cases in which the plugin used for the local communication, *flutter_nearby_connections*, can detect other devices, what are the limitations, what are other tools that could help achieve the same goal and other pertinent questions that could lead to a contribution for the plugin or a development of a new and more reliable resource.

Concerning the application performance and usability, some improvements can be made to provide the users with a more simple, consistent and polished experienced saving time and guiding the user in case of errors. The performance, whilst mainly providing the expected experience, suffers delays in the studied cases and thus can also be improved upon by minimizing layout passes caused by intrinsic operations and by controlling build() cost, avoiding repetitive and costly work in build() methods since build() can be invoked frequently when ancestor widgets rebuild.

The last possible addition to the system, is the integration with government systems, that are accessible to the DGRM and that deal with infractions or illegal actions. Since it already has the ability to store evidences (such as photos and videos about the inspected vessels) that could help in legal cases and that would help prove any illegality detected during the inspections, it could also be integrated with those services automating some parts of the process of alerting the correct entities.

Appendix A

Views regarding the frontend mobile application and its architecture

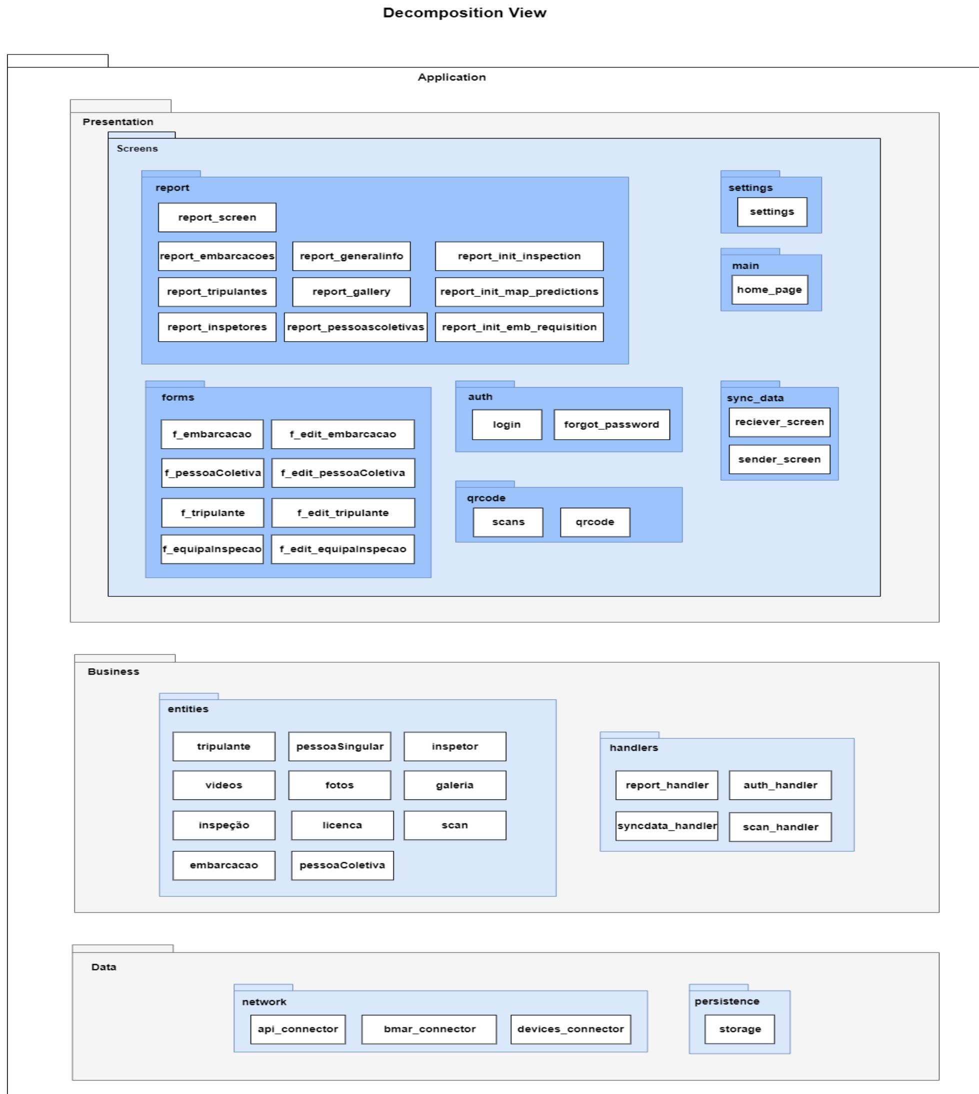


Figure A.1: Frontend Decomposition View Detailed

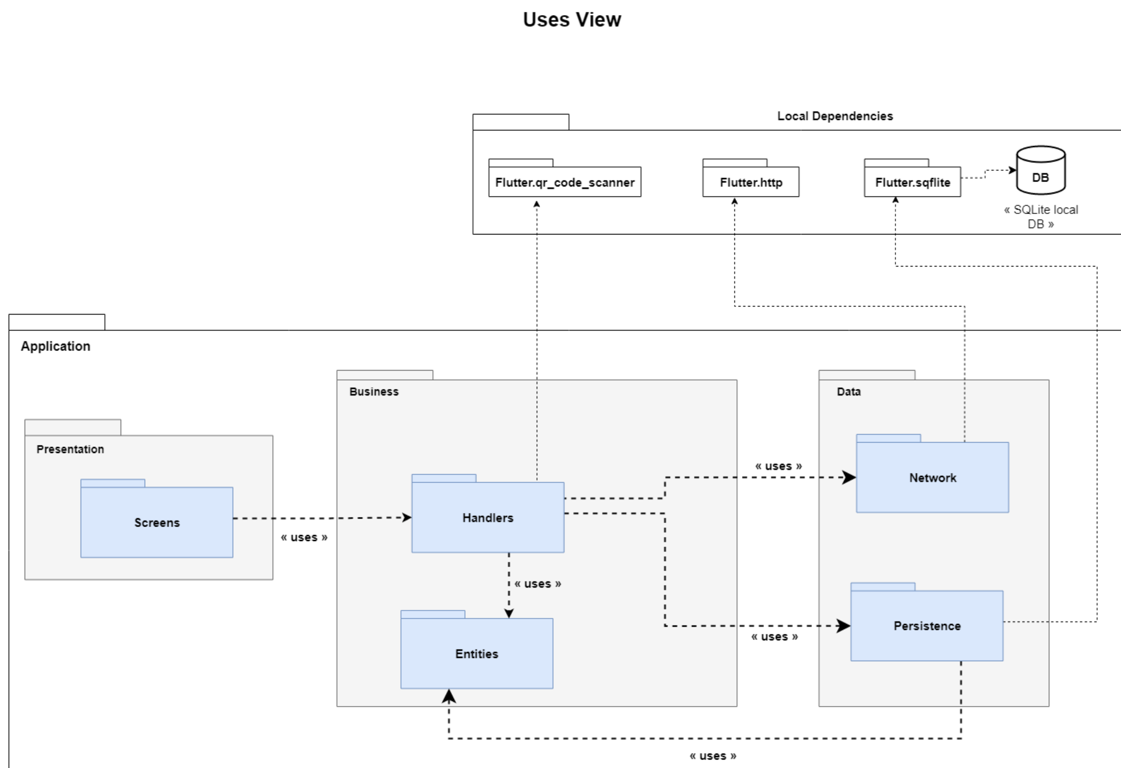


Figure A.2: Frontend Uses View

Figure A.3: Frontend Components and Connectors View

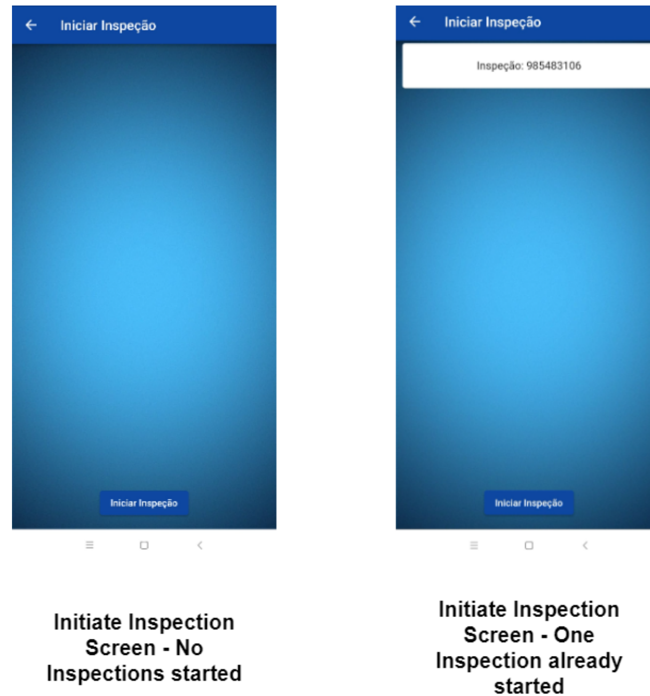


Figure A.4: Mobile application initiate inspection screen, without inspections started and with one inspection already started (from left to right)

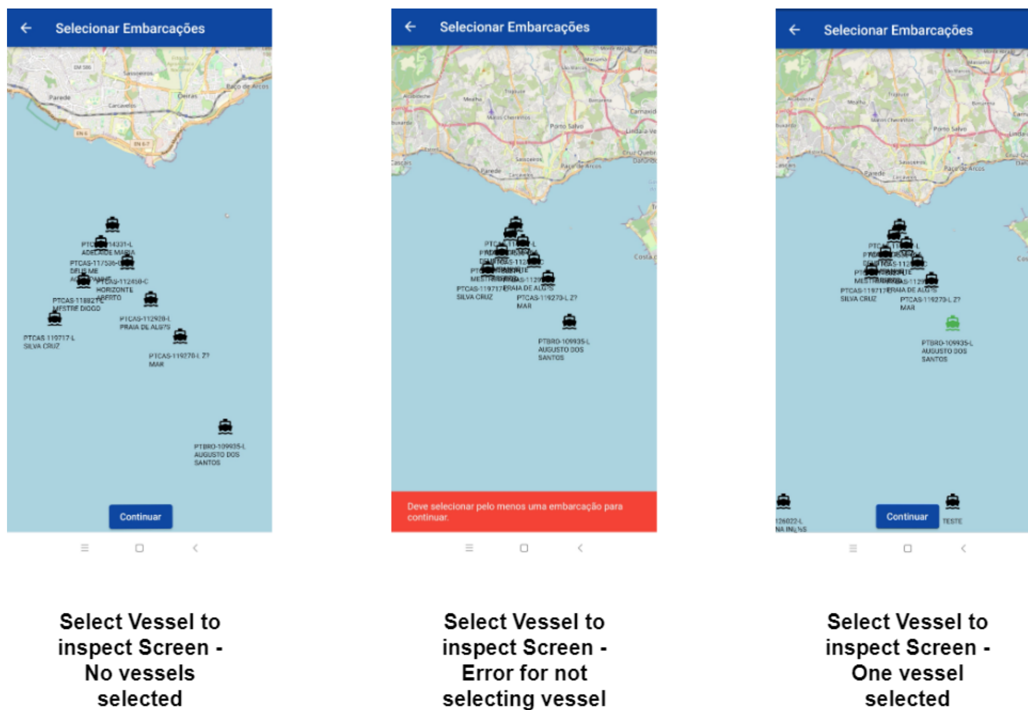


Figure A.5: Mobile application select vessel to inspect/predictions screen. Different examples of interactions: the default, presents error when no vessel is selected to inspect and when one vessel is selected (from left to right)

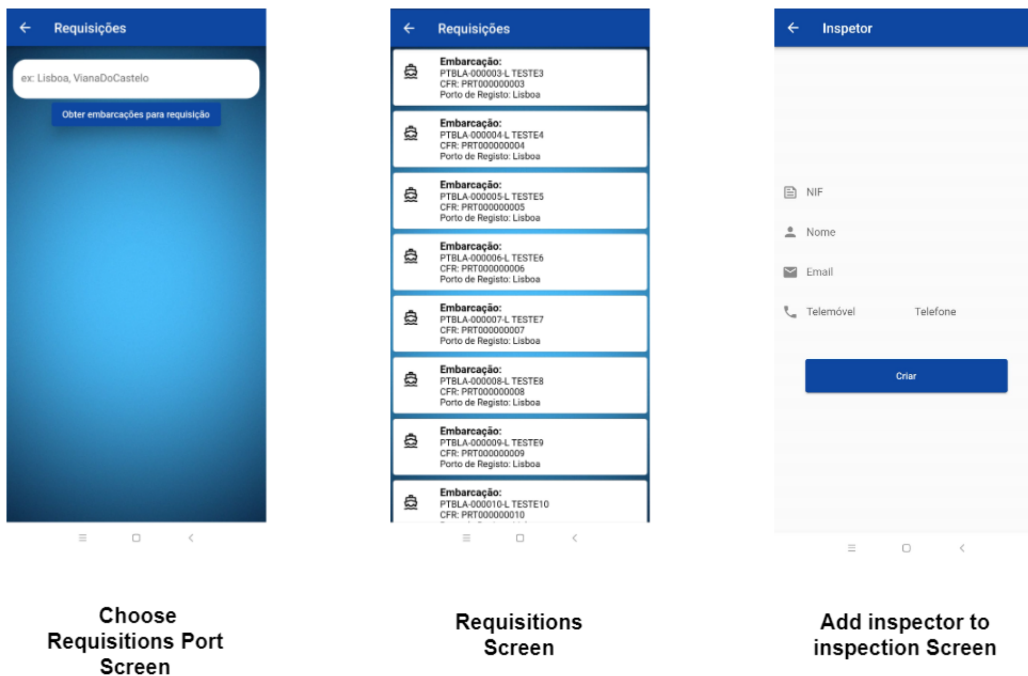


Figure A.6: Mobile application final screens of the initiate inspection process. Choose requisitions port screen, Requisitions screen and Add inspector screen (from left to right)

Appendix B

Tables regarding the backend MySQL databases

Table Inspeções DB			
Name	Type	Name	Type
*id	INT	lotacao_max_seguranca	INT
ano_licenca	INT	valido_desde	VARCHAR(100)
regiao	VARCHAR(150)	valido_ate	VARCHAR(100)
cfr	VARCHAR(150)	identificador_autorizacao	INT
nome	VARCHAR(150)	tem_area_internacional	VARCHAR(100)
conjunto_identificacao	VARCHAR(128)	identificador_grupo_arte	INT
titularnif	INT	grupo_de_arte	VARCHAR(150)
titular_da_licenca	VARCHAR(150)	identificador_arte	INT
tem_vms_instalado	VARCHAR(26)	arte	VARCHAR(150)
tem_dpe_instalado	VARCHAR(26)	identificador_sub_arte	VARCHAR(100)
tem_diario_de_pesca_em_papel	VARCHAR(26)	sub_arte	VARCHAR(100)
zona_de_navegacao	VARCHAR(100)	identificador_sub_sub_arte	INT
tipo_de_embarcacao	VARCHAR(150)	sub_sub_arte	VARCHAR(100)
segmento_pop	VARCHAR(100)	codigo_fao_arte	VARCHAR(100)
descodificacao	VARCHAR(150)	identificador_carateristicas	VARCHAR(100)
codigo_porto_registo	VARCHAR(100)	caracteristica	VARCHAR(300)
porto_registo	VARCHAR(100)	codigo_fao_area	VARCHAR(26)
indicativo_chamada	VARCHAR(100)	area	VARCHAR(100)
arqueacao_bruta_tab	FLOAT	especie	VARCHAR(100)
arqueacao_bruta_gt	FLOAT	grupo_de_especie	VARCHAR(100)
comprimento_fora_a_fora	FLOAT	autorizacao_condicao	VARCHAR(800)
comprimento_perpendicular	FLOAT	estado_licenca	VARCHAR(100)
potencia_motor	FLOAT	data_de_validade_da_licenca	VARCHAR(100)
lotacao_min_seguranca	INT	tipo_renovacao	VARCHAR(100)

Table B.1: Table from the Database that supports the Embarcação Microservice

Table Utilizadores	
Name	Type
*nif	INT
nome	VARCHAR(250)
email	VARCHAR(250)
telefone	INT
telemovel	INT
funcao	VARCHAR(20)

Table B.2: Table from the Database that supports the PessoaSingular Microservice

Table PessoasColetivas	
Name	Type
*nipc	INT
denomicacao_firma	VARCHAR(250)
sede	VARCHAR(250)
contacto	INT

Table B.3: Table from the Database that supports the PessoaColetiva Microservice

Table Licenças	
Name	Type
*id	INT
txt_conj_ident	VARCHAR(100)
num_cfr	VARCHAR(26)
nom_tp_documento	VARCHAR(200)
data_emissao	VARCHAR(100)
data_inicio_validade	VARCHAR(100)
data_validade	VARCHAR(100)
codigo_validacao	VARCHAR(100)
cod_estado_origem	VARCHAR(100)
data_do_estado_documento	VARCHAR(100)

Table B.4: Table from the Database that supports the Licença Microservice

Table Requisições	
Name	Type
*id	INT
cfr	VARCHAR(150)
conjunto_identificacao	VARCHAR(128)
porto_registo	VARCHAR(150)
disponivel	INT

Table B.5: Table from the Database that supports the Requisições Microservice

Table Inspeções	
Name	Type
*id	INT
cfr_embarcacao_requisitada	VARCHAR(150)
coordenadas	VARCHAR(100)
data	VARCHAR(200)

Table B.6: Table from the Database that supports the Inspeção Microservice

Table Inspeções_Embarcações	
Name	Type
*id	INT
*id_inspecao (FK)	INT
ano_licenca	INT
cfr	VARCHAR(150)
nome	VARCHAR(150)
conjunto_identificacao	VARCHAR(128)
titularnif	INT
titular_da_licenca	VARCHAR(150)
tem_vms_instalado	VARCHAR(26)
tem_dpe_instalado	VARCHAR(26)
tem_diario_de_pesca_em_papel	VARCHAR(26)
zona_de_navegacao	VARCHAR(100)
tipo_de_embarcacao	VARCHAR(150)
porto_registo	VARCHAR(100)
indicativo_chamada	VARCHAR(100)
arqueacao_bruta_tab	FLOAT
arqueacao_bruta_gt	FLOAT
comprimento_fora_a_fora	FLOAT
comprimento_perpendicular	FLOAT
potencia_motor	FLOAT
lotacao_min_seguranca	INT
lotacao_max_seguranca	INT

Table B.7: Table from the Database that supports the Inspeção Microservice, storing information about the inspected vessels

Table Inspeções_PessoasColetivas	
Name	Type
*nipc	INT
*id_inspecao (FK)	INT
denominacao_firma	VARCHAR(250)
sede	VARCHAR(250)
contacto	INT

Table B.8: Table from the Database that supports the Inspeção Microservice, storing information about the collective persons

Table Inspeções_Scans	
Name	Type
*id	INT
*id_inspecao (FK)	INT
utn	VARCHAR(100)
valido	VARCHAR(50)
hora	VARCHAR(250)

Table B.9: Table from the Database that supports the Inspeção Microservice, storing information about the scanned documents

Table Inspeções_Utilizadores	
Name	Type
*nif	INT
*id_inspecao (FK)	INT
nome	VARCHAR(250)
email	VARCHAR(250)
telefone	INT
telemovel	INT
funcao	VARCHAR(20)

Table B.10: Table from the Database that supports the Inspeção Microservice, storing information about the crew members and inspectors

Table Previsões	
Name	Type
*id	INT
lat	FLOAT
lon	FLOAT
conjunto_identificacao	VARCHAR(128)
cfr	VARCHAR(150)

Table B.11: Table from the Database that supports the Previsões Microservice

Acronyms

AIS Automatic Identification System. 1

API Application Programming Interface. 1

CC Cloud Computing. 1

CFR Community Fleet Register. 1

CI/CD Continuous Integration and Continuous Delivery and Deployment. 1

CNA Cloud Native Applications. 1

DB Database. 1

DDD Domain-Driven Design. 1

DGRM Direção Geral dos Recursos Marítimos. 1

EKS Amazon Elastic Kubernetes Service. 1

ESB Enterprise Service Bus. 1

FPS Frames Per Second. 1

GKE Google Kubernetes Engine. 1

IaaS Infrastructure as a Service. 1

MC Mobile Computing. 1

MCC Mobile Cloud Computing. 1

MDD Model-Driven Development. 1

PaaS Platform as a Service. 1

PV Persistent Volume. 1

PVC Persistent Volume Claim. 1

SaaS Software as a Service. 1

SOA Service Oriented Architecture. 1

VM Virtual Machine. 1

VTS Vessel Traffic Service. 1

Bibliography

[1] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. Microservice based architecture: Towards high-availability for stateful applications with kubernetes. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pages 176–185, 2019.

[2] Sareh Aghaei, Mohammad Ali Nematbakhsh, and Hadi Khosravi Farsani. Evolution of the world wide web: From web 1.0 to web 4.0. *International Journal of Web & Semantic Technology*, 3(1):1–10, 2012.

[3] Mohiuddin Ahmed, Abu Sina Md Raju Chowdhury, Mustaq Ahme, and Md Mahmudul Hasan Rafee. An advanced survey on cloud computing and state-of-the-art research issues. *International Journal of Computer Science Issues (IJCSI)*, 9(1):201, 2012.

[4] Abdulwahid Al Abdulwahid, Nathan Clarke, Ingo Stengel, Steven Furnell, and Christoph Reich. Continuous and transparent multimodal authentication: reviewing the state of the art. *Cluster Computing*, 19, 03 2016.

[5] Samaher Al-Janabi, Ibrahim Al-Shourbaji, Mohammad Shojafar, and Mohammed Abdelhag. Mobile cloud computing: Challenges and future research directions. In *2017 10th International Conference on Developments in eSystems Engineering (DeSE)*, pages 62–67, 2017.

[6] Mojtaba Alizadeh, Saeid Abolfazli, Mazdak Zamani, Sabariah Baharun, and Kouichi Sakurai. Authentication in mobile cloud computing: A survey. *Journal of Network and Computer Applications*, 61:59–80, 2016.

[7] Armbrust, Michael, Armando Fox, Armando, Griffith, Rean, Joseph, Anthony D, Randy Katz, Randy H, Andy Konwinski, Andrew, Gunho Lee, Gunho, Patterson, David A, Rabkin, Ariel, Stoica, and Matei. Above the clouds: A berkeley view of cloud computing. 01 2009.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53:50–58, 04 2010.

[9] Mohammad Sadegh Aslanpour, Sukhpal Singh Gill, and Adel Toosi. Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research. *Internet of Things*, 12:100273, 08 2020.

[10] P. M. Atkinson and A. R. L. Tatnall. Introduction neural networks in remote sensing. *International Journal of Remote Sensing*, 18(4):699–709, 1997.

[11] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[12] Sukadev Bhattiprolu, Eric W. Biederman, Serge Hallyn, and Daniel Lezcano. Virtual servers and checkpoint/restart in mainstream linux. *SIGOPS Oper. Syst. Rev.*, 42(5):104–113, jul 2008.

[13] Phil Blanco, Rick Kotermanski, and Paulo Merson. Evaluating a service-oriented architecture. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2007.

[14] Shady Boukhary and Eduardo Colmenares. A clean approach to flutter development through the flutter clean architecture package. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1115–1120, 2019.

[15] Antonio Bucchiarone, Nicola Dragoni, Schahram Dustdar, Stephan T. Larsen, and Manuel Mazzara. From monolithic to microservices: An experience report from the banking domain. *IEEE Software*, 35(3):50–55, 2018.

[16] J. Carolan and S. Gaede. Introduction to cloud computing architecture, sun microsystems inc. white paper. 2009.

[17] Susanta Nanda Tzi-cker Chiueh and Stony Brook. A survey on virtualization technologies. *Rpe Report*, 142, 2005.

[18] Caio Costa, João Vianney, Paulo Maia, and Francisco Oliveira. Sharding by hash partitioning - a database scalability pattern to achieve evenly sharded database clusters. volume 1, 04 2015.

[19] Vandna Dahiya and Sandeep Dalal. Fog computing: A review on integration of cloud computing and internet of things. In *2018 IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS)*, pages 1–6, 2018.

[20] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. 06 2016.

[21] Christian Esposito, Aniello Castiglione, and Kim-Kwang Raymond Choo. Challenges in delivering software in the cloud as microservices. *IEEE Cloud Computing*, 3(5):10–14, 2016.

[22] Martin Fowler and James Lewis. Microservices. 2014.

[23] Jon Friedman and Daniel V. Hoffman. Protecting data on mobile devices: A taxonomy of security threats to mobile computing and review of applicable defenses. *Inf. Knowl. Syst. Manag.*, 7(1,2):159–180, apr 2008.

[24] Simson L. Garfinkel and Harold Abelson. Architects of the information society: 35 years of the laboratory for computer science at mit. 1999.

[25] David Garlan. Software architecture. 2001.

[26] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, jun 2002.

[27] Timothy Grance and Wayne Jansen. Guidelines on security and privacy in public cloud computing, 2011-12-09 2011.

[28] Sabireen H. and Neelanarayanan V. A review on fog computing: Architecture, fog with iot, algorithms and research challenges. *ICT Express*, 7(2):162–176, 2021.

[29] Stephan HEATH. Methods and/or systems for an online and/or mobile privacy and/or security encryption technologies used in cloud computing with the combination of data mining and/or encryption of user's personal data and/or location data for marketing of internet posted promotions, social messaging or offers using multiple devices, browsers, operating systems, networks, fiber optic communications, multichannel platforms, U.S. Patent US10129211B2, Nov. 2018.

[30] Steven Ihde. From a monolith to microservices + rest: the evolution of linkedin's service architecture. `https://www.infoq.com/presentations/linkedin-microservices-urn/`, 2011-10-12. Accessed: 2022-07-16.

[31] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[32] Isam Mashhour Al Jawarneh, Paolo Bellavista, Filippo Bosi, Luca Foschini, Giuseppe Martuscelli, Rebecca Montanari, and Amedeo Palopoli. Container orchestration engines: A thorough functional and performance comparison. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, pages 1–6, 2019.

[33] R. E. Kalman. A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering*, 82(1):35–45, 03 1960.

[34] R. E. Kalman and R. S. Bucy. New Results in Linear Filtering and Prediction Theory. *Journal of Basic Engineering*, 83(1):95–108, 03 1961.

[35] R.E. Kalman. Contributions to the theory of optimal control, 1960.

[36] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. Edge computing: A survey. *Future Generation Computer Systems*, 97:219–235, 2019.

[37] Nane Kratzke and Peter-Christian Quint. Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. *Journal of Systems and Software*, 126:1–16, 2017.

[38] Kubernetes Team. Kubernetes components. `https://kubernetes.io/docs/concepts/overview/components/`. Accessed: 2022-07-28.

[39] Sarita Kumari. A research paper on cryptography encryption and compression techniques. *International Journal Of Engineering And Computer Science*, 04 2017.

[40] C. MacKenzie, Kenneth Laskey, Francis Mccabe, Peter Brown, and Rebekah Metz. Reference model for service oriented architecture 1.0. *Public Rev. Draft*, 2, 08 2006.

[41] Martin Fowler. Monolith first. `https://martinfowler.com/bliki/MonolithFirst.html`, 2015. Accessed: 2022-07-24.

[42] Tony Mauro. Adopting microservices at netflix: Lessons for architectural design, 2015.

[43] Mwathi D. Mbae, O. and E. Too. Secure cloud based approach for mobile devices user data. *Open Access Library Journal*, 09 2022.

[44] Peter Mell and Timothy Grance. The nist definition of cloud computing, 2011-09-28 2011.

[45] Yuki Minami, Atsushi Taniguchi, Taichi Kawabata, Norio Sakaida, and Katsuhiro Shimano. An architecture and implementation of automatic network slicing for microservices. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–4, 2018.

[46] Muhammad Baqer Mollah, Md. Abul Kalam Azad, and Athanasios Vasilakos. Security and privacy challenges in mobile cloud computing: Survey and way ahead. *Journal of Network and Computer Applications*, 84:38–54, 2017.

[47] Immaculée Josélyne Munezero, Doreen-Tuheirwe Mukasa, Benjamin Kanagwa, and Joseph Balikuddembe. Partitioning microservices: A domain engineering approach. In *2018 IEEE/ACM Symposium on Software Engineering in Africa (SEiA)*, pages 43–49, 2018.

[48] A.J. Nicholson, M.D. Corner, and B.D. Noble. Mobile device security using transient authentication. *IEEE Transactions on Mobile Computing*, 5(11):1489–1502, 2006.

[49] Jakob Nielsen. Usability heuristics for user interface design, 10.

[50] Jakob Nielsen. *Usability engineering*. Morgan Kaufmann, 1994.

[51] Nick Nikiforakis, Wouter Joosen, and Martin Johns. Abusing locality in shared web hosting. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[52] Talal H. Noor, Sherali Zeadally, Abdullah Alfazi, and Quan Z. Sheng. Mobile cloud computing: Challenges and future research directions. *Journal of Network and Computer Applications*, 115:70–85, 2018.

[53] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud container technologies: A state-of-the-art review. *IEEE Transactions on Cloud Computing*, PP:1–1, 05 2017.

[54] Francisco Ponce Mella, Gastón Márquez, and Hernán Astudillo. Migrating from monolithic architecture to microservices: A rapid review. 09 2019.

[55] Florian Rademacher, Jonas Sorgalla, and Sabine Sachweh. Challenges of domain-driven microservice design: A model-driven perspective. *IEEE Software*, 35(3):36–43, 2018.

[56] Chris Richardson. microservices.io. 2018.

[57] Anubha Sharma, Manoj Kumar, and Sonali Agarwal. A complete survey on software architectural styles and patterns. *Procedia Computer Science*, 70:16–28, 2015. Proceedings of the 4th International Conference on Eco-friendly Computing and Communication Systems.

[58] Saurabh Singh, Young-Sik Jeong, and Jong Hyuk Park. A survey on cloud computing security: Issues, threats, and solutions. *Journal of Network and Computer Applications*, 75:200–222, 2016.

[59] Staci Kramer. The biggest thing amazon got right: The platform. `https://old.gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/`, 2011-10-12.

[60] Yongfeng Suo, Wenke Chen, Christophe Claramunt, and Shenhua Yang. A ship trajectory prediction framework based on a recurrent neural network. *Sensors*, 20(18), 2020.

[61] D Taibi, V Lenarduzzi, and Claus Pahl. Architectural patterns for microservices: A systematic mapping study. Setúbal, 2018. SCITEPRESS.

[62] Davide Taibi and Valentina Lenarduzzi. On the definition of microservice bad smells. *IEEE Software*, vol 35, 05 2018.

[63] Davide Taibi, Valentina Lenarduzzi, Claus Pahl, and Andrea Janes. Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. pages 1–5, 05 2017.

[64] Huang Tang, Yong Yin, and Helong Shen. A model for vessel trajectory prediction based on long short-term memory neural network. *Journal of Marine Engineering & Technology*, 21(3):136–145, 2022.

[65] Tim Berners-Lee. The world wide web: A very short personal history. `https://www.w3.org/People/Berners-Lee/ShortHistory.html`, 1998/05/07. Accessed: 2022-07-14.

[66] Tim O'Reilly. What is web 2.0: Design patterns and business models for the next generation of software. `https://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html`, 09/30/2005. Accessed: 2022-07-14.

[67] Andrea Tosatto, Pietro Ruiu, and Antonio Attanasio. Container-based orchestration in cloud: State of the art and challenges. pages 70–75, 07 2015.

[68] Enmei Tu, Guanghao Zhang, Shangbo Mao, Lily Rachmawati, and Guang-Bin Huang. Modeling historical AIS data for vessel path prediction: A comprehensive treatment. *CoRR*, abs/2001.01592, 2020.

[69] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in a shared internet hosting platform. *ACM Trans. Internet Technol.*, 9(1), feb 2009.

[70] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, pages 583–590, 2015.

[71] Qi Zhang, Lu Cheng, and R. Boutaba. Cloud computing: State-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1:7–18, 05 2010.

[72] Olaf Zimmermann. Microservices tenets: Agile approach to service development and deployment. *Computer Science - Research and Development*, 32, 11 2016.