Instituto Politécnico do Cávado e do Ave
Escola Superior de Tecnologia

Arquitetura de Sistemas

Micromobilidade Elétrica

Pedro Cerqueira Nº14157

Janeiro de 2022

ESCOLA SUPERIOR DE TECNOLOGIA IPCA

Índice

ESCOLA SUPERIOR DE TECNOLOGIA IPCA

Lista de Figuras

ESCOLA SUPERIOR DE TECNOLOGIA IPCA

Lista de Tabelas

Introdução

O presente relatório foi elaborado no âmbito da unidade curricular Arquitetura de Sistemas do primeiro semestre do segundo ano letivo do Mestrado em Engenharia Informática (MEI) do Instituto Politécnico do Cávado e do Ave (IPCA). De modo a cimentar os conhecimentos adquiridos durante as aulas foi proposto a realização de um Trabalho Pratico cujo objetivo consiste no desenvolvimento de um sistema distribuído que permite agilizar o aluguer de viaturas elétricas no cenário de micromobilidade urbana, gerindo todo o processo e disponibilizando informação como a localização do veículo mais próximo e trajeto mais rápido. De um modo geral, o trabalho consistiu em arquitetar e desenvolver múltiplos *microservices* tendo em vista os requisitos do trabalho. Os *microservices* são uma abordagem de arquitetura e de organização no desenvolvimento de software, na qual este *software* consiste em pequenos serviços independentes que se comunicam usando APIs bem definidas. Cada um destes *microservices* irá ter a responsabilidade de gerir uma funcionalidade em concreto. Para o desenvolvimento deste trabalho foram utilizadas múltiplas tecnologias, como por exemplo, o NodeJs, o MongoDB, o Python, entre outras. Ao longo deste relatório, procura-se descrever ao leitor a forma como foram organizadas as tarefas durante a elaboração do trabalho. Procura-se enumerar as várias fases que deram origem ao resultado final, refletindo acerca das várias etapas e procedimentos seguidos. Apresenta-se ainda a estrutura final do trabalho, expondo a aplicação desenvolvida, ao longo do relatório, é feita uma reflexão acerca das dificuldades e a forma como foram ultrapassadas essas dificuldades.

Descrição dos Objetivos

Os objetivos delineados para este projeto, são:

1. Design de uma arquitetura de *microservices*;
2. Utilização de *microservices*: boas práticas, segurança, etc;
3. Interação com sistemas de base de dados NoSQL;
4. Utilização e contacto com IA para fazer reconhecimento de género e idade;
5. Utilização e contacto com serviços do tipo *webmapping*;

Requisitos Funcionais

Os requisitos funcionais encontram-se classificados pela metodologia *MoSCoW* que identifica os requisitos mais importantes até aos menos importantes. Cada tipo de requisito *(Must, Should, Could e Would)* representam diferentes significados face ao foco e importância que deverá ser levada em conta durante o processo de desenvolvimento. Assim sendo:

1. *Must*: classificados como requisitos mais críticos ou indispensáveis para o produto, pretende-se o total foco sobre estes tipos de requisitos durante o processo de desenvolvimento;
2. *Should*: são tão importantes como os requisitos Must, contudo, não são considerados tão priorizados;
3. *Could*: entende-se como os requisitos desejáveis, mas também não necessários;

4. *Would*: estes requisitos são itens menos críticos e com menor valor, podendo ser implementados ou não.

Na Tabela 1: Requisitos funcionais do sistema., são descritos todos os requisitos funcionais recolhidos.

ID	Ator	Requisito	Descrição	Prioridade
1	Sistema	Preçário	O sistema deverá permitir que as várias tipologias de veículos devem dispor de preçários diferentes.	Must
2	Sistema	Saldo	O sistema deverá verificar se o cliente tem saldo de modo a utilizar um veículo.	Must
3	Sistema	Carregamento do Saldo	O sistema deverá permitir ao cliente atualizar o seu saldo.	Must
4	Sistema	Critério de Cálculo	O sistema deverá ter um critério de calculo do período de utilização do veículo.	Must
5	Sistema	Perfis de Utilizador	O sistema deverá permitir que sejam criados diferentes tipos de utilizador.	Must
6	Gestor	Gestão de operação	O utilizador do tipo Gestor deverá conseguir fazer a	Must

			gestão de toda a operação.	
7	Gestor	Histórico da Situação dos Veículos	O utilizador do tipo Gestor deverá conseguir visualizar o histórico e resumo da utilização de todos os veículos.	Must
8	Gestor	Adicionar Veículo	O utilizador do tipo Gestor deverá conseguir adicionar novos veículos ao sistema.	Must
9	Gestor	Adicionar Tipo de Veículo	O utilizador do tipo Gestor deverá conseguir adicionar novos tipos de veículos ao sistema.	Must
10	Gestor	Definição dos preçários	O utilizador do tipo Gestor deverá conseguir atualizar os preçários dos tipos de veículos ao sistema.	Must
11	Cliente	Registo e Login	O utilizador do tipo Cliente deverá conseguir fazer o registo e login no sistema.	Must
12	Cliente	Validação da Idade	O utilizador do tipo Cliente deverá apenas conseguir alugar um veículo	Must

			se tiver uma idade superior a 16 anos.	
13	Cliente	Informação dos veículos	O utilizador do tipo Cliente deverá conseguir visualizar os veículos na sua área de localização.	Must
14	Cliente	Informação Sobre as Viagens	O utilizador do tipo Cliente deverá conseguir visualizar as informações sobre as viagens feitas.	Must
15	Cliente	Informações sobre o Perfil	O utilizador do tipo Cliente deverá conseguir visualizar as informações sobre o seu perfil.	Must
16	Cliente	Informações sobre o Preçário	O utilizador do tipo Cliente deverá conseguir visualizar as informações sobre os tipos de veículos e respetivo preçário sobre cada tipo de veículo.	Must
17	Cliente	Aluguer de Veículo	O utilizador do tipo Cliente deverá conseguir conseguir alugar um veículo(dependendo das restrições	Must

			acima mencionadas).	
18	Sistema	Atualização do Saldo do Cliente	O sistema deverá atualizar o saldo do cliente durante a viagem, ou seja, deverá ser debitar o custo a cada minuto da viagem que está a ser realizada.	Must
19	Sistema	Atualização do Carga do Veículo	O sistema deverá atualizar ao carga do veículo durante a viagem.	Must

Tabela 1 - Requisitos funcionais do sistema

Requisitos Não Funcionais

Ao contrário dos requisitos funcionais, os requisitos não funcionais não especificam nenhum serviço diretamente, funcionando como restrições aos serviços e funcionalidades do sistema. A escolha da metodologia a seguir para a identificação e organização dos requisitos não funcionais recaiu sobre FURPS+, uma metodologia muito usada devido à sua estrutura estar bem definida e organizada, e por os grupos de características serem os mais adequados para uma correta avaliação final deste projeto. A FURPS é um acrónimo que representa um modelo para classificar a qualidade dos atributos de um sistema, sendo que estes atributos não são mais do que os requisitos, quer funcionais quer não funcionais (Finkbine, 1996).

- Funcionalidade - requisitos funcionais, que descrevem os comportamentos e capacidades funcionais que o cliente espera da aplicação;

- Usabilidade - requisitos do ponto de vista do utilizador, ou seja, aspetos como os graus de eficácia e de eficiência do produto, se a interface tem uma estética aceitável e é consistente, e precisão e completude da documentação;
- Confiabilidade (*Reliability*) - requisitos referentes a disponibilidade do sistema (tempo que está disponível), exatidão nos cálculos e capacidade para recuperar de falhas;
- Performance - características como rendimento e diferentes tempos, de resposta, de recuperação, de iniciação e de encerramento;
- Suporte - aspetos sobre possibilidade de fazer testes ou não, adaptação, manutenção, compatibilidade, configuração, instalação, escalabilidade e localização.

Quanto à metodologia FURPS+, não é mais do que uma extensão da metodologia FURPS que ajuda a ter em conta mais algumas necessidades que os clientes possam ter:

- Requisitos de Design - especifica as opções para desenhar um sistema, como por exemplo, a especificação de uso de uma base de dados relacional;
- Requisitos de Implementação - obrigação ou não do uso de standards no desenvolvimento, uso de determinadas linguagens de programação, e existência de limitação nos recursos disponíveis;
- Requisitos de Interface - aspetos relacionados com interação: restrições de formatos ou itens externos que o sistema tem de usar;
- Requisitos Físicos - que tipo de hardware é necessário para suportar o produto final.

Na Tabela 2: Requisitos não funcionais do sistema., são descritos todos os requisitos não funcionais recolhidos.

ID	Área	Descrição	Categoria FURP +
RNF01	Perfomance	O número médio de erros realizados	Implementação

		pelos operadores não deve ultrapassar os dois por semana.	
RNF02	Performance	O sistema não deve demorar mais do que 30 segundos a dar resposta a um pedido.	Implementação
RNF03	Tecnologia	O sistema deve ser desenvolvido utilizando a tecnologia NodeJs para todos os seguintes serviços: *rentService, users, vehicles, routeprice* e *rentalrequestservic.*	Implementação
RNF04	Tecnologia	A persistência de dados deve ser feita por uma base de dados não relacional MongoDB.	Implementação
RNF05	*Perfomance*	O back-end deverá ser desenvolvido de modo a disponibilizar *endpoints* para serem consumidos pelos clientes.	Implementação

RNF07	*Perfomance*	O sistema deverá aceitar imagens do tipo .jpeg e .png.	Funcionalidade

Tabela 2 - Requisitos não funcionais do sistema

Descrição da Solução

O presente capítulo apresenta detalhes relacionados com o enquadramento e implementação da solução. Todas as funcionalidades implementadas e decisões tomadas são incluídas neste capítulo, de forma a fornecer ao leitor toda a linha de pensamento que conduziu ao design final e à implementação que originou a solução final descrita no presente documento.

Tecnologias Utilizadas

As tecnologias utilizadas neste projeto, foram:

- MongoDB: O MongoDB é um novo paradigma no que toca aos conceitos do que são as bases de dados tradicionais, pois este SGBD (sendo uma base de dados NoSQL) guarda todas as informações importantes num único documento, livre de esquemas onde possui ainda identificadores únicos, possibilitando a consulta dos documentos através de métodos avançados de agrupamento e filtragem permitindo com isto redundância e inconsistência [1];

- NodeJS: O Node.js é um interpretador de código JavaScript que funciona do lado do servidor. Esta plataforma permite aos programadores o desenvolvimento de aplicações em rede, em tempo real e de alta escalabilidade, de uma forma simples e rápida. O Node.js é baseado no interpretador V8 da Google [2];
- Docker: O Docker é uma plataforma Open Source de desenvolvimento, provisionamento e execução de aplicações que tem como base a linguagem de programação em Go da Google. Esta plataforma tem como principal objetivo facilitar a criação e gestão de ambientes isolados com recurso à tecnologia de *containers* [3];
- Portainer: O Portainer consiste ferramenta para ajudar na administração de *containers*, sejam eles locais, ou rodando em modo "*swarm*" [4];
- Python: A linguagem de programação Python é uma linguagem de alto nível (*VHLL - Very High Level Language*), interpretada e interativa, que foi criada pelo holandês Guido Van Rossum. Esta é uma linguagem orientada a objetos, modular, com uma sintaxe muito intuitiva e muito simples de aprender. Tal como o Perl, o código fonte do Python está disponível sob a licença GNU *General Public License* (GPL) [6];
- Git: O Git é um sistema gratuito, *open source*, rápido e distribuído para controlo de versões de software [7]. Repositório utilizado: https://github.com/a14157/AS;
- NodeRed - O Node-RED permite a programação através de uma interface gráfica bastante simples e intuitiva. Para desenvolver uma aplicação com recurso ao Node-RED, o programador apenas tem de ligar nós uns aos outros, em que cada um deles tem a suas próprias funções. Esta plataforma permite também a escrita de código mais elaborado utilizando JavaScript [8].

Diagrama de Componentes

A solução desenvolvida está divida nos seguintes componentes:

- Node-Red: Responsável por simular o comportamento dos veículos. Recebe um pedido para iniciar a viagem via protocolo MQTT e faz a simulação de todo o comportamento do veículo: atualização da carga do veículo, atualização do saldo do utilizador, verifica quando a viagem deve começar e terminar (dependendo da data inicial e final da viagem). Esta gestão, por norma, deverá ser feita do lado do *front-end* mas, dado que não existe um *front-end* neste projeto, a estratégia passou por fazer com que o próprio Node-Red simulasse alguma da interação do utilizador com a solução.
- MQTT Broker: Consiste num protocolo de envio de mensagens. Este protocolo foi utilizado para enviar e receber mensagens do NodeJs para o Node-Red.
- Rental Request Register: Este serviço foi desenvolvido em NodeJS e é responsável pela comunicação com entre o NodeJs e o Node-Red. A comunicação é feita a partir do protocolo MQTT.
- Geo Server API: Serviço de *webmapping* que permite obter a rota mais curta e a distância entre dois pontos, ou seja, o ponto de partida e o ponto de chegada definida pelo utilizador.
- RoutePrice: Este serviço foi desenvolvido em NodeJs e é responsável pela comunicação com o GeoServer API. Neste serviço, é verificado se os pontos de partida e chegada estão dentro da área disponível bem como, é definido um critério de cálculo do período de utilização: o tempo de viagem é calculado a partir da multiplicação da distância entre os dois pontos vezes dois (distância x 2). O número dois foi um número escolhido sem qualquer razão óbvia, apenas para chegar ao tempo de viagem. Por outro lado, neste serviço também é calculado o preço total da viagem. Este preço é obtido a partir da multiplicação do preço por hora do veículo (dependendo do tipo de veiculo, existe um preço diferente) vezes a distância entre os dois pontos.

- User: Este serviço foi desenvolvido em NodeJs e tem como objetivo gerir autenticações, utilizadores, tipos de utilizadores.
- Vehicle/Type of Vehicle: Este serviço foi desenvolvido em NodeJs e tem como objetivo gerir os veículos e tipos de veículos.
- Gender/Age Recogniton: Este serviço foi desenvolvido em Python e tem como objetivo permitir reconhecer a idade e género a partir de uma fotografia.
- Rent Service: Este serviço foi desenvolvido em NodeJs e tem como objetivo interligar todos os serviços explicados anteriormente(modelo em estrela). É a partir deste serviço que o utilizador utiliza todas as funcionalidades oferecidas pela aplicação. É também neste serviço que está a lógica de negócio relativa às permissões de utilizadores.

É importante referir que todos os serviços que são consumidos diretamente pela Rent Service, estão protegidos por uma chave. Esta funcionalidade foi incorporada de modo a aumentar a segurança de cada serviço. Na Figura 1, é apresentado o diagrama de componentes da solução descrita acima.

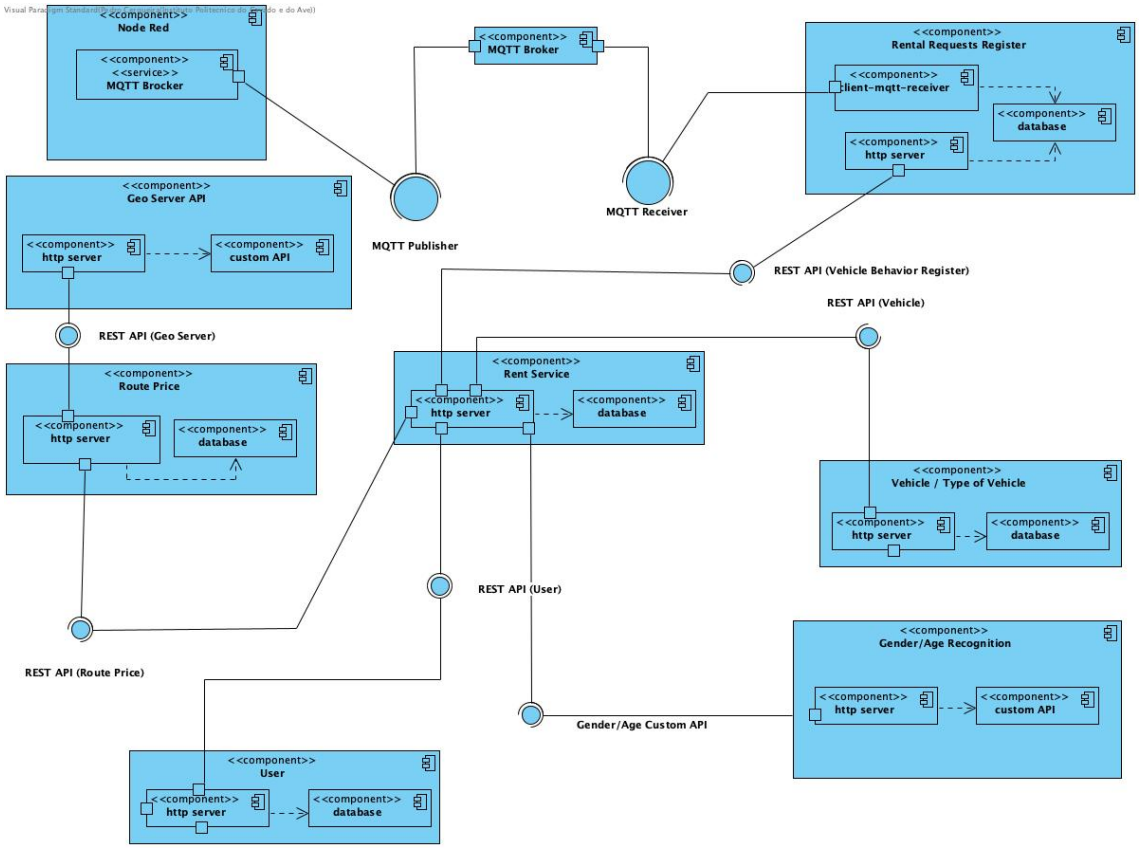

Figura 1 - Diagrama de Componentes da Solução

Arquitetura

A arquitetura da aplicação desenvolvida explica-se da seguinte forma:

- Cinco *containers (rentService, users, vehicles, routeprice* e *rentalrequestservice)* desenvolvidos em NodeJs, em que cada *container* é constituído por 4 camadas:
  - *Route*: Camada onde estão as definições dos *endpoints* da aplicação;
  - *Controller*: Camada que recebe os dados recebidos pelos *endpoints* da aplicação e que manda os dados solicitados pelos *endpoints*
  - *Service*: Camada que contém toda a logica da aplicação. Durante o desenvolvimento da aplicação teve-se o cuidado de desenvolver esta camada completamente independente do resto da aplicação, de modo a ser possível a sua reutilização;
  - *Model*: Camada que faz a gestão da definição e persistência dos dados.
- Um *container* desenvolvido em Python;
- Um *container* que representa o MqttBrocker;
- Um *container* que representa o NodeRed;
- Um *container* que representa o GeoServer;
- Um *container* que representa o phAdmin4;
- Um *container* que representa o PostGis;
- Um *container* que representa o MongoDD;
- Em todos os *containers* desenvolvidos em NodeJs, foram criados ficheiros *.env*. Em cada um destes ficheiros, foram definidos os URL's, portas e outros dados sensíveis.

Na Figura 2 e Figura 3, é possível ver um figura onde é explicado o comportamento dos *containers* desenvolvidos em NodeJs.

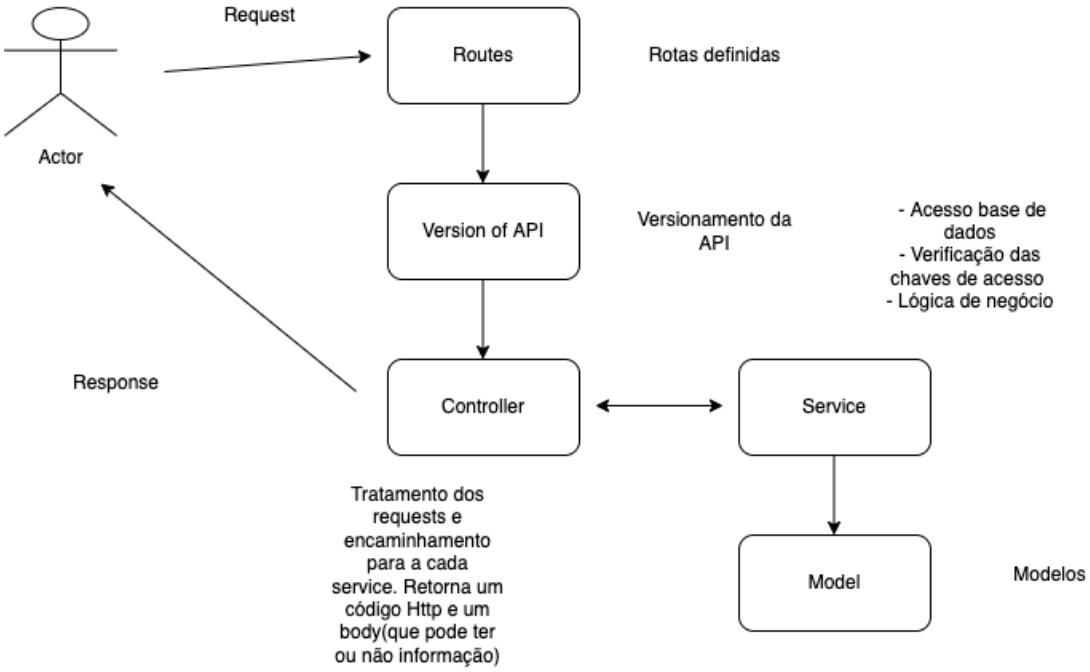

Figura 2 - Diagrama de Arquitetura

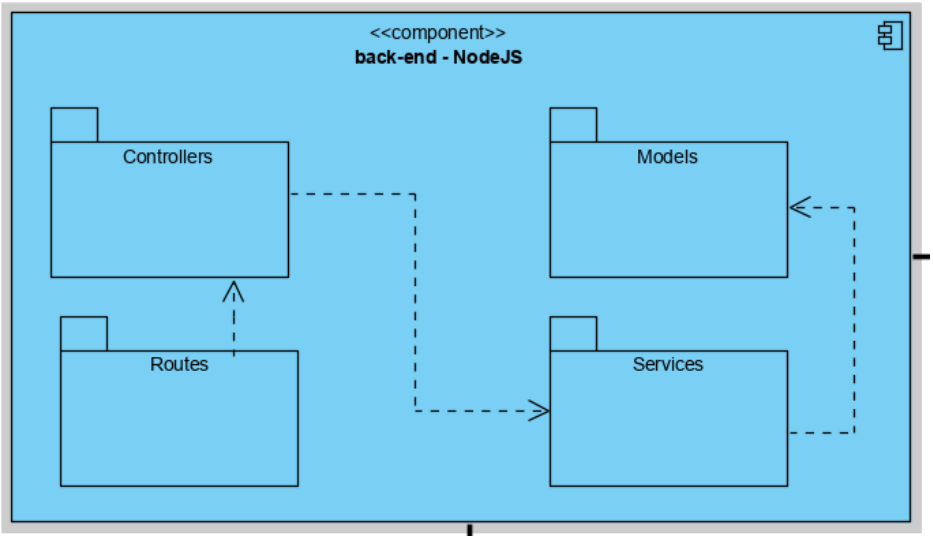

Figura 3 - Diagrama de componentes dos módulos de micro serviços do NodeJs

Documentação

Durante o desenvolvimento dos múltiplos serviços, foram desenvolvidos múltiplos módulos que permitem ao utilizador obter uma documentação detalhada sobre os *endpoints* implementados. Estes módulos foram implementados com a ajuda do *Swagger*. O *Swagger* é tipicamente utilizado para descrever, produzir, consumir, testar e visualizar uma API *RESTful*. Fornece várias ferramentas para gerar automaticamente a documentação [9]. Nos anexos deste relatório, podem ser encontrados os múltiplos URL's com com a documentação gerada pelo *Swagger*. Na pasta dos entregáveis deste projeto, também será entregue uma coleção do Postman com todos os *endpoints* disponíveis para testar e utilizar as funcionalidades desta aplicação.

Conclusões e Análise de Resultados

Concluído o projeto, pensa-se que foi desenvolvida uma solução com qualidade, que permite a iteração sobre o produto e adição de novos requisitos. Considera-se que os objetivos iniciais do projeto eram ambiciosos, mas realistas. Numa análise geral, o trabalho cumpriu todos os objetivos planeados inicialmente.

O planeamento inicial foi pensado de forma realista, tendo por base o fator tempo, que foi influenciado pela necessária pesquisa das tecnologias a usar e devida aprendizagem, tendo sido um pouco reduzido o tempo disponível.

Existiram também atrasos gerados devido à dificuldade da utilização do Node-Red e do Portainer. Ambas as tecnologias eram desconhecidas para o elemento do grupo de trabalho e, portanto, foi necessário despender mais tempo para obter o conhecimento necessário para utilizar estas tecnologias no âmbito do trabalho. Uma das maiores dificuldades encontradas, consistiu mesmo, na utilização e configuração do Portainer como plataforma de gestão dos *kubernetes*.

Relativamente ao trabalho futuro, apesar de a aplicação cumprir todos os objetivos planeados inicialmente, pensa-se que a implementação de um modulo que faça a gestão da cache e(como por exemplo o Redis[5])nos *endpoints* desenvolvidos seria uma das funcionalidades mais interessantes a desenvolver.

Toda a solução desenvolvida encontra-se devidamente documentada, recorrendo-se para isso a vários artefactos, em formato de texto e visual, para simplificar a compreensão do sistema desenvolvido. A implementação da solução, sendo bastante modular, permite a fácil modificação de qualquer componente sem causar grande perturbação nos componentes a que serve de dependência. Isto permite ainda a fácil introdução de novas funcionalidades ao sistema desenvolvido e a continuação do trabalho elaborado no futuro, caso seja preciso adicionar novas funcionalidades. A arquitetura e tecnologias utilizadas cumpriram os objetivos e criaram uma solução robusta e escalável.

Referências

[1] Mongo. (2022). MongoDB. Retrieved from https://docs.mongodb.com/

[2] NodeJS. (2022). Nodejs. Retrieved from https://nodejs.org/en/docs/

[3] Docker. (2022). Docker. Retrieved from https://docs.docker.com/compose/

[4] Portainer. (2022). Portainer. Retrieved from https://docs.portainer.io/

[5] Redis. (2022). Redis. Retrieved from https://redis.io/documentation

[6] Python. (2022). Python. Retrieved from https://docs.python.org/3/

[7] Git. (2022). Git. Retrieved from https://git-scm.com/doc

[8] Node-Red. (2022). Node-Red. Retrieved from https://nodered.org/docs/

[9] Swagger. (2022). Swagger. Retrieved from https://swagger.io/docs/

Anexos - Documentação

Links para a documentação de cada serviço:

Serviço *Python*: http://192.168.64.2:30048/api-docs/

Serviço *RentService*: http://192.168.64.2:30045/api-docs/

Serviço *Users*: http://192.168.64.2:30043/api-docs/

Serviço *Vehicles*: http://192.168.64.2:30044/api-docs/

Serviço *RoutePrice*: http://192.168.64.2:30045/api-docs/

Serviço *RentalRequestService*: http://192.168.64.2:30047/api-docs/