INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Archetype para Projetos Full-stack

TELMO JOSÉ GUEDES ALPOIM FERREIRA
Outubro de 2021

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Archetype para Projetos Full-stack

Telmo Ferreira

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Sistemas Computacionais

Orientador: Alexandre Bragança, Professor, DEI/ISEP

Júri:
Presidente:
Paulo Baltarejo Sousa, Professor, DEI/ISEP

Arguente:
Jacome Cunha, Professor, Universidade do Minho

Porto, 14 de outubro de 2021

Resumo

O conceito moderno de desenvolvimento ágil de *software* foi introduzido nos anos 90. Desde a sua conceção, este conceito tem dominado a indústria devido às vantagens que apresenta em relação a modelos anteriores de desenvolvimento (Hoda, Salleh e Grundy 2018). Contudo, para o desenvolvimento ágil ser eficazmente aplicado, a empresa tem de possuir um fluxo de trabalho bem otimizado.

O começo de um novo projeto de *software* é, normalmente, um momento onde as equipas perdem algum tempo. Para além da definição e estruturação de todos os conceitos que envolvem o projeto, é também necessário tratar de todas as "burocracias técnicas". Essas burocracias incluem tarefas como criar os repositórios onde o código vai estar alojado e onde será feito o controlo de versões, definir as estruturas de ficheiros para esses repositórios, entre muitas outras. Esse processo exige algum cuidado, uma vez que envolve algumas decisões que vão acompanhar o projeto daí para a frente. Contudo, é também, habitualmente, um processo maioritariamente manual, onde erros podem ser cometidos.

A solução descrita neste documento tem como objetivo ajudar as equipas nessas fases de início de projetos, gerando automaticamente um esqueleto para todo o projeto, com *templates* de código para as componentes *back-end* e *front-end*, assim como *pipelines* e mecanismos de *Continuous Integration, Continuous Delivery* (CI/CD) para tornar o projeto *deploy-ready*. Assim, esta solução iria proporcionar às equipas uma forma automática de realizar várias tarefas que são normalmente efetuadas de forma manual, dando-lhes assim a oportunidade de utilizarem esse tempo para começarem imediatamente o desenvolvimento dos projetos e evitando potenciais lapsos provenientes de erro humano.

Este projeto de tese foi realizado para a empresa Critical TechWorks, que é uma *joint venture* entre a Critical Software e o Grupo BMW. A solução aqui descrita foi idealizada para a área de *Infotainment & Interactivity Services*, cujo objetivo é desenvolver serviços para o sistema de infoentretenimento dos carros da BMW.

Palavras-chave: *Archetype*, *Scaffolding*, *Full-stack*, Automação, Desenvolvimento, *Software*

Abstract

The modern concept of agile software development was introduced in the 1990s. Since its conception, this concept has dominated the industry due to the advantages it presents over previous development models. However, for agile development to be effectively applied, the company must have a well-optimized workflow.

The start of a new software project is usually a time when teams lose some time. In addition to the definition and structuring of all the concepts that involve the project, it is also necessary to address all the "technical bureaucracies". These bureaucracies include tasks such as creating the repositories where the code will be hosted and where version control will be done, defining the file structures for these repositories, among many others. This process requires some caution since it involves some decisions that will accompany the project from that moment forward. However, it is also usually a process that's mostly manual, where mistakes can be made.

The solution described in this document aims to help teams in these phases of project initiation, by automatically generating a skeleton for the entire project, with code templates for the back-end and front-end components, as well as pipelines and Continuous Integration, Continuous Delivery (CI/CD) mechanisms to make the project deploy-ready. Thus, this solution would provide teams with an automatic way to carry out various tasks that are normally done manually, thus giving them the opportunity to use that time to start the project development immediately and avoid potential lapses resulting from human error.

This thesis project was carried out for the company Critical TechWorks, which is a joint venture between Critical Software and the BMW Group. The solution described here was designed for the Infotainment & Interactivity Services area, whose objective is to develop services for the infotainment system of BMW cars.

Agradecimentos

Em primeiro lugar, gostaria de agradecer ao Engenheiro Alexandre Bragança, o meu orientador de tese, por todo o auxílio e acompanhamento ao longo deste projeto.

Gostaria de agradecer também à Critical TechWorks pela oportunidade deste projeto e por todas as capacidades técnicas e pessoais que me instruíram.

Agradeço ao Instituto Superior de Engenharia do Porto e a todos os seus docentes que me acompanharam nesta minha jornada, desde a licenciatura até ao mestrado, por toda a qualidade de ensino e por me ajudarem a chegar a este ponto.

Por fim, um agradecimento especial à minha namorada, amigos e pais por todo o apoio ao longo destes anos. Esta viagem não foi fácil, mas com vocês ao meu lado era impossível falhar.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Código

Lista de Acrónimos

AHP	*Analytic Hierarchy Process*.
AMI	Amazon *Machine Image*.
API	*Application Programming Interface*.
ASGI	*Asynchronous Server Gateway Interface*.
AWS	Amazon Web Services.
CERN	Conseil Européen pour la Recherche Nucléaire.
CI/CD	*Continuous Integration, Continuous Delivery*.
CRUD	*Create, Read, Update and Delete*.
CSS	*Cascading Style Sheets*.
CWI	Centrum Wiskunde Informatica.
DOM	*Document Object Model*.
E2E	*End-to-end*.
EC2	*Elastic Compute Cloud*.
ECR	*Elastic Container Registry*.
EE	*Enterprise Edition*.
EKS	*Elastic Kubernetes Service*.
FEI	*Front End of Innovation*.
FFE	*Fuzzy Front End*.
GIS	*Geographic Information Systems*.
GUI	*Graphical User Interface*.
HCL	Hashicorp *Configuration Language*.
HOQ	*House of Quality*.
HTML	*Hypertext Markup Language*.
HTTP	*Hypertext Transfer Protocol*.
I/O	*Input/output*.
IaaS	*Infrastructure as a Service*.
IC	Índice de Consistência.
IDE	*Integrated Development Environment*.
IP	*Internet Protocol*.
JIT	*Just-In-Time*.
JSON	JavaScript *Object Notation*.
JSP	Jakarta *Server Pages*.
JVM	Java *Virtual Machine*.

MP	MicroProfile.
MVC	*Model-View-Controller*.
NCD	*New Concept Development*.
NLB	*Network Load Balancer*.
NPD	*New Product Development*.
NPM	*Node Package Manager*.
NPPD	*New Product and Process Development*.
PaaS	*Platform as a Service*.
PoC	*Proof of Concept*.
QA	*Quality Assurance*.
QFD	*Quality Function Deployment*.
RC	Razão de Consistência.
REST	*Representational State Transfer*.
SaaS	*Software as a Service*.
SE	*Standard Edition*.
SPA	*Single-Page Application*.
SSH	*Secure Socket Shell*.
SUS	*System Usability Scale*.
TI	Tecnologia da Informação.
UI	*User Interface*.
UML	*Unified Modeling Language*.
URI	*Uniform Resource Identifier*.
URL	*Uniform Resource Locator*.
VPC	*Virtual Private Cloud*.
WSGI	*Web Server Gateway Interface*.
YAML	YAML *Ain't Markup Language*.

Capítulo 1

Introdução

Neste capítulo é feita uma introdução do tema principal desta dissertação, e dos seus principais aspetos.

O capítulo encontra-se dividido em cinco secções. A primeira secção faz uma breve apresentação da empresa onde este projeto de tese foi realizado, a Critical TechWorks. A segunda secção faz uma contextualização resumida do meio em que se vai inserir a solução proposta neste documento. A terceira secção apresenta o problema que existe atualmente, e que esta solução visa resolver. Na quarta secção são discutidos os objetivos principais da solução. Por fim, na quinta e última secção do capítulo, é mostrada a estrutura deste documento, com uma breve descrição de cada capítulo.

1.1 Empresa

A Critical TechWorks, tal como mencionado no Resumo, é uma *joint venture* entre a Critical Software e o Grupo BMW (BMW 2020). Fundada em 2018, a Critical TechWorks nasceu da recente motivação da BMW de dar mais foco ao *software* e à qualidade dos serviços oferecidos nos seus veículos.

Na estratégia delineada pelo Grupo BMW para os próximos 100 anos, três áreas tiveram particular destaque: carros elétricos, condução autónoma e *software*. Isso fez com que a BMW decidisse procurar um parceiro tecnológico para os auxiliar a atingir estes objetivos. Assim, a BMW criou uma lista com várias empresas de todo o mundo, optando no final por selecionar a empresa portuguesa Critical Software como o seu parceiro tecnológico.

Contudo, a quantidade de projetos requisitados pela BMW era tão vasta que a Critical Software não tinha recursos para os executar. Assim, surgiu então a motivação para a criação de uma nova empresa exclusivamente para a realização destes projetos, de forma a conseguir dar resposta às necessidades da BMW. Nasceu, então, a Critical TechWorks.

Juntando a perícia da indústria automóvel da BMW com a cultura de trabalho e metodologias ágeis da Critical Software, a Critical TechWorks tinha reunidas todas as ferramentas para conseguir solucionar os desafios tecnológicos do carro do futuro.

1.2 Contexto

Num mercado tão competitivo como a indústria automóvel, é importante que os fabricantes ofereçam aos seus clientes o máximo de valor possível em cada veículo. Numa era digital como a que se vive atualmente, em que a tecnologia cada vez se encontra mais presente no dia-a-dia da população geral, este está a tornar-se um fator decisivo na escolha de um carro, especialmente

no mercado *premium*. Enquanto que os compradores de automóveis no mercado *premium* ficavam tradicionalmente satisfeitos com carros rápidos, seguros e luxuosos, nos dias de hoje é cada vez mais exigido que os seus carros sejam um equipamento inteligente, acompanhando assim os seus *smartphones* ou computadores (Xu e Liu 2018).

Na Coreia do Sul, por exemplo, que é atualmente um dos cinco maiores países fabricantes de automóveis, tem-se verificado nas últimas duas décadas que a cota de mercado de carros domésticos tem vindo a diminuir e a de carros importados tem vindo a aumentar. Comparando as marcas mais vendidas de cada setor, a Hyundai para carros domésticos e a BMW para carros importados, e recorrendo a metodologias de comparação tecnológica, Jun e Park (2016) teorizam que a forte aposta tecnológica da BMW tem sido uma das razões para a sua crescente cota de mercado, e devido a isso estimam que esta continue a aumentar nos próximos anos.

Para se manter relevante no mercado *premium*, e para se manter competitiva face à forte concorrência de marcas como a Mercedes ou a Audi, a BMW tem de ser eficaz na sua evolução tecnológica. Assim, os serviços desenvolvidos pela Critical TechWorks tornam-se fundamentais neste desafio.

A Critical TechWorks, utilizando os valores originados da Critical Software, usa metodologias ágeis de desenvolvimento de *software* nos seus projetos, proporcionando assim mais valor aos seus clientes numa menor quantidade de tempo. Na Critical TechWorks, particularmente na área de *Infotainment & Interactivity Services*, o método ágil utilizado é o Scrum. Desenvolvido nos anos 90, o Scrum é uma *framework* que ajuda pessoas, equipas e organizações a gerar valor através de soluções adaptadas a problemas complexos (Schwaber e Sutherland 2011). Os trabalhos e desenvolvimentos são iterados em *sprints*, sendo que cada *sprint* deve acrescentar algum valor ao produto.

1.3 Problema

Sendo o Scrum uma metodologia ágil focada em gerar valor de forma iterativa e célere, é importante que as equipas possuam um fluxo de trabalho otimizado em todas as fases do ciclo de vida de um projeto de *software*. Uma fase que é notoriamente mais lenta que as restantes é o início de um novo projeto.

Seguem-se alguns exemplos de tarefas que são efetuadas nesta fase:

- **Criar os repositórios**

  No começo de um novo projeto de software, é necessário criarem-se os vários repositórios do projeto. Para um projeto *full-stack*, normalmente irão existir pelo menos dois repositórios, um para a componente *back-end* e outro para a componente *front-end*. Adicionalmente, poderá também existir um repositório para alojar o código da infraestrutura necessária.

  Estes repositórios irão ser onde os diversos desenvolvimentos serão feitos e onde será também realizado o controlo de versões do código. Tipicamente, os repositórios irão estar alojados em serviços próprios para o efeito, como o GitHub ou o Bitbucket.

- **Definir a estrutura de ficheiros**

  Após a criação dos repositórios, é então necessário definir a estrutura dos ficheiros do projeto. Um projeto de *software*, particularmente se for um projeto empresarial, pode tornar-se bastante complexo com o tempo. Assim, é essencial que a equipa desenvolva o código de forma organizada e estruturada.

  Dependendo da tipologia do projeto (isto é, se é *front-end* ou *back-end*, por exemplo) e das tecnologias utilizadas, existem várias estruturas que são consideradas boas práticas. Ou então, a equipa poderá definir uma nova estrutura que lhe seja mais conveniente. Independentemente da estrutura escolhida, esta tarefa é algo que habitualmente requere o envolvimento de toda a equipa, uma vez que terá alguns impactos sobre os desenvolvimentos futuros.

- **Criar a infraestrutura**

  Qualquer projeto de software necessita de uma infraestrutura que o suporte. Esta infraestrutura pode ter várias formas, dependendo das necessidades do projeto: por exemplo, pode conter máquinas virtuais, armazenamento de *containers* ou uma base de dados.

  A infraestrutura pode ser criada manualmente no fornecedor *cloud* respetivo, ou então pode ser criada com código, utilizando ferramentas que permitam infraestrutura como código. De qualquer forma, esta fase por si só também envolve algumas decisões, como por exemplo a configuração da infraestrutura.

  Outro possível atraso associado a esta fase poderá estar também relacionado com os conhecimentos de infraestrutura da equipa. Caso esta tarefa da criação inicial da infraestrutura fique a cabo de um elemento com poucos conhecimentos sobre este domínio, e se este não tiver outros exemplos por onde se basear, a tarefa iria demorar mais tempo do que o esperado, uma vez que este elemento iria ter que primeiro adquirir os conhecimentos que necessita.

- **Configurar os mecanismos CI/CD**

  A utilização de mecanismos de CI/CD tem benefícios no processo de entrega ágil de software, tornando o processo mais eficiente e aumentando a produtividade do sistema (Arachchi e Perera 2018). A execução de testes, o *deployment* dos serviços e a criação e configuração da infraestrutura são tarefas que podem ser automatizadas com *pipelines* CI/CD.

  No início de um novo projeto é necessário criar e configurar todos estes processos CI/CD, algo que requere algum tempo e esforço por parte da equipa.

Estas tarefas são meramente alguns exemplos. Contudo, dependendo da complexidade e da dimensão do projeto em si, poderão eventualmente existir ainda mais.

A principal conclusão é que se trata de uma fase burocrática que acaba por exigir algum tempo da equipa, consistindo maioritariamente em tarefas manuais. A automação destas tarefas iria acelerar o desenvolvimento inicial do projeto, podendo dessa forma a equipa começar a produzir valor mais rapidamente para os seus clientes.

1.4 Objetivos

Um *archetype* (ou arquétipo, em português) é um conceito que representa um primeiro modelo ou padrão, a partir do qual outras coisas são copiadas ou baseadas. Dessa forma, este projeto

consiste no desenvolvimento de uma solução para a criação de um *template* inicial de um projeto *full-stack*, para que uma equipa tenha imediatamente um ponto de partida já desenvolvido no início de um novo projeto.

Por outras palavras, o objetivo principal deste projeto é desenvolver uma solução que consiga dar resposta aos problemas levantados na secção anterior. Nomeadamente, uma solução que consiga automatizar as tarefas mais habituais no início de um novo projeto empresarial, tarefas essas que frequentemente são executadas de forma manual, impedindo assim a ocorrência de erros e poupando tempo e esforço à equipa de desenvolvimento.

Os requisitos específicos desta solução, tanto os funcionais como os não funcionais, são analisados em mais detalhe na secção 3.1 deste documento.

1.5 Estrutura do Documento

O presente documento está dividido em sete capítulos. A sua organização está realizada da seguinte forma:

- **Capítulo 1: Introdução**

  Este capítulo é a introdução do relatório. Aqui faz-se uma descrição da empresa onde decorreu o estágio, a contextualização e a descrição do problema a ser solucionado, bem como os objetivos da solução desenvolvida.

- **Capítulo 2: Contexto e Estado da Arte**

  Neste capítulo é aprofundado o contexto da empresa onde a solução será inserida. Depois, é feita uma breve apresentação de outros trabalhos e artigos relacionados com o tema. Para além disso, é realizada uma análise de ferramentas e tecnologias que foram consideradas para o desenvolvimento da solução final, ou que serviram de inspiração para o seu desenvolvimento.

- **Capítulo 3: Análise**

  Neste capítulo é efetuada uma análise a vários aspetos da solução, nomeadamente uma análise dos requisitos identificados para a solução, uma análise de valor e, finalmente, uma análise das alternativas identificadas no capítulo 2.

- **Capítulo 4: *Design***

  Este capítulo aborda o *design* da solução desenvolvida, fornecendo uma explicação detalhada do fluxo principal da solução, falando sobre a sua arquitetura e explorando a especificação do *back-end*.

- **Capítulo 5: Implementação**

  Este capítulo explora em mais detalhe o processo de implementação da solução desenvolvida, descrevendo as fases e os conceitos mais importantes.

- **Capítulo 6: Experimentação e Avaliação**

  Este capítulo aborda os aspetos da solução que foram avaliados e a forma como essa avaliação foi feita. Para isso foram definidas as hipóteses de investigação, a metodologia de avaliação e os resultados dos métodos de avaliação.

- **Capítulo 7: Conclusão**

  Este capítulo finaliza esta dissertação, fazendo um resumo do trabalho realizado, dando alguns pontos que foram identificados como melhorias para um possível trabalho futuro, assim como uma apreciação final de todo este projeto.

Capítulo 2

Contexto e Estado da Arte

Este capítulo explora um pouco mais o contexto do na Critical TechWorks, explora outros trabalhos relacionados com o problema a ser solucionado, e aborda várias ferramentas e tecnologias que foram consideradas para este projeto. Devido à quantidade de tecnologias necessárias para cumprir todos os requisitos desta solução, neste capítulo apenas é realizada uma descrição relativamente básica destas várias ferramentas e tecnologias, não indo por isso a detalhes muito específicos ou exaustivos sobre todas as suas funcionalidades.

O capítulo está organizado em quatro secções. A primeira secção detalha mais o contexto da empresa e os problemas do processo atual. A segunda secção explora alguns trabalhos e artigos realizados pela comunidade científica de projetos de geração de código e *scaffolding*, e de como as soluções desenvolvidas nesses projetos se comparam com a solução proposta na presente dissertação. A terceira secção descreve algumas ferramentas e soluções já existentes no mercado com capacidades de geração de código ou capacidades de abstração de lógica para o desenvolvimento de aplicações. A última secção explora algumas tecnologias para cada uma das componentes individuais do projeto.

2.1 Contexto da Empresa

A Critical TechWorks é uma empresa com uma dimensão considerável, contando já com mais de mil colaboradores. A empresa é composta por várias unidades, cada uma com um setor de negócio diferente da BMW. Por exemplo, existem unidades que criam *software* para as fábricas da BMW, unidades que criam *software* para o carro, unidades que criam ferramentas internas (tanto para a Critical TechWorks, como para a BMW), etc. E cada uma dessas unidades está dividida em áreas mais específicas.

A solução descrita no presente documento foi desenvolvida para uma das unidades que desenvolve *software* para o carro, e especificamente para a área de *Infotainment & Interactivity Services*. Esta área, como já mencionado, tem como objetivo o desenvolvimento de serviços *online* para o sistema de *infotainment* dos veículos BMW. Assim, sempre que neste documento forem mencionadas as equipas da Critical TechWorks, ou os clientes para os quais se destina esta solução, está-se a referir especificamente às equipas de desenvolvimento da área de *Infotainment & Interactivity Services*, e não à empresa toda.

Esta área da Critical TechWorks tem tido um crescimento saudável desde a criação da empresa. Quando esta área foi criada em 2018, contava apenas com uma equipa e um serviço. Hoje em dia, a área de *Infotainment & Interactivity Services* já conta com quatro equipas e vários serviços diferentes.

Todos os produtos ou serviços desenvolvidos pelas equipas de *Infotainment & Interactivity Services*, como base, possuem uma infraestrutura partilhada. Por exemplo, todos os serviços encontram-se no mesmo *cluster*. Contudo, isso não significa que a infraestrutura de cada serviço é exatamente igual. Cada equipa vai acabar por ter partes da infraestrutura únicas aos seus serviços.

Quando uma nova equipa é criada , ou quando uma equipa começa a desenvolver um novo produto ou serviço, o processo inicial costuma ser semelhante em todos os casos, sendo que todas as equipas acabam por fazer as mesmas tarefas, enumeradas na secção 1.3. Isso faz com que os inícios de novos projetos sejam mais lentos do que poderiam ser, uma vez que as equipas gastam sempre algum tempo simplesmente a fazer o "*setup*" do projeto. Com uma solução que automatizasse este processo, iriam começar a desenvolver o serviço e a criar valor mais rapidamente.

2.2 Trabalhos Relacionados

Esta secção vai abordar alguns trabalhos já realizados pela comunidade científica sobre o tema de geração de projetos. A seleção de trabalhos aqui presente foi resultante de uma pesquisa efetuada com o intuito de ganhar conhecimento sobre o tema e analisar diferentes abordagens para resolver o problema em questão. No entanto, é de importante referência que não foram encontrados muitos estudos ou implementações de ferramentas deste género, o que poderá significar que este problema é ainda uma questão pouco explorada pela comunidade de engenharia informática. Isso talvez seja pela dificuldade em se desenvolver uma ferramenta que consiga dar resposta a uma área que se encontra em constante mudança tecnológica, e em que os projetos podem ter requisitos completamente distintos.

No artigo de investigação "*Model-Based Scaffolding Code Generation for Cross-Platform Applications*", Inayatullah, Azam e Anwar (2019) propuseram um modelo para gerar um projeto simultaneamente para várias plataformas, com componente *web*, *mobile* e um serviço RESTful, isto é, um serviço que aplica os princípios *Representational State Transfer* (REST). A solução descrita nesse artigo gera um projeto com uma *stack* tecnológica pré-definida: Angular ou React para a componente *web*, Ionic ou React Native para a componente *mobile* e ASP.NET para o serviço RESTful. Como ferramenta de geração de código, foi utilizado o Acceleo, que é uma ferramenta baseada em *templates*. No caso do artigo em questão, a abordagem seguida foi um sistema baseado em modelos. O utilizador iria desenhar o sistema em diagramas *Unified Modeling Language* (UML), sendo que a estrutura do sistema iria ser definida por diagramas de classes. Depois, a ferramenta desenvolvida iria gerar o código do projeto com base nesses diagramas.

No artigo seguinte, intitulado "*Scaffolding and in-browser generation of web-based GIS applications in a SPL tool*", Cortiñas et al. (2017) desenvolveram uma ferramenta para gerar código de uma aplicação *web* de *Geographic Information Systems* (GIS). Esta ferramenta é composta por dois componentes principais: a *Web Interface*, onde o utilizador vai especificar a aplicação que quer gerar, e o *Derivation Engine*, que é a parte principal da ferramenta e aquela que vai gerar o código final. Em termos tecnológicos, o artigo não detalha as tecnologias utilizadas, apenas dizendo que a *Web Interface* foi realizada em Angular e o *Derivation Engine* foi desenvolvido utilizando uma biblioteca JavaScript baseada em *scaffolding*. A geração do código é realizada com base num *template* pré-definido onde o utilizador especifica o produto.

O primeiro artigo descreveu uma abordagem interessante para geração de código, com base em modelos UML. No entanto, não é claro se as equipas teriam controlo sobre o código gerado ou sobre a estrutura de ficheiros definida, ou se teriam possibilidades de personalizar a ferramenta. O segundo artigo descreveu uma solução específica para o desenvolvimento de aplicações GIS,

e não entrou em detalhe suficiente sobre a forma como os autores implementaram e desenvolveram a solução. Outro ponto a considerar é que as ferramentas desenvolvidas nos artigos mencionados apenas geram o código para as aplicações em si, e não para todas as componentes que suportam e interagem com uma aplicação, tais como *containerization*, infraestrutura ou CI/CD. Pela pesquisa que foi efetuada, não foi encontrado nenhum artigo ou nenhum outro projeto que conseguisse dar resposta a todas as necessidades do problema que se quer solucionar nesta dissertação.

2.3 Soluções Existentes

Nesta secção é realizada uma pesquisa e análise breve de algumas soluções existentes no mercado que ajudem no processo de criação de um novo projeto de *software*.

Devido à quantidade e especificidade dos requisitos da ferramenta a ser desenvolvida, que serão analisados em mais detalhe no capítulo seguinte, não existe nenhuma única solução capaz de responder a todos os requisitos. Porém, estas soluções foram analisadas de forma a realizar-se um reconhecimento de características e funcionalidades que eventualmente poderiam ser aproveitadas na solução final, ou que poderiam servir de referência durante o seu desenvolvimento. Também é importante referir que esta lista não inclui toda a extensão de ferramentas de geração de código existentes no mercado, mas apenas uma amostra de algumas das ferramentas mais populares nas linguagens de programação mais utilizadas. As ferramentas apresentadas nesta secção são o resultado de uma pesquisa efetuada sobre esta secção de mercado.

2.3.1 Yeoman

O Yeoman é atualmente uma das ferramentas de *scaffolding* mais populares no mercado. Este tipo de ferramentas de geração de código são também conhecidas por ferramentas de *scaffolding*[1], uma vez que são responsáveis por gerar os "andaimes" dos projetos.

O Yeoman sugere um fluxo de trabalho baseado em três tipos de ferramentas, como mostrado pela figura 2.1.

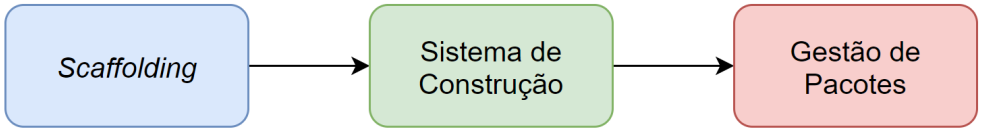

FIGURA 2.1: Os três tipos de ferramentas utilizadas no fluxo de trabalho do Yeoman.

Como ferramenta de *scaffolding*, o Yeoman utiliza o yo, que é uma ferramenta utilizada em linha de comandos para gerar projetos baseados em *templates* de *scaffolding*, que no Yeoman são conhecidos por *generators*. O Yeoman oferece uma vasta seleção de *generators* construídos pela comunidade, e também permite a criação de novos *generators*, caso o utilizador não encontre nenhum que vá de encontro às suas necessidades.

O sistema de construção é utilizado para construir e testar o projeto. O Yeoman sugere ferramentas como o Gulp e Grunt, pois existem vários *generators* que dependem destas ferramentas.

[1] *scaffold* significa "andaime" em português

Finalmente, O Yeoman utiliza um gestor de pacotes para automatizar o processo de instalar, atualizar, configurar e gerir as dependências dos projetos. O gestor de pacotes utilizado pelo Yeoman é o *Node Package Manager* (NPM).

Para utilizar o Yeoman, como já mencionado, é necessário instalar a ferramenta yo. Essa ferramenta pode ser instalada com o NPM, utilizando o seguinte comando:

```
npm install -g yo
```

EXCERTO 2.1: Comando para instalar o yo.

De seguida, é instalado o *generator* que o utilizador desejar. Por exemplo, para instalar o *generator* "webapp", é executado o seguinte comando:

```
npm install -g generator-webapp
```

EXCERTO 2.2: Comando para instalar o *generator* "webapp".

Por fim, o *generator* é executado para gerar o *scaffolding* do projeto:

```
yo webapp
```

EXCERTO 2.3: Comando para executar o *generator* "webapp".

2.3.2 Spring Boot

Baseada em *framework* Spring, o Spring Boot é uma das ferramentas de geração de código mais conhecidas no universo Java. Porém, não tem de ser utilizado somente com Java, o Spring Boot pode ser também utilizado com as linguagens Kotlin e Groovy.

Spring é uma *framework open-source* criada para facilitar alguns dos aspetos mais complexos do desenvolvimento de aplicações empresariais em Java. Contudo, uma desvantagem que a *framework* Spring tem é a quantidade inicial de ficheiros de configuração que necessita. Por exemplo, para criar uma simples aplicação *web* utilizando Spring, vão ser necessários, no mínimo, os seguintes passos (Walls 2016):

- Uma estrutura de projeto, completa com um ficheiro Maven ou Gradle para construir o projeto e instalar as dependências necessárias.
- Um ficheiro `web.xml` para declarar o `DispatcherServlet` do Spring. A principal função do `DispatcherServlet` é processar os pedidos *Hypertext Transfer Protocol* (HTTP).
- Uma configuração para ativar o Spring *Model-View-Controller* (MVC).
- Uma classe *controller* com a lógica da aplicação para responder aos pedidos HTTP.
- Um servidor *web*, como o Tomcat, por exemplo, para alojar a aplicação.

De toda esta lista, o único item que contém lógica específica do desenvolvimento da aplicação é o *controller*. O resto são ficheiros genéricos de configuração do Spring.

O Spring Boot tem como objetivo resolver este problema, criando automaticamente os ficheiros de configuração necessários. Isso significa que não é necessário nenhum ficheiro com instruções de construção da aplicação, não é necessário nenhum ficheiro `web.xml`, e também não é necessário nenhum servidor para executar a aplicação. Apenas é necessário o código com a lógica da aplicação, e o Spring Boot trata do resto automaticamente.

2.3.3 JHipster

O JHipster é também uma ferramenta de *scaffolding*, capaz de gerar de um projeto *full-stack*. Com esta ferramenta é possível:

- Gerar um projeto *front-end* em React, Angular ou Vue.js.
- Gerar um projeto *back-end* em Java, baseado em Spring, através da utilização do Spring Boot. Para a componente *back-end* pode ser utilizado o Maven ou o Gradle para construir, testar e executar a aplicação.
- Testar tanto o *front-end* como o *back-end*. Para testar o *front-end* é utilizado o Jest e o Protractor. Para testar o *back-end* é utilizado o JUnit 5 e o ArchUnit. Adicionalmente, o JHipster pode também gerar testes de desempenho com o Gatling e testes funcionais com o Cucumber. Para além disso, é ainda possível testar a qualidade do código com o SonarCloud.
- Gerar uma base de dados, que pode ser tanto relacional como não relacional. As bases de dados que podem ser geradas com o JHipster são: H2, MySQL, MariaDB, PostgreSQL, MSSQL, Oracle, MongoDB, Cassandra, Couchbase ou Neo4j.
- Ter suporte para *containerization*, através do Docker e Docker Compose.
- Ter mecanismos de CI/CD, através de ferramentas como o Jenkins, Travis ou CircleCI.
- Fazer *deploy* das aplicações para fornecedores *cloud* como AWS e Azure.

É, portanto, uma ferramenta bastante completa e com bastantes opções de customização.

Existem algumas formas diferentes de instalar o JHipster, sendo que a forma mais popular é a instalação local através do NPM. Para isso, o JHipster tem como requisitos o Java 11 e o Node.js. Tendo esses dois requisitos instalados, basta executar o seguinte comando para instalar esta ferramenta:

```
npm install -g generator-jhipster
```

EXCERTO 2.4: Comando para instalar o JHipster.

2.3.4 Helidon

Helidon é um conjunto de bibliotecas Java com o objetivo de facilmente gerar micro-serviços. Desenvolvido pela Oracle inicialmente para utilização interna, esta *framework* foi depois lançada para o público num contexto *open-source*. A *framework* Helidon está disponível em duas versões diferentes: Helidon *Standard Edition* (SE) e Helidon MicroProfile (MP).

A versão SE usa a sua própria *Application Programming Interface* (API) e utiliza um estilo de programação funcional, distinguindo-se pelo uso bastante reduzido de anotações. A versão MP tem uma implementação to Eclipse MicroProfile, que torna possível a utilização de APIs do Java *Enterprise Edition* (EE) como JAX-RS ou CDI, e também define novas APIs como um *health check* e uma página de métricas (disponíveis nos *endpoints* `\health` e `\metrics`, respetivamente). A versão MP, ao invés da SE, tende a utilizar um estilo de programação mais declarativo, com uma forte utilização de anotações (Oracle 2020).

Segue-se um excerto de código para gerar um projeto com o Helidon MP, utilizando o seu *archetype* Maven:

```
mvn -U archetype:generate -DinteractiveMode=false \
    -DarchetypeGroupId=io.helidon.archetypes \
    -DarchetypeArtifactId=helidon-quickstart-mp \
    -DarchetypeVersion=2.2.0 \
    -DgroupId=io.helidon.examples \
    -DartifactId=helidon-quickstart-mp \
    -Dpackage=io.helidon.examples.quickstart.mp
```

EXCERTO 2.5: Código para executar o *archetype* Maven.

O projeto gerado também possui um ficheiro `Dockerfile`, pelo que se consegue construir e executar a aplicação num *container* Docker. A documentação oficial da ferramenta também tem instruções para, posteriormente, efetuar facilmente o *deploy* desse *container* num *cluster* Kubernetes, assumindo que já existe um *cluster* Kubernetes previamente instalado.

2.3.5 Express.js

O Express.js é a *framework web* mais popular para Node.js, que proporciona um conjunto robusto de funcionalidades para o desenvolvimento de aplicações *web* e aplicações móveis (Mardan 2014). Apesar de tecnicamente não ser uma ferramenta de geração de código, a sua utilização acaba por abstrair muitas das complexidades do desenvolvimento de aplicações *web*, sendo por isso uma ferramenta interessante para este projeto.

O Node.js é um ambiente em tempo de execução[2] *open-source* que permite a execução de código JavaScript, que previamente apenas era possível ser executado em navegadores de internet, em servidores. Utiliza um modelo de *Input/output* (I/O) não bloqueador e *single-threaded*, o que significa que uma *thread* pode trabalhar numa tarefa enquanto está a aguardar que uma outra tarefa termine (Gleason 2020), assegurando assim que novas tarefas não são bloqueadas por outras tarefas que ainda não terminaram. Para conseguir isso, o Node.js utiliza funções assíncronas, sendo por isso uma ferramenta eficaz para o desenvolvimento de sistemas escaláveis (Foundation 2020).

Quando um programador quer criar uma aplicação *web* em Node.js, pode criar uma única função JavaScript para toda a aplicação. Esta função é conhecida como um *request handler*, que simplesmente significa que é uma função que processa pedidos realizados por um cliente, e realiza uma determinada tarefa com base nesses pedidos. Todavia, as APIs do Node.js podem tornar-se complexas, sendo que caso se queira um *request handler* mais robusto ou com mais funcionalidades, a sua implementação pode ficar desnecessariamente confusa. O Express.js vem resolver estes problemas com duas principais medidas: primeiro, tornando a API do servidor HTTP do Node.js mais simples, abstraindo assim muita da sua complexidade; depois, permitindo que os programadores tornem um *request handler* monolítico em pedaços mais pequenos com funcionalidades mais específicas (Hahn 2016).

A sua instalação é bastante simples, fazendo uso do popular NPM, através do seguinte comando:

```
npm install express --save
```

EXCERTO 2.6: Comando para instalar o Express.js.

O seguinte excerto de código mostra a facilidade com que se cria uma aplicação *web* com o Express.js:

[2] *runtime environment*

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
    res.send('Hello World!')
})

app.listen(port, () => {
    console.log(`Example app listening at http://localhost:${port}`)
})
```

EXCERTO 2.7: Código para criar uma aplicação *web* com o Express.js.

O excerto de código 2.7 inicia um servidor na porta 3000, ficando depois à espera de conexões. A aplicação vai responder com "Hello World!" a pedidos efetuados ao *Uniform Resource Locator* (URL) base (neste caso, `http://localhost:3000/`).

2.3.6 Django

Criado em 2003, a *framework* Django foi desenvolvida para permitir a criação de aplicações *web* utilizando Python. Esta *framework* foi criada especialmente para programadores que privilegiem um código limpo e uma arquitetura simples e clara. Outra vantagem é também o facto de ser uma *framework* costumizável e modular. Existe na comunidade uma boa seleção de módulos Django para uma variedade de cenários diferentes, que os programadores podem incluir na sua aplicação (Dauzon, Bendoraitis e Ravindran 2016).

A sua instalação é também um processo simples, tendo como pré-requisitos apenas a instalação do Python, uma vez que é a linguagem que está na base desta ferramenta, e o `pip`, que é uma ferramenta de instalação de pacotes Python. Tendo estes pré-requisitos instalados, a instalação do Django consegue-se com o seguinte comando:

```
python -m pip install Django
```

EXCERTO 2.8: Comando para instalar o Django com o `pip`.

Depois, para criar um projeto utilizando o Django, executa-se o seguinte comando:

```
django-admin startproject mysite
```

EXCERTO 2.9: Comando para criar um projeto com o Django.

O comando 2.9 vai criar o projeto `mysite` com a seguinte estrutura de ficheiros:

```
mysite/
├── manage.py
└── mysite/
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    ├── asgi.py
    └── wsgi.py
```

Sendo que:

- A pasta `mysite/` exterior é o diretório base do projeto.

- O ficheiro `manage.py` é uma ferramenta em linha de comandos para executar tarefas de administrador.
- A pasta `mysite/` interior é a pasta que contém o projeto Python propriamente dito. O nome desta pasta é o nome que deve ser utilizado nas importações no código.
- O ficheiro `mysite/__init__.py` é um ficheiro vazio que serve para o Python saber que esta pasta se trata de um pacote Python.
- O ficheiro `mysite/settings.py` contém algumas configurações deste projeto.
- O ficheiro `mysite/urls.py` contém as declarações URL do projeto.
- Os ficheiros `mysite/asgi.py` e `mysite/wsgi.py` são os pontos de entrada para servidores *web* compatíveis com *Asynchronous Server Gateway Interface* (ASGI) ou *Web Server Gateway Interface* (WSGI), respetivamente.

Para iniciar o servidor, basta executar o seguinte comando na pasta `mysite/` exterior:

```
python manage.py runserver
```

EXCERTO 2.10: Comando para iniciar o servidor.

2.3.7 Flask

O Flask, tal como o Django, é uma *framework web* baseada em Python. Todavia, distingue-se do Django por ser uma *framework* consideravelmente mais leve e compacta. Por exemplo, o seguinte excerto de código é uma aplicação *web* em Flask:

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route('/')
5 def hello_world():
6     return 'Hello, World!'
7
8 if __name__ == '__main__':
9     app.run()
```

EXCERTO 2.11: Aplicação *web* utilizando o Flask.

Neste excerto de código, em primeiro lugar está a ser importada a classe `Flask`. Depois, essa classe é instanciada, o que irá gerar a aplicação WSGI. Na instanciação está a ser passado o argumento `__name__`, pois o nome vai ser diferente dependendo se for inicializado como aplicação ou se for importado como um módulo. Caso seja inicializado como aplicação, este valor será `__main__`, que é a validação que está a ser feita no final do código. Finalmente, a meio do código, é utilizada a função `route()`, que diz ao Flask o URL que deve despoletar a função (Grinberg 2018).

Embora muito simples, a aplicação descrita com o excerto de código 2.11 é uma aplicação *web* válida, respondendo com "Hello, World!" a pedidos feitos à porta 5000 do `localhost` num navegador de internet. Uma aplicação *web* equivalente, mas utilizando Django ao invés de Flask, necessitaria de mais código *boilerplate* (Python 2020).

Para a sua instalação, o Flask tem algumas dependências. Para começar, tal como com o Django, é necessário ter o Python instalado, uma vez que também está na base desta *framework*. As restantes dependências são as seguintes (Projects 2020):

- **Werkzeug**: Implementa o WSGI, que é a interface mais comum entre aplicações e servidores.
- **Jinja**: É um motor de *templating* responsável pelo processamento e apresentação das páginas *web* disponibilizadas pela aplicação.
- **MarkupSafe**: Vem incluído com o Jinja, e é responsável pela sanitização dos dados de entrada para impedir possíveis vulnerabilidades e ataques de injeção.
- **ItsDangerous**: Responsável por assinar dados de forma segura, para assim assegurar a sua integridade. É utilizado para proteger a *cookie* de sessão do Flask.
- **Click**: É uma *framework* para escrever aplicações em linha de comandos. É responsável pelo comando `flask` e permite também a criação de novos comandos personalizáveis de administração.

2.4 Tecnologias das Componentes

Nesta secção são analisadas as tecnologias para cada uma das componentes individuais da solução. Foi realizado um levantamento de algumas tecnologias de cada segmento, sendo que serão posteriormente comparadas em mais detalhe no próximo capítulo.

Tal como mencionado na secção anterior, esta secção também não vai incluir toda a seleção atualmente existente de tecnologias, mas apenas uma amostra das tecnologias mais relevantes de cada segmento, explorando um pouco as suas funcionalidades e como estas poderiam ser úteis para o projeto.

2.4.1 *Front-end*

A camada *front-end* diz respeito à parte da aplicação do lado do cliente[3]. É a camada com a qual o utilizador interage para utilizar a aplicação.

Na camada *front-end* existem diversas bibliotecas e *frameworks* muito populares hoje em dia e com um grande conjunto de funcionalidades. Contudo, a base da vasta maioria dos projetos *front-end* é quase sempre idêntica, sendo constituída pelo que é considerado atualmente como o trio padrão de tecnologias *front-end*, que são o *Hypertext Markup Language* (HTML), o *Cascading Style Sheets* (CSS) e o JavaScript.

O HTML é uma linguagem de *templating* que está na base da criação de qualquer página *web*. É uma linguagem baseada em *tags*, ou anotações, que serve para estruturar e organizar o conteúdo das páginas. Foi desenvolvida na Conseil Européen pour la Recherche Nucléaire (CERN) com o intuito de organizar documentos científicos submetidos por cientistas de todo o mundo com sistemas de computadores incompatíveis. O desenvolvimento de códigos de formatação e estruturação, bem como códigos de hiperligação, tornou possível a organização de documentos com referências cruzadas de informação através de *links* (Musciano e Kennedy 2002).

O CSS é uma linguagem que normalmente está sempre associada ao HTML. Enquanto que o HTML é utilizado para construir a página *web*, o CSS é usado para personalizar os elementos da página.

[3] *client-side*

Embora também seja possível alterar os estilos dos elementos no próprio HTML, este nunca foi o objetivo dessa linguagem. Em vez disso, é considerado boa prática realizar as personalizações dos elementos num documento CSS.

O JavaScript é uma linguagem de programação para complementar as duas linguagens apresentadas previamente. Enquanto que o HTML serve para construir a página e o CSS define a sua aparência, o JavaScript serve para desenvolver a lógica da página e as funcionalidades da componente *front-end* da aplicação (Flanagan e Matilainen 2007). Apesar de o JavaScript ter nascido inicialmente como uma linguagem *front-end* para ser executada pelo navegador de internet, atualmente é também uma ferramenta bastante capaz em *back-end* devido ao Node.js.

Em tempos prévios no mundo do desenvolvimento *front-end*, as aplicações *web* eram compostas por várias páginas, em que um novo documento HTML era carregado sempre que o utilizador navegava para outra página. Hoje em dia existe o modelo *Single-Page Application* (SPA), onde é carregada uma única página HTML e o conteúdo dessa página é alterado dinamicamente com as ações do utilizador. Dessa forma, apenas são feitos pedidos ao servidor das partes da página que são alteradas, oferecendo assim um maior desempenho.

Devido a isto, o modelo SPA tem vindo a ser cada vez mais utilizado ao longo dos anos e, consequentemente, também as bibliotecas e *frameworks* que são usadas para construir SPAs têm vindo a ganhar mais popularidade. Estas bibliotecas e *frameworks*, baseadas em JavaScript, facilitam o processo de desenvolvimento através de um vasto leque de funcionalidades que os programadores podem aproveitar para os seus projetos.

A figura 2.2 representa um gráfico parcial obtido do questionário de 2020 realizado pelo conhecido *website* Stack Overflow. Este questionário trata-se de um questionário anual que o Stack Overflow lança à sua comunidade. Sendo este *website* um dos sítios com uma maior comunidade de profissionais na área de informática, os seus resultados podem ser considerados bons indicadores das tecnologias que a comunidade no geral está a utilizar ou quer aprender. Esta figura em específico retrata as dez *frameworks web* mais utilizadas pela comunidade, e inclui tanto *frameworks front-end* como *back-end*. Olhando apenas para as *frameworks web front-end* utilizadas no desenvolvimento de SPAs, conclui-se que as três *frameworks front-end* mais populares atualmente são o React, o Angular e o Vue.js, que foram as tecnologias consideradas para este projeto.

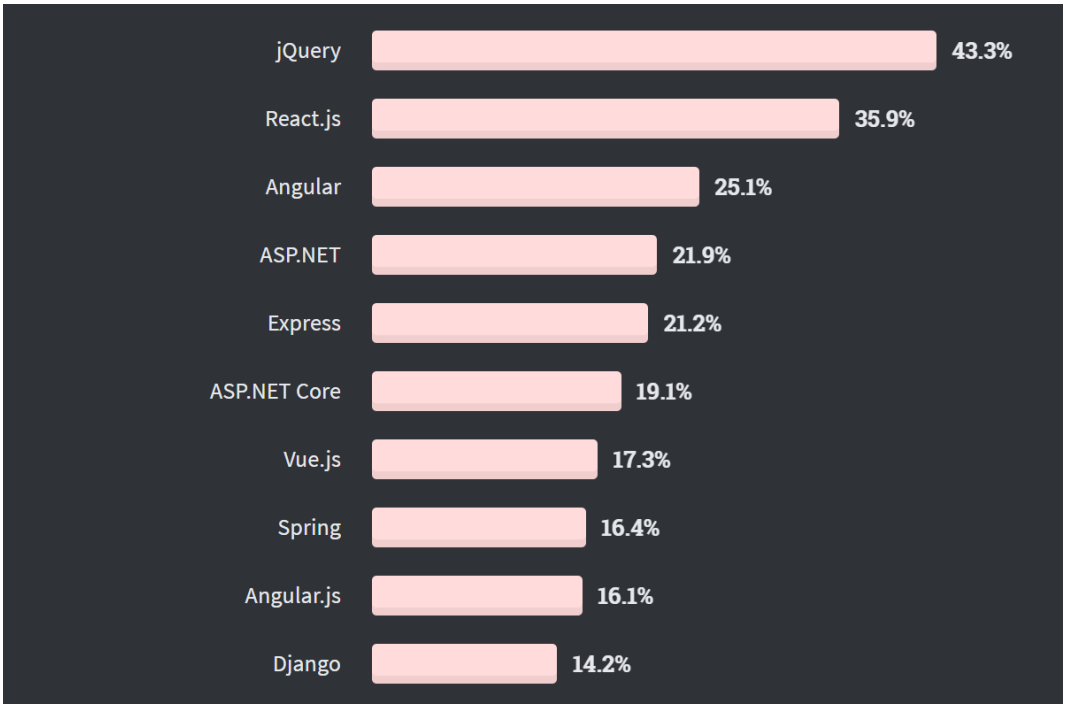

FIGURA 2.2: As *frameworks web* mais populares, segundo o questionário de 2020 do Stack Overflow. Imagem obtida em 2021-01-18, de: `https://insights.stackoverflow.com/survey/2020`.

React

Apesar de muitas vezes ser tratada como tal pela comunidade, o React não se auto-intitula como uma *framework*, mas sim uma biblioteca. Desenvolvida em 2011 pelo Facebook, inicialmente para uso interno, a biblioteca foi posteriormente disponibilizada em regime *open-source* em 2013, ganhando rapidamente bastante popularidade no universo de desenvolvimento *front-end*.

Desde então, o React vindo a receber novas adições e atualizações. Uma dessas adições foi o React Native em 2015, que é uma solução baseada em React mas orientada para o desenvolvimento de aplicações móveis.

O React foi desenhado para construir elementos interativos de *User Interface* (UI) num sistema baseado em componentes, componentes esses que podem ser combinados para criar UIs mais complexas. Os componentes são desenvolvidos em JavaScript, permitindo assim de forma mais fácil o fluxo e manipulação de dados (Saks 2019).

Angular

O Angular é uma *framework front-end* desenvolvida pela Google. Foi criada em 2008, na altura ainda sobre o nome de AngularJS, por engenheiros da Google que estavam a criar uma nova ferramenta interna (Saks 2019). Numa altura em que a maioria dos *websites* e aplicações *web* ainda utilizavam uma abordagem multi-página, o AngularJS foi uma das primeiras *frameworks* que impulsionaram o conceito de SPA (Wohlgethan 2018).

O AngularJS segue parcialmente a arquitetura MVC, que pode ser resumida da seguinte forma:

Tabela 2.1: Explicação da arquitetura MVC.

Model	Responsável por gerir todos os dados da aplicação
View	O que o utilizador vê quando usa a aplicação
Controller	Altera os dados do *model* e atualiza a *view* com os novos dados

Contudo, o AngularJS introduziu uma mudança na implementação tradicional da arquitetura MVC com uma nova funcionalidade e um novo conceito, que foi a ligação direta e bidirecional de dados[4]. Nas implementações tradicionais da arquitetura MVC, quando o *controller* modifica o *model*, é necessário recarregar a página para ver essa alteração na *view*. Com a ligação direta e bidirecional de dados não é necessário qualquer recarregamento, qualquer alteração no *model* é imediatamente atualizada na *view* em tempo real, e qualquer alteração na *view* é atualizada no *model*. Funcionalidades como a ligação direta e bidirecional de dados, ou a possibilidade de organização de módulos para importar *scripts* externos, fizeram com que o AngularJS conseguisse ultrapassar outras *frameworks web* populares da altura, como o jQuery, por exemplo.

Em 2014 foi anunciado o Angular 2, uma atualização significativa da *framework*. Esta nova versão foi uma versão completamente diferente do Angular 1, envolvendo várias alterações nos conceitos base da *framework*. As versões a partir do Angular 2 são todas retrocompatíveis com versões anteriores, menos com o Angular 1. Assim, para não criar confusão na comunidade, o Angular 1 ficou conhecido como AngularJS, e as versões a partir da versão 2 são conhecidas apenas como Angular.

Vue.js

Ao contrário do React ou do Angular, o Vue.js não foi desenvolvido nem é mantido por uma grande empresa. Apesar de tanto o React e o Angular estarem atualmente num regime *open-source*, estes continuam a ser mantidos pelo Facebook e pela Google, respetivamente.

O Vue.js foi desenvolvido em 2014 por um ex-funcionário da Google, o Evan You, que considerava que o Angular era uma *framework* demasiado pesada para certos casos de uso. Então, Evan You extraiu as partes que gostava do Angular e baseou-se nesses conceitos para criar uma *framework* consideravelmente mais leve (Saks 2019).

Apesar de o projeto ter sido iniciado apenas por uma pessoa, atualmente este é mantido por uma equipa dedicada, e conta também com o apoio da comunidade em geral, uma vez que se encontra em regime *open-source*.

2.4.2 *Back-end*

A camada do *back-end* engloba a parte da aplicação do lado do servidor[5]. É a camada que vai conter a lógica de negócio da aplicação ou do serviço, e é a camada que vai processar os pedidos realizados pelo cliente no *front-end*. A componente *back-end* tradicionalmente engloba três partes principais: a aplicação, o servidor e a base de dados. Nesta secção são referidas somente tecnologias para a parte da aplicação e do servidor. A parte da base de dados não irá ser abordada nesta solução, uma vez que este projeto se trata apenas de um *archetype* genérico. A base de dados escolhida para um determinado projeto de *software* vai depender das necessidades e das

[4] *two-way data-binding*
[5] *server-side*

características do projeto, pois é isso que vai determinar se é melhor utilizar uma base de dados relacional ou não relacional.

Em termos de tecnologias utilizadas para um projeto *back-end*, conseguem-se tirar algumas conclusões a partir do mesmo questionário de 2020 do Stack Overflow que foi mencionado na secção anterior. A figura 2.3 mostra as dez linguagens mais populares de programação, *scripting* e *markup*.

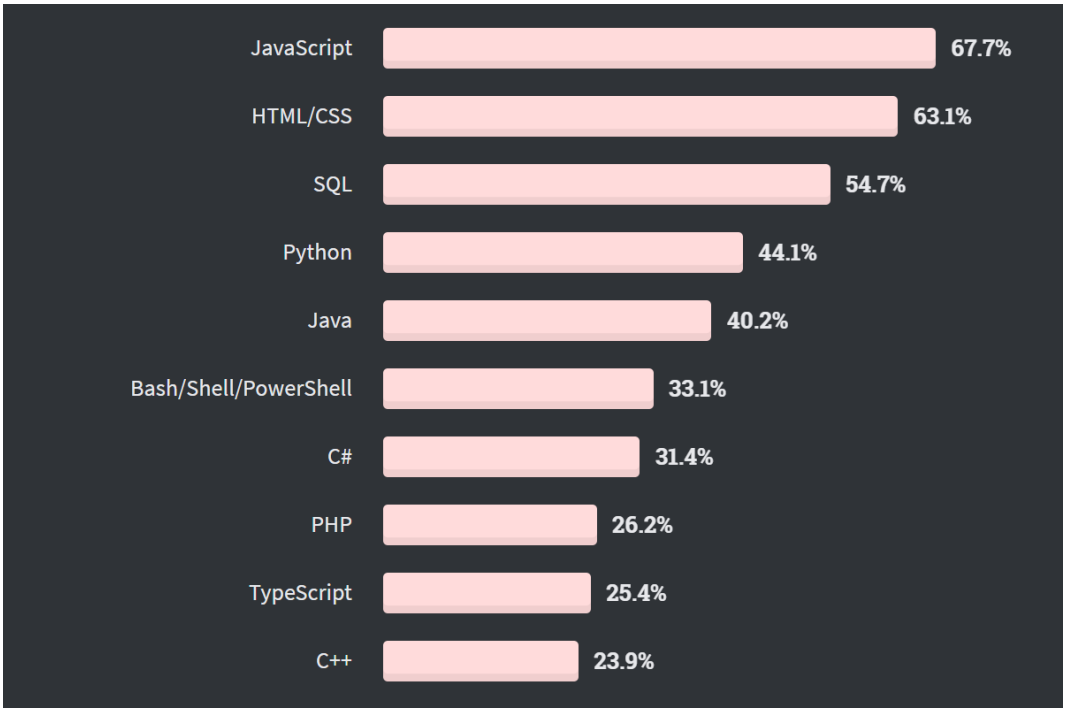

FIGURA 2.3: As linguagens de programação mais populares, segundo o questionário de 2020 do Stack Overflow. Imagem obtida em 2021-01-18, de: `https://insights.stackoverflow.com/survey/2020`.

Como se pode ver pela figura 2.3, o JavaScript encontra-se na liderança como a linguagem de programação mais utilizada pela comunidade, algo que já o faz pelo oitavo ano consecutivo, segundo o Stack Overflow. A razão desta liderança não está apenas relacionada com a contínua popularidade das aplicações *web*, mas também com a crescente utilização de JavaScript para desenvolvimento *back-end*, graças ao Node.js.

De seguida, e apenas falando em linguagens que podem ser utilizadas na componentes *back-end*, em aplicações e servidores, vem o Python e Java. Estas são linguagens bastante completas, podendo ser utilizadas numa variedade de contextos diferentes devido à sua versatilidade.

Assim, as linguagens consideradas para a componente *back-end* da solução deste projeto foram o JavaScript (nomeadamente o Node.js), Python e Java. Estas tecnologias foram selecionadas não só pela sua popularidade, mas por todas as restantes qualidades que oferecem, que são analisadas em mais detalhe no próximo capítulo.

Java

A linguagem Java foi originalmente desenvolvida e lançada em 1995 pelo James Gosling na empresa Sun Microsystems, que foi entretanto adquirida pela Oracle. É uma linguagem de programação baseada em classes e orientada a objetos, desenhada com a filosofia de "escrever uma vez, executar em qualquer lado", uma vez que código Java compilado pode ser executado em qualquer plataforma que suporte Java sem a necessidade de recompilação (Gosling, Holmes e Arnold 2005).

Para além disso, o Java continua a ser das escolhas mais populares para programadores pelas suas atualizações e consistência de qualidade ao longo dos anos. Apesar de já contar com quase três décadas de existência, a linguagem de programação foi sendo refinada, testada e estendida por uma comunidade dedicada de programadores, arquitetos e entusiastas. Assim, o Java é atualmente uma das linguagens de programação mais estáveis e mais poderosas para desenvolver uma grande variedade de projetos.

No entanto, apesar de ser possível utilizar Java para *front-end*, através de ferramentas como JavaFX e Swing para aplicações *desktop*, e do Jakarta *Server Pages* (JSP) para aplicações *web*, nos dias de hoje a comunidade tende a preferir outras ferramentas mais apropriadas e com mais funcionalidades para esses fins. Assim, o ponto forte da linguagem Java encontra-se especialmente no desenvolvimento *back-end*.

Node.js

O Node.js foi criado em 2009 pelo Ryan Dahl. A principal motivação de Dahl foi o facto de o servidor *web* mais popular em 2009, o Apache HTTP *Server*, ter possibilidades limitadas no processamento de muitas conexões concorrentes, e pelo facto de os processos poderem ser bloqueados pelo código em situações de conexões simultâneas.

Para corrigir essas lacunas, Ryan Dahl desenhou o Node.js com uma arquitetura não bloqueadora. Para além disso, o Node.js é *single-threaded* por natureza, ou seja, apenas tem uma *thread* por processo. Estes dois fatores impossibilitam a ocorrência de *deadlocks*, que é o fenómeno em que dois ou mais processos ficam bloqueados à espera uns dos outros, o que causa o bloqueio de todo o programa. Isso torna o Node.js uma boa ferramenta para soluções que necessitem de fazer bastantes ações de I/O, como por exemplo processamento de arquivos em grande escala (Pereira 2014).

Ao instalar o Node.js, é também instalado o NPM, um dos maiores gestores de pacotes existentes atualmente, e o gestor de pacotes padrão no universo JavaScript. O NPM já foi referido neste documento, na secção 2.3.5, onde também se encontra um exemplo de como instalar um pacote com esta ferramenta.

Python

A linguagem Python foi desenvolvida em 1990 por Guido van Rossum no Centrum Wiskunde Informatica (CWI). Python distingue-se das demais linguagens de programação devido a uma sintaxe particularmente simples e acessível, existindo mesmo certos pedaços de código que parecem frases em inglês corrente. Contudo, apesar disso, é uma linguagem bastante capaz e poderosa, incluindo diversas estruturas de alto nível e uma ampla coleção de módulos que os programadores podem importar para os seus projetos (Borges 2014).

A sua relativa baixa curva de aprendizagem, bem como a sua versatilidade, tornam o Python uma boa escolha para uma grande variedade de projetos, como, por exemplo, *scripting*, computação

científica ou *machine learning* (Pedregosa et al. 2011). Para além disso, e como já foi previamente mencionado nas secções 2.3.6 e 2.3.7, existem também *frameworks* que fazem do Python uma opção sólida para o desenvolvimento de aplicações e servidores *web*.

2.4.3 *Containerization*

À medida que as necessidades computacionais do mundo aumentavam ano após ano, também aumentava a pressão das empresas para que os seus centros de processamento de dados[6] conseguissem dar resposta aos requisitos dos seus clientes. Nesse sentido, a computação na *cloud* provocou uma mudança de paradigma no mundo tecnológico, fazendo com que cada vez mais empresas migrassem os seus centros de processamento de dados físicos para a *cloud*. Um dos principais conceitos que torna possível a computação na *cloud* é a virtualização.

A virtualização de *software* permite que um único servidor físico possa conter um ou mais ambientes virtuais. É a virtualização que torna possível a infraestrutura na *cloud* que é utilizada hoje em dia, como por exemplo o Amazon *Elastic Compute Cloud* (EC2). O principal objetivo da virtualização é obter uma melhor gestão da carga de trabalho num determinado sistema, tornando-o mais escalável, eficiente e económico (Malhotra, Agarwal, Jaiswal et al. 2014). Quer isso dizer que enquanto um sistema físico está constrangido pelas suas limitações de *hardware*, um sistema virtual pode ser escalado de forma mais flexível, dependendo da carga de trabalho que tiver. Ou seja, pode-se criar um sistema virtual com uma capacidade computacional mais reduzida para uma carga de trabalho normal, e depois aumentar dinamicamente os recursos desse sistema em alturas de maior carga.

Os sistemas virtuais dependem de um componente denominado de "hipervisor", que é um *software* que interage com o *hardware* do sistema físico e garante o correto funcionamento do ambiente virtual (Malhotra, Agarwal, Jaiswal et al. 2014). A figura 2.4 é um diagrama simplificado que representa uma arquitetura de um ambiente de virtualização. Como a figura mostra, é o hipervisor que permite que múltiplas máquinas virtuais (que podem ter sistemas operativos diferentes) possam ser executados na mesma máquina física, e que todos os ambientes virtuais têm acesso aos mesmos recursos físicos. O hipervisor é também responsável por orquestrar e separar os recursos disponíveis (como memória, espaço disponível, processamento, etc.) pelos diferentes ambientes virtuais.

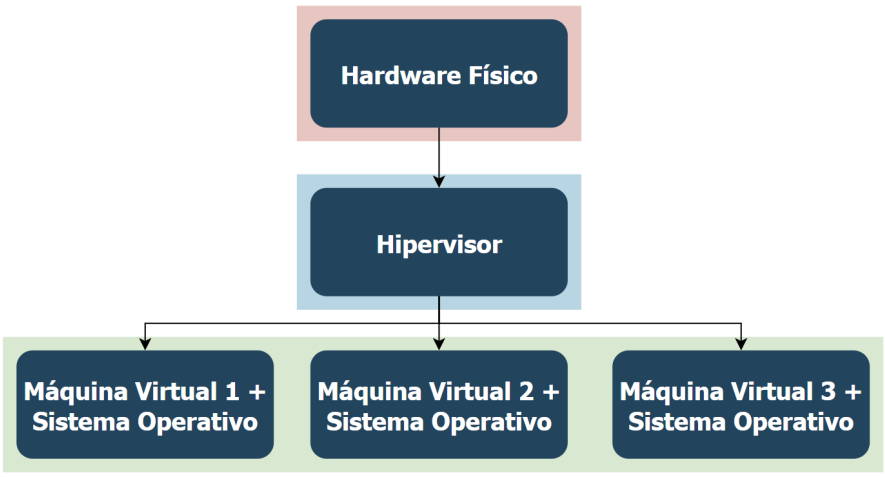

FIGURA 2.4: Arquitetura básica de um ambiente de virtualização.

[6] *data centers*

Contudo, a virtualização também possui algumas desvantagens, sendo a principal os recursos de que necessita. Cada máquina virtual tem de ter um sistema operativo e uma cópia virtual de todo o *hardware* necessário para o correto funcionamento desse sistema operativo. Isso significa que cada máquina virtual vai necessitar de recursos significativos em termos de memória RAM e de processador (Scheepers 2014).

Os *containers* podem ser considerados como uma evolução tecnológica da virtualização, apesar de servirem propósitos ligeiramente diferentes. Os *containers* vieram preencher as necessidades de programadores que precisem de partilhar o seu *software* num ambiente perfeitamente controlado, onde todas as dependências e bibliotecas necessárias já estão presentes (Díaz Alonso 2017). A virtualização, embora também possa ser utilizada para esse objetivo, tem ainda um grau adicional de isolamento proporcionado pelo hipervisor, sendo por isso considerada mais segura em relação a *containers* (Mouat 2015).

A figura 2.5 é uma representação simplista de uma arquitetura de *containerization*. Em contraste com a arquitetura de virtualização representada na figura 2.4, uma arquitetura de *containerization* é baseada numa máquina física com um único sistema operativo. Nessa máquina é configurado um *container engine*, isto é, uma ferramenta de *containerization*, que vai depois ser responsável por fazer a gestão de todos os *containers*, semelhante ao papel do hipervisor. Contudo, os *containers* têm uma diferença significativa, que é o facto de todos os *containers* partilharem o mesmo *kernel* da máquina física. Isso faz com que todos os processos que estão a ser executados em cada *container* sejam tratados como processos nativos da máquina física, não incorrendo dessa forma os mesmos gastos em termos de *performance* associados à virtualização (Mouat 2015).

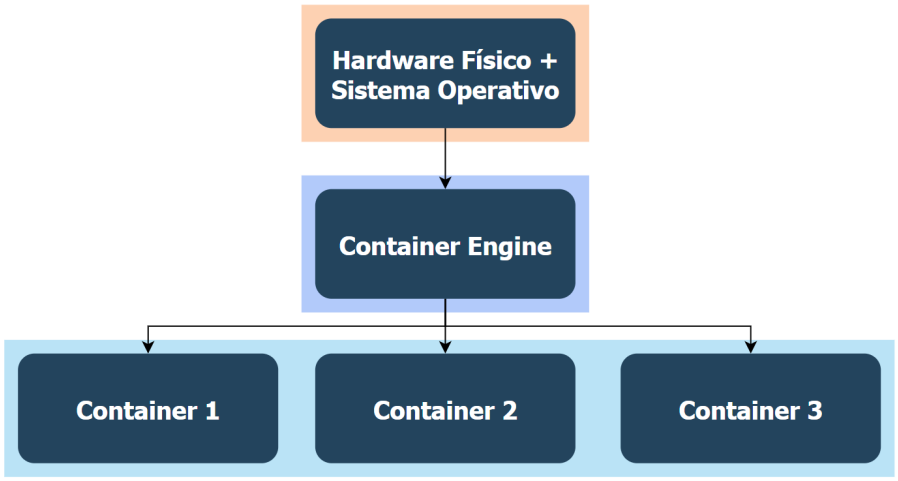

FIGURA 2.5: Arquitetura básica de um ambiente de *containerization*.

Docker

O Docker é uma plataforma *open-source* que permite desenvolver, executar e distribuir aplicações facilmente, através da utilização de *containers* (Rad, Bhatti e Ahmadi 2017). O conceito de *containers* já existia há vários anos antes do aparecimento do Docker, particularmente no universo Unix. No entanto, foi esta ferramenta que impulsionou a utilização de *containers* para o palco principal do mundo do desenvolvimento de *software*.

Lançado em regime *open-source* em 2013, o Docker melhorou as tecnologias de *containerization* existentes no Linux em várias maneiras, particularmente através da utilização de imagens portáteis e de uma UI amigável para o utilizador. Uma imagem Docker é um ficheiro com as instruções necessárias para gerar uma versão executável de uma aplicação, e os *containers* são as instâncias criadas a partir dessas imagens.

A ferramenta Docker é composta por dois componentes principais: o Docker *Engine*, responsável pela criação e gestão dos *containers*, e o Docker *Hub*, uma plataforma na *cloud* para alojar e distribuir imagens Docker (Mouat 2015).

LXC

LXC, também conhecido por Linux Containers, é uma tecnologia ao nível do *kernel* que consegue executar uma variedade de processos, cada um no seu ambiente isolado (Scheepers 2014). Foi a primeira grande tecnologia de *containerization* disponível, e foi a tecnologia precedente do Docker. Contudo, ainda hoje tem uma comunidade ativa que utiliza esta ferramenta.

O LXC é composto por três componentes principais: o `lxc`, o que é o *runtime* da ferramenta, o `lxd`, que é um *daemon* escrito na linguagem Go e é responsável por gerir os *containers* e imagens, e o `lxfuse`, responsável por gerir o sistema de ficheiros (Doerrfeld 2019).

O LXC tem uma principal diferença em relação ao Docker, que é o facto de permitir que vários processos possam ser executados em cada *container*, enquanto que o Docker está desenhado para executar apenas um processo em cada *container* (Kisller 2020).

2.4.4 Orquestração de *Containers*

Atualmente no mundo do desenvolvimento de *software*, a utilização de *containers* é mais usual do que nunca antes foi, sendo agora considerado como um padrão na indústria. Dessa forma, torna-se imperativo a necessidade de ferramentas e tecnologias responsáveis pela orquestração de *containers*.

A orquestração de *containers*, como já mencionado previamente neste documento, é a gestão dos vários estados do ciclo de vida de um *container*, desde a sua criação à sua destruição. É possível utilizar a orquestração de *containers* para uma variedade de cenários, tais como (Eldridge 2018):

- Provisionamento de *containers*.
- Redundância e disponibilidade de *containers*.
- Escalonamento dos *containers*, tanto para cima como para baixo, dependendo do nível de carga da aplicação.
- Alocação de recursos entre *containers*.

Dependendo da ferramenta de orquestração de *containers* utilizada, vai existir um ficheiro de configuração com as definições de escalonamento. Essa ferramenta vai depois monitorizar os *containers* e vai capturar métricas pré-definidas, como, por exemplo, a utilização de processador ou memória. Com base nas definições do ficheiro de configuração, a ferramenta vai tratar de provisionar ou destruir *containers*, com base no nível de carga atual da aplicação.

Kubernetes

Desenvolvido pela Google em 2014, o Kubernetes é uma ferramenta *open-source* de orquestração de *containers*. É atualmente considerada como a ferramenta padrão para este fim na indústria de

software, sendo apoiada por empresas como a Google, Amazon, Microsoft, IBM, Intel, Cisco e Red Hat (Eldridge 2018).

Em termos de funcionalidade, de forma geral, o Kubernetes tem as funcionalidades que já foram mencionadas nos parágrafos introdutórios desta secção. É, no entanto, um sistema com alguma complexidade, contando com vários componentes diferentes na sua arquitetura. A figura 2.6 mostra alguns dos componentes que existem no Kubernetes, e como estes estão relacionados.

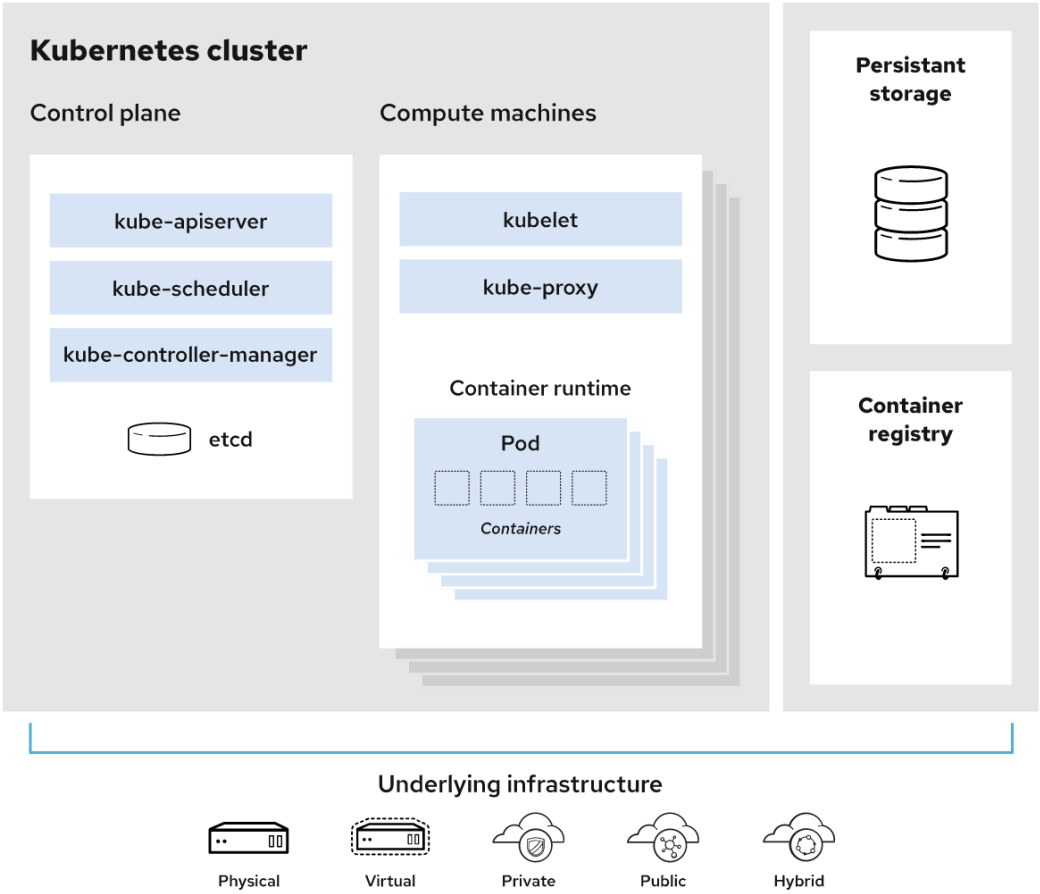

FIGURA 2.6: Diagrama que mostra como vários componentes do Kubernetes se relacionam. Imagem obtida em 2021-01-30, de: `https://www.redhat.com/pt-br/topics/containers/kubernetes-architecture`.

Segue-se uma descrição básica dos conceitos e componentes mais importantes do Kubernetes (Sayfan 2017):

- ***Node*:** É uma máquina, que tanto pode ser física como virtual, cuja responsabilidade é executar *pods*. Cada *node* é constituído por vários componentes do Kubernetes, e é coordenado e gerido pelo *master*.
- ***Cluster*:** Representa um conjunto de *nodes*, sendo que cada *cluster* tem pelo menos um *node master*.
- ***Master*:** É o *node* responsável por planear e provisionar instâncias da aplicação pelos restantes *nodes*. É o *control plane* da figura 2.6, e é constituído por vários componentes do Kubernetes, nomeadamente o *API server*, responsável pela comunicação entre o *master* e

os restantes *nodes*, o *scheduler*, responsável por assignar *nodes* aos *pods* que são criados, e o *controller manager*, responsável por monitorizar o estado dos serviços provisionados.

- ***Pod*:** É uma unidade de trabalho no Kubernetes. Cada *pod* contém um ou mais *containers*. Todos os *containers* de um *pod* estão localizados na mesma máquina e todos possuem o mesmo endereço *Internet Protocol* (IP), podendo assim partilhar recursos livremente.

Docker Swarm

O Docker Swarm é a solução nativa do Docker para a orquestração de *containers*. Foi o produto de um plano para criar um sistema de gestão de *clusters* nativo para o Docker, iniciado em 2014 (Soppelsa e Kaewkasi 2016).

Apesar de o Docker Swarm ser uma solução nativa do Docker, a popularidade e a adoção do Kubernetes pela comunidade fez com o Docker acabasse por suportar oficialmente o Kubernetes em 2017 (Chanana 2017). No entanto, o Docker Swarm ainda continua a ser um produto mantido e oferecido pela empresa, pelo que é ainda uma ferramenta viável para a orquestração de *containers*.

De forma geral, o Docker Swarm é considerada uma ferramenta menos complexa que o Kubernetes, devido a uma arquitetura mais direta e simples de se compreender. No entanto, as arquiteturas de ambas as ferramentas partilham algumas similaridades. A figura 2.7 é um diagrama simples da arquitetura do Docker Swarm.

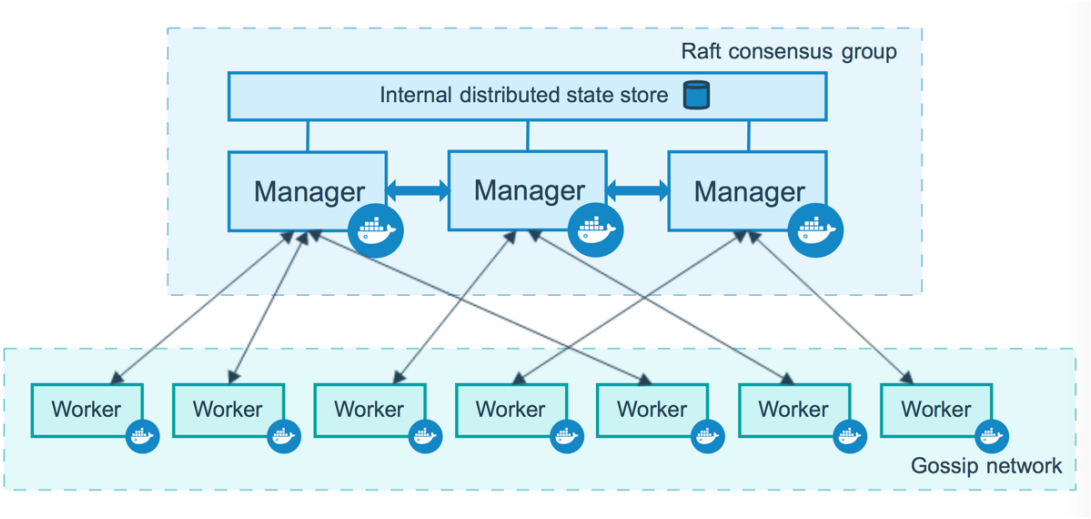

FIGURA 2.7: Arquitetura do Docker Swarm. Imagem obtida em 2021-01-30, de: `https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/`.

Os principais componentes de uma arquitetura do Docker Swarm são (Eldridge 2018):

- ***Swarm*:** É equivalente a um *cluster* em Kubernetes. Representa um conjunto de *nodes*, com pelo menos um *manager node* e vários *worker nodes*, que tanto podem ser máquinas físicas como virtuais.
- ***Manager node*:** Responsável por várias funções quando ocorre um *deployment* de uma aplicação para um *swarm*. É o *manager node* que atribui tarefas aos *worker nodes*, e é também responsável pela gestão do estado do *swarm* onde se encontra.

- ***Worker nodes*:** Responsáveis por executar as tarefas distribuídas pelo *manager node*. Cada *worker node* possui um agente que reporta ao *manager node* o estado das suas tarefas. Isso permite com que o *manager node* consiga fazer uma melhor gestão das tarefas que estão a ser executadas no *swarm*.

Apache Mesos

O Apache Mesos é uma ferramenta ligeiramente mais antiga que o Kubernetes e o Docker Swarm, tendo sido desenvolvida em 2009, na Universidade de Califórnia, Berkeley. Foi utilizado em produção por várias grandes empresas, como Twitter e Airbnb (Kakadia 2015).

A figura 2.8 mostra os componentes principais do Apache Mesos. Neste exemplo, o Hadoop e MPI são duas aplicações que partilham o *cluster*.

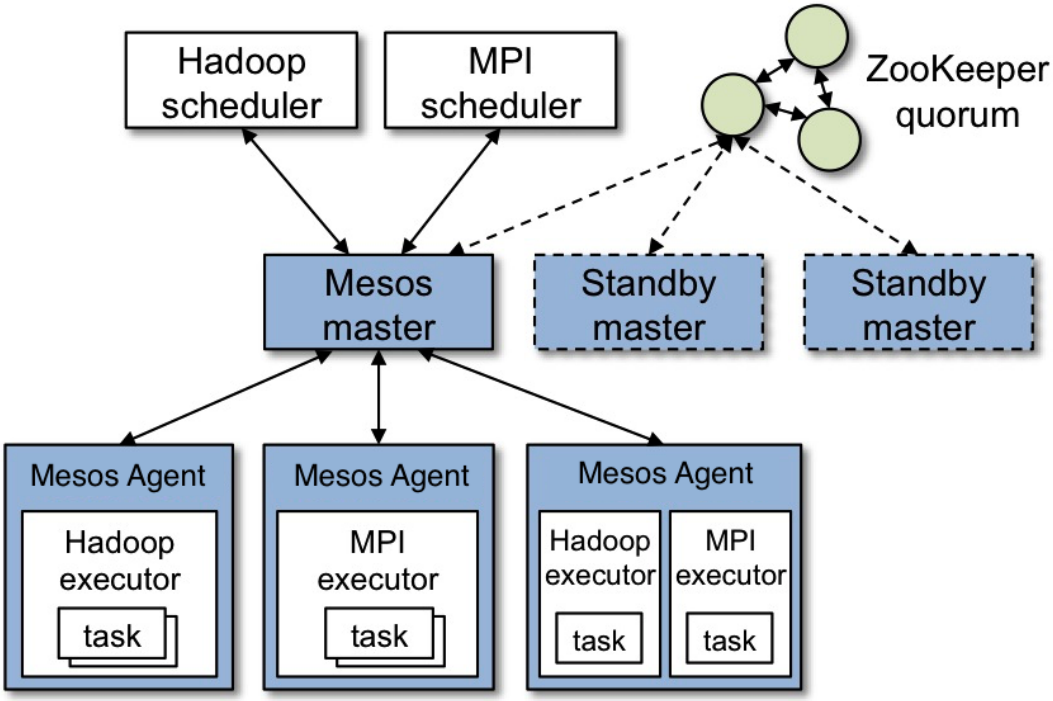

FIGURA 2.8: Arquitetura do Apache Mesos. Imagem obtida em 2021-01-30, de: `http://mesos.apache.org/documentation/latest/architecture/`.

Segue-se uma breve descrição dos componentes mais importantes do Apache Mesos (Kakadia 2015):

- ***Master*:** O *master* é responsável por gerir os *agents* que estão a ser executados em cada *node* do *cluster*.
- ***Agents*:** São os *agents* que executam as tarefas num *cluster* de Apache Mesos, executando as tarefas submetidas pelas *frameworks*. Para além disso, gerem também vários recursos de cada *node*, como processador, memória, portas, entre outros.
- ***Frameworks*:** São as aplicações que estão a ser executadas no Apache Mesos, sendo que cada aplicação é composta por um *scheduler* e um *executor*.

No entanto, o Apache Mesos tem uma importante distinção em relação ao Kubernetes e ao Docker Swarm, na medida em que o Mesos é apenas uma ferramenta de gestão de *clusters*, e não

de orquestração de *containers*. Para o Mesos conseguir capacidades de orquestração de *containers*, é necessária a utilização do Marathon. O Marathon é uma plataforma de orquestração de *containers* que pode ser usada com o Apacha Mesos.

2.4.5 Fornecedores *Cloud*

Como previamente mencionado neste documento, a computação na *cloud* é cada vez mais uma componente fulcral na indústria da informática. Através da computação na *cloud*, as organizações podem consumir recursos computacionais partilhados, em vez de construírem, gerirem e melhorarem a sua própria infraestrutura (Varia, Mathew et al. 2014).

Na Critical TechWorks, particularmente na área de *Infotainment & Interactivity Services*, todos os projetos possuem uma infraestrutura na *cloud*. Assim, é imperativo que a solução desenvolvida neste documento também inclua uma integração com a *cloud*.

A figura 2.9 é o *Magic Quadrant* de 2020, desenvolvido pela empresa de consultoria de Tecnologia da Informação (TI) Gartner, sobre os fornecedores de infraestrutura e serviços na *cloud*.

FIGURA 2.9: O *Magic Quadrant* de 2020 dos fornecedores de infraestrutura e serviços na *cloud*. Imagem obtida em 2021-01-30, de: `https://www.gartner.com/doc/reprints?id=1-1ZDZDMTF&ct=200703&st=sb`.

Este *Magic Quadrant* é composto por quatro categorias:

- ***Leaders*:** Gigantes da indústria que executam com sucesso a sua visão.

- ***Challengers*:** Empresas que dominam um grande segmento de mercado.
- ***Visionaries*:** Fornecedores que têm uma forte visão do mercado futuro.
- ***Niche Players*:** Fornecedores focados num segmento de mercado específico.

Os líderes de mercado têm-se mantido constantes nestes últimos anos, sendo estes a Amazon Web Services (AWS) em primeiro lugar, seguido pelo Microsoft Azure e finalmente pelo Google Cloud. De forma geral, todos estes fornecedores de *cloud* oferecem aos seus clientes funcionalidades semelhantes, tendo no seu catálogo produtos de poder computacional, virtualização, armazenamento, bases de dados, autenticação, entre muitos outros (Pranay Dutta e Prashant Dutta 2019).

Amazon Web Services

Sendo uma das principais e mais populares plataformas de comércio eletrónico no mundo, a Amazon tem uma longa história de utilização de uma infraestrutura de TI descentralizada. Foi isso que permitiu que as suas equipas de desenvolvimento tivessem acesso imediato a recursos computacionais e de armazenamento, aumentado assim significativamente a sua produtividade e agilidade (Varia, Mathew et al. 2014).

Em 2005 a Amazon já tinha investido uma quantidade substancial de fundos numa infraestrutura capaz de suportar uma plataforma da sua dimensão. Assim, a Amazon tomou a decisão de criar oficialmente a AWS, oferecendo dessa forma a outras organizações a possibilidade de usufruírem dessa mesma infraestrutura, e de todos os benefícios inerentes.

A AWS oferecia as vantagens de um plano de preços flexível, em que os seus clientes apenas pagavam aquilo que utilizassem, e pelo tempo que utilizassem. Este aspeto da AWS era atrativo para as organizações, especialmente para aquelas com negócios mais sazonais em que tinham mais tráfego em certas alturas do ano. Para essas empresas, nessas alturas do ano, podiam simplesmente requisitar mais recursos na AWS, voltando depois novamente a retirar esses recursos quando o tráfego normalizasse. Conseguem-se perceber as vantagens deste modelo de negócios, ao invés de a empresa ser obrigada a manter um *data center* capaz de suportar as alturas com mais tráfego, que depois iria ser mal aproveitado durante o resto do ano.

Microsoft Azure

Anunciado em 2008, o Microsoft Azure é a entrada da Microsoft no mundo da computação na *cloud* disponibilizada para clientes. A computação na *cloud* usualmente é dividida em três categorias: *Software as a Service* (SaaS), *Platform as a Service* (PaaS) e *Infrastructure as a Service* (IaaS). O Microsoft Azure, tal como a AWS, tem uma oferta de produtos e serviços nestas três categorias (Collier e Shahan 2015).

Apesar de a AWS continuar a ser o líder de mercado na computação na *cloud*, o Microsoft Azure é opção interessante para organizações que já estejam dentro do universo de produtos da Microsoft, nomeadamente com o Office 365 e o Microsoft Teams, uma vez que o Azure tem uma melhor integração com estes produtos (Pranay Dutta e Prashant Dutta 2019).

Google Cloud

O Google Cloud foi lançado em 2008, e é a proposta da Google para a vasta área de computação na *cloud*. Em termos de funcionalidades, produtos e serviços, tal como acontece com o Azure e a AWS, o Google Cloud também possui ofertas nas áreas de SaaS, PaaS e IaaS.

Devido a um segmento de mercado consideravelmente mais pequeno comparado com o Azure e a AWS, o Google Cloud posiciona-se num mercado mais direcionado a empresas de *big data*, aplicações analíticas, *machine learning* e aplicações nativas na *cloud*. Para além disso, e também devido à sua menor quota de mercado, o Google Cloud é mais flexível em termos de descontos e contratos com empresas (Pranay Dutta e Prashant Dutta 2019).

2.4.6 Gestão da Infraestrutura

Tal como já foi referido na secção 1.4, construir a infraestrutura através de código é considerado uma boa prática. O conceito de ter todos os aspetos da infraestrutura, incluindo o *design*, configuração e dependências necessários, na mesma linguagem padrão, sob forma de *scripts* que possam ser utilizados ou incluídos em processos de automação, é chamado de "infraestrutura como código" (Artac et al. 2017).

Seguindo os princípios da infraestrutura como código, toda a infraestrutura que um projeto necessita deverá estar sob forma de código, idealmente de modo a que nenhum componente tenha que ser criado manualmente. Isso é vantajoso por duas razões:

1. Garante consistência. Se acontecer algum desastre que faça com que a infraestrutura fique indisponível, ou caso seja necessário por outra razão qualquer recriar toda a infraestrutura, basta simplesmente executar os *scripts* e toda a infraestrutura será criada exatamente da mesma forma.

2. Previne erros humanos. Criar infraestrutura manualmente envolve o preenchimento de alguma configuração, sendo que por vezes basta um erro simples para o resultado final não ser o esperado, erro esse que pode depois demorar algum tempo a ser descoberto. Com infraestrutura como código basta configurar tudo bem uma vez.

Terraform

Desenvolvido pela empresa HashiCorp em 2014, o Terraform é uma ferramenta *open-source* para automação de infraestrutura. Hoje em dia, de todas as ferramentas disponíveis para provisionar infraestrutura como código, o Terraform é a mais popular, sendo utilizada por empresas como a Uber, Slack e a Twitch (Stackshare 2020).

As ferramentas capazes de realizar infraestrutura como código, de forma geral, podem ser divididas em duas categorias: ferramentas de gestão de configuração e ferramentas de orquestração. Ao contrário de ferramentas como o Ansible e o Chef, que são mencionadas nas secções seguintes e que são consideradas ferramentas de gestão de configuração, o Terraform é uma ferramenta de orquestração. A diferença entre estes dois tipos de ferramentas é que as ferramentas de gestão de configuração são capazes de instalar e gerir *software* em servidores existentes, enquanto que as ferramentas de orquestração são desenhadas para provisionar os próprios servidores. No entanto, é relevante mencionar que estas categorias não são mutuamente exclusivas, podendo as ferramentas de gestão de configuração também ter alguma capacidade de aprovisionamento, e as ferramentas de orquestração alguma capacidade de gestão de configuração (Brikman 2016).

O Terraform utiliza a sua própria linguagem, o Hashicorp *Configuration Language* (HCL). É uma linguagem relativamente simples, desenhada para ser facilmente interpretada por humanos, mas ao mesmo tempo oferendo um bom leque de funcionalidades e controlo.

Em termos de compatibilidade, o Terraform é compatível com a grande maioria dos fornecedores *cloud* mais utilizados, podendo por isso ser utilizado em quase todos os projetos de *software*. Outra vantagem que o Terraform tem é que não necessita que seja instalado nenhum cliente

na infraestrutura que vai gerir. Basta simplesmente instalar o Terraform na máquina local do programador, por exemplo, ativar o *plugin* do fornecedor *cloud* utilizado, definir algumas variáveis de configuração para o Terraform se conectar à infraestrutura, e a partir desse ponto o Terraform está pronto para tratar ele mesmo do resto do trabalho. O executável do Terraform vai-se conectar diretamente ao respetivo fornecedor *cloud* e vai invocar as APIs necessárias para provisionar e configurar toda a infraestrutura necessária (Nayak 2019).

O Terraform tem duas fases de execução: a fase do planeamento, que é onde o Terraform executa todo o código em modo de "simulação", onde lista todas as alterações que vão ser efetuadas mas sem as fazer na realidade, e a fase da aplicação, que é onde o Terraform vai de facto aplicar todas essas alterações. Este mecanismo existe como uma medida de precaução, para que não sejam aplicadas medidas incorretas de forma acidental.

O estado da infraestrutura provisionada pelo Terraform é guardado num ficheiro com a extensão `.tfstate`, e todas as alterações efetuadas posteriormente pelo Terraform vão ser validadas contra esse ficheiro. Assim, o Terraform sabe sempre quais os componentes que precisam de ser alterados e quais os que se mantêm constantes. No entanto, qualquer alteração realizada fora do Terraform vai criar alguns conflitos na próxima vez que os *scripts* Terraform forem executados.

Ansible

O Ansible foi desenvolvido em 2012, tendo sido em 2015 adquirido empresa Red Hat. É uma ferramenta *open-source* de aprovisionamento, gestão de configuração e *deployments* de aplicações, permitindo assim a infraestrutura como código.

O Ansible, como mencionado na secção anterior, é considerado como uma ferramenta de gestão de configuração. Contudo, pode também ser utilizado para aprovisionamentos. A utilização de uma única ferramenta para gestão de configuração e para aprovisionamento pode ser benéfico em certas situações, devido à acrescida simplicidade dos processos envolvidos (Hochstein e Moser 2017).

Os ficheiros que vão conter os *scripts* para serem executados pelo Ansible são denominados de *playbooks*. A sintaxe de um *playbook* é na linguagem YAML, sendo por isso facilmente interpretável por humanos.

Por fim, e tal como o Terraform, o Ansible também não requere nenhum agente ou outra dependência especial na máquina que vai gerir. Todas as comunicações e ações efetuadas pelo Ansible apenas requerem que a máquina que vai gerir tenha *Secure Socket Shell* (SSH) e Python instalados, que são bibliotecas consideradas padrão hoje em dia, especialmente se a máquina for baseada em Unix.

Chef

Chef é uma ferramenta de infraestrutura como código, desenvolvida em 2009. Em termos de funcionalidades, é particularmente semelhante ao Ansible, sendo também considerada uma ferramenta de gestão de configuração.

Continuando as referências culinárias derivadas do próprio nome, no Chef os ficheiros de configuração são chamados de *cookbooks*. Estes *cookbooks* são as "receitas" da configuração que vai ser aplicada, e são escritos na linguagem Ruby (Marschall 2015).

2.4.7 CI/CD

Nestes últimos anos tem-se verificado uma alteração de paradigma na indústria de *software*, nomeadamente em relação às equipas e às suas funções. Enquanto que antes era normal haver várias equipas separadas, cada uma com a sua especialidade, hoje em dia a tendência tem sido combinar essas competências na mesma equipa. Três áreas em particular foram afetadas nesta alteração: desenvolvimento, *Quality Assurance* (QA) e operações.

Antes, estas funções estavam separadas. As funções da equipa de desenvolvimento eram apenas desenvolver o produto em questão. A equipa de QA realizava testes nesse produto, de forma a testar a sua qualidade e as suas funcionalidades. Por fim, depois de o produto ter sido lançado para o público, a equipa de operações fornecia suporte operacional ao produto, tratando e endereçando todos os problemas levantados pelos utilizadores do produto.

O conceito que veio juntar todas estas competências na mesma equipa é o *DevOps*, conceito esse cujo nome é derivado da junção entre *development* e *operations*. A missão do *DevOps* é garantir o desenvolvimento e lançamento de *software* com qualidade, de forma rápida e flexível. Para conseguir isso, são utilizados mecanismos automáticos de desenvolvimento, lançamento, testes e monitorização (Ebert et al. 2016). Estes mecanismos automáticos são conseguidos com CI/CD.

O conceito de CI/CD está dividido em duas partes:

- CI: *Continuous integration* é a filosofia de desenvolver *software* com incrementos pequenos e submeter essas alterações nos repositórios de controlo de versões. Nesta fase é também boa prática implementar *continuous testing*, onde cada alteração submetida lança uma *pipeline* automática para correr testes sobre essas alterações, garantido assim a integridade do código.
- CD: *Continuous delivery* começa onde o *continuous integration* termina. Depois de as alterações terem sido feitas, as *pipelines* de *continuous delivery* automatizam o lançamento dos produtos para os vários ambientes de desenvolvimento (tais como, por exemplo, teste, *End-to-end* (E2E) e produção).

Jenkins

Esta ferramenta de CI/CD foi desenvolvida inicialmente sob o nome Hudson, em 2005. Em 2009, após a aquisição da empresa Sun pela Oracle, o código do Hudson foi também herdado durante esse processo. No entanto, em 2011, discórdias entre a Oracle e a comunidade *open-source* atingiram o ponto de ruptura, fazendo com que a comunidade fizesse um *fork* do projeto Hudson, criando assim o Jenkins. O Hudson continou a ser mantido pela Oracle, enquanto que o Jenkins foi desenvolvido a partir desse momento pela comunidade *open-source* (Smart 2011).

Desde então que o Jenkins tem vindo a ganhar mais popularidade na comunidade, sendo atualmente a ferramenta de CI/CD com maior quota de mercado. As razões para isso devem-se à flexibilidade do Jenkins, podendo ser utilizado em projetos baseados em várias linguagens, deste Ruby, PHP, Java, entre outras. Depois, tem uma boa interface de utilizador, fornecendo assim uma experiência simples e intuitiva. Por fim, é uma ferramenta com bastante poder e extensibilidade, podendo ser utilizada para realizar uma grande variedade de tarefas, especialmente considerando a generosa oferta de *plugins* disponíveis e o tamanho da comunidade de utilizadores.

A forma mais comum de utilizar o Jenkins é criando uma *pipeline*, que é um documento com as tarefas que o Jenkins deve executar. O documento que contém as instruções da *pipeline* é denominado `Jenkinsfile`, e pode ser escrito com duas sintaxes diferentes: a sintaxe declarativa ou a sintaxe *scripted*. A sintaxe mais atual é a declarativa, e é geralmente a sintaxe recomendada.

CircleCI

CircleCI é uma ferramenta popular de CI/CD, utilizado por empresas como Facebook, Spotify e GoPro. Em termos de funcionalidades é relativamente semelhante ao Jenkins, sendo também uma ferramenta para que automatiza as várias fases do ciclo de vida do *software*.

É possível integrar o CircleCI com o GitHub ou o Bitbucket. Depois do repositório ter sido submetido e aprovado em `https://circleci.com/`, qualquer atualização nesse repositório vai despoletar validações automáticas num *container* ou máquina virtual. O CircleCI também tem um sistema de notificações para avisar o utilizador quando os *jobs* terminarem, e se terminaram em sucesso ou em falha (CircleCI 2020).

TeamCity

O TeamCity é um servidor de CI desenvolvido e mantido pela empresa JetBrains, conhecida pelos seus *Integrated Development Environment*s (IDEs), tais como o IntelliJ IDEA para Java e o PyCharm para Python.

Tal como as outras ferramentas que foram faladas nesta secção, o TeamCity consegue detetar alterações nos repositórios de controlo de versões, e automaticamente lançar *jobs* para executar uma variedade de tarefas, como compilar o código, executar testes unitários, *deploying* da aplicação ou guardar os artefactos gerados (Mahalingam 2014).

As principais funcionalidades do TeamCity incluem:

- Suporte para projetos escritos em várias tecnologias diferentes.
- Uma grande variedade de *plugins*, desenvolvidos tanto pela JetBrains como pela comunidade.
- Uma API REST, que torna possível que a equipa de desenvolvimento execute ações remotamente, como, por exemplo, lançar *jobs* e verificar o estado de *jobs* existentes.
- Um *dashboard* com uma boa UI, que fornece uma boa visibilidade sobre o estado atual do servidor.

2.5 Sumário

Neste capítulo foi abordado o contexto da empresa, foram analisados alguns trabalhos relacionados com o problema em questão, e foram exploradas algumas tecnologias relevantes para o cumprimentos de todos os requisitos desta solução. Aqui incluem-se não só as tecnologias que foram selecionadas para o projeto, mas também outras tecnologias que foram consideradas.

Foram exploradas tanto ferramentas com capacidades de geração de código, como também tecnologias para cada componente individual do projeto. Com o levantamento efetuado, conseguiu-se obter uma boa base das funcionalidades e possibilidades existentes no mercado, informação essa que é fundamental para o desenvolvimento de qualquer projeto de *software*.

Capítulo 3

Análise

Neste capítulo é realizada uma análise mais aprofundada à solução, com o objetivo de se obter informações necessárias para as fases seguintes de *design* e implementação.

O capítulo está organizado em três secções. Na primeira secção é efetuada uma análise de requisitos, onde são descritos os requisitos funcionais e não funcionais desta solução. De seguida, na segunda secção, é realizada a análise de valor, onde é feita a análise do problema a ser resolvido, assim como a proposta de valor da solução. Finalmente, na terceira e última secção, é feita uma análise de alternativas das ferramentas discutidas no capítulo 2, de forma a selecionar as ferramentas e tecnologias utilizadas nesta solução.

3.1 Análise de Requisitos

Nesta secção é realizada uma análise aos requisitos identificados para esta solução, incluindo os requisitos funcionais e não funcionais.

3.1.1 Requisitos Funcionais

Os requisitos funcionais focam-se no comportamento básico do sistema. Estes requisitos descrevem as funções que o sistema deve ser capaz de executar (Lightsey 2001).

Os requisitos funcionais da solução são os seguintes:

1. **Criar as componentes *back-end* e *front-end***

   Sendo esta solução apontada para um projeto *full-stack*, esta deve criar um repositório com o código *back-end* e outro repositório com o código *front-end*. Uma vez que a solução deve ser genérica, podendo ser aplicada para qualquer projeto de *software*, estes repositórios vão ter apenas uma estrutura esqueleto com alguns exemplos de código, servindo assim como um *template* e devendo ser posteriormente incrementado pela equipa de desenvolvimento.

   As tecnologias destas componentes vão ser avaliadas num capítulo futuro deste documento. No entanto, com base nas tecnologias utilizadas, a estrutura de ficheiros gerada pela solução deve estar coerente com as melhores práticas da respetiva tecnologia, sendo assim expectável que a equipa de desenvolvimento mantenha a estrutura que foi gerada.

2. **Estabelecer comunicação entre as componentes *back-end* e *front-end***

   Num projeto *full-stack*, é naturalmente importante que o *back-end* e *front-end* se encontrem interligados. Quer isto dizer que no *front-end* deverá ser possível interagir com a aplicação, realizando as operações permitidas pelo *back-end*.

Assim, durante a geração do código *template* das componentes *back-end* e *front-end*, esta solução deve também automaticamente criar uma comunicação entre estas duas componentes, tornando assim este projeto verdadeiramente *full-stack*.

3. **Criar *templates* para testes**

   Qualquer projeto de software necessita de vários tipos de testes para ser possível assegurar a sua qualidade (Luo 2001). Para além disso, os testes são também importantes para permitirem uma iteração segura e confiável do código do projeto. Por exemplo, caso a equipa de desenvolvimento crie funcionalidades, ou altere funcionalidades previamente existentes, uma boa coleção de testes vai permitir avaliar rapidamente se o código estiver a funcionar da forma que é expectável.

   Assim, esta solução deve também criar código *template* para testes unitários, de integração e funcionais, tanto para a componente *front-end* como *back-end*. Desta forma, a equipa vai ter a estrutura dos ficheiros de testes já definida, bem como alguns modelos de testes que poderão usar como exemplos para os testes futuros.

4. **Utilizar tecnologias de *containerization* das aplicações**

   A utilização de *containers* visa resolver o antigo problema de software ter comportamentos diferentes nos diversos ambientes. Por exemplo, no ambiente local do desenvolvedor a aplicação poderia ter o comportamento correto e expectável, contudo, depois de ter sido lançada no ambiente de produção, a aplicação teria um outro comportamento diferente.

   Este problema pode ocorrer devido às diferenças existentes no software envolvente. Quer isto dizer que o código da aplicação pode ser exatamente igual nos dois ambientes, no entanto, podem existir diferenças em software dependente da aplicação. Um ambiente pode ter uma versão de uma certa biblioteca que a aplicação necessita, enquanto que outro ambiente pode ter uma versão distinta dessa mesma biblioteca. Ou então podem existir diferenças ao nível da topologia de rede, ou os ambientes podem ter políticas de segurança diferentes, entre outras disparidades.

   Um *container*, tal como o nome indica[1], vai encapsular a aplicação e todas as suas dependências, bibliotecas, binários e outros ficheiros de configuração. Dessa forma, o *container* vai conter tudo o que a aplicação necessita para executar, criando assim um ambiente estável e consistente que vai produzir sempre os mesmos resultados.

   Em função do que foi descrito, a solução produzida deve utilizar tecnologias de *containerization* para encapsular as componentes *back-end* e *front-end* em *containers*, de forma prevenir os problemas mencionados.

5. **Utilizar ferramentas de orquestração de *containers***

   Tendo as aplicações encapsuladas em *containers*, seria útil ter também uma ferramenta de orquestração de *containers*.

   A orquestração de *containers* automatiza vários aspetos do ciclo de vida de um *container*, como o processo de *deployment*, a sua gestão e o escalonamento (Hat 2020). Outra vantagem em utilizar uma ferramenta de orquestração a gerir o ciclo de vida de *containers* é que também habilita a equipa de desenvolvimento a integrar esse processo mais facilmente nos seus processos de CI/CD.

[1] *container* significa "contentor" em português

6. **Utilizar ferramentas de criação, manutenção e destruição da infraestrutura**

   Este projeto deverá ter uma infraestrutura na *cloud* que suporte a aplicação. O fornecedor de infraestrutura *cloud*, bem como as componentes de infraestrutura necessários, vão ser analisados num capítulo posterior.

   Contudo, independente das escolhas selecionadas, a infraestrutura deverá ser construída com código. Construir infraestrutura com código é sempre uma boa prática, pois é uma forma estável de se assegurar sempre o mesmo resultado e que a infraestrutura seja sempre construída da mesma maneira. Para além disso, evitam-se também possíveis erros humanos que poderiam ocorrer caso se criasse a infraestrutura manualmente.

   Assim, esta solução deve também criar *scripts* para construir, manter e destruir a infraestrutura necessária da aplicação. A infraestrutura criada pela solução vai servir apenas como uma base para suportar o esqueleto da aplicação, por isso é expectável que a equipa tenha depois de modificar a infraestrutura à medida que a aplicação vá crescendo. Todavia, os *scripts* gerados vão ser uma boa ajuda no início do projeto, fornecendo um exemplo que a equipa de desenvolvimento pode depois iterar com facilidade.

7. **Utilizar mecanismos de CI/CD**

   Finalmente, para orquestrar e automatizar todas as fases do ciclo de vida das aplicações, é necessário criar mecanismos de CI/CD. Esses mecanismos iriam consistir em criar *pipelines* para efetuar diversas tarefas, como por exemplo executar os testes ou realizar um *deployment*.

   Também é possível automatizar a execução das *pipelines*. Por exemplo, poder-se-ia executar uma pipeline que executasse todos os testes da aplicação em cada *commit* efetuado. Dessa forma, a equipa de desenvolvimento teria a confiança que as suas alterações no código não estão a gerar resultados inesperados, pois iriam conseguir ver se os seus *commits* estão a quebrar algum teste.

   A solução deve criar todos estes mecanismos de CI/CD, tornando este projeto *deploy-ready*.

3.1.2 Requisitos Não Funcionais

Definem-se como requisitos não funcionais aqueles que não estão relacionados com as funcionalidades do produto, mas sim com o uso do produto em termos de desempenho, usabilidade, segurança e qualidade (Chung e Prado Leite 2009). São requisitos que, embora não correspondam ao propósito do produto propriamente dito, melhoram a experiência de utilização.

1. **Modularidade e baixo acoplamento**

   Idealmente, a solução deverá ser o mais modular possível, sendo que os diversos componentes da solução deverão ter baixo acoplamento. O propósito disso é oferecer alguma flexibilidade à equipa de desenvolvimento.

   Posteriormente neste documento será feita uma análise de alternativas e uma seleção das tecnologias a serem utilizadas. Contudo, poderá haver uma nova equipa no futuro que queira utilizar esta solução e que não vá usar as mesmas tecnologias que foram selecionadas. Ou então, pode até existir uma equipa que já possua um *back-end* e que queira apenas utilizar esta solução para gerar um *front-end* e a respetiva infraestrutura e *pipelines* CI/CD.

Desta forma, esta solução tem de ser modular de forma a os seus componentes serem o mais independente possíveis, de forma a poderem ser substituídos no futuro caso surja essa necessidade.

2. **Ser completamente personalizável**

   Este requisito dita que todos os aspetos da solução devem ser controlados pelas equipas de desenvolvimento. A razão principal é tentar construir uma solução que possa ser iterada e incrementada ao longo do tempo, pois as necessidades tecnológicas que as equipas sentem hoje podem não ser as mesmas que sentem no futuro.

   Assim sendo, a solução deverá ser construída de forma a que depois as equipas de desenvolvimento tenham acesso a todo o código envolvido, e tenham a flexibilidade de efetuarem as alterações que desejarem.

3.2 Análise de Valor

No livro "*Techniques of Value Analysis and Engineering*", Miles (1989) começa por abordar o conceito de análise de valor como algo que está presente na realidade humana praticamente desde o seu início. É algo que está inerente na própria natureza humana, na sua capacidade de resolver problemas, tendo sido continuamente desenvolvida com o passar das gerações.

É indiscutível que o sucesso da espécie humana se deveu essencialmente à sua inteligência. Não sendo seres particularmente fortes ou com grandes armas naturais, os primeiros humanos tiveram de recorrer às suas capacidades de resolução de problemas para prosperar. Como não tinham garras naturais para caçar animais, tiveram de criar armas para esse efeito. E, inconscientemente, aplicaram métodos de análise de valor para conseguir chegar à conclusão que armas feitas com madeira e pedra eram mais eficazes do que armas feitas com materiais menos resistentes. Por outras palavras, a madeira e a pedra eram materiais que ofereciam mais valor para a resolver o seu problema de caçar animais.

Os princípios de análise de valor aplicados nessa altura não são particularmente diferentes dos princípios aplicados hoje em dia nas mais variadas áreas do mundo moderno. A principal diferença é que atualmente esse conceito já se encontra bem estudado, existindo assim métodos bem definidos de como o aplicar eficazmente.

Numa definição mais ajustada ao mundo empresarial moderno, a análise de valor é um sistema concebido para identificar de forma eficiente custos desnecessários que não produzam valor para os seus clientes, através da utilização de certas técnicas e métodos (Miles 1989). É, portanto, um conjunto de medidas cujo objetivo é assegurar as funções necessárias de um produto com o mínimo custo, sem que isso ponha em causa a sua qualidade, fiabilidade e desempenho.

3.2.1 Análise do Problema

Realizar uma análise de um problema, especialmente numa perspetiva de análise de valor, envolve perceber as inovações que devem ser feitas para resolver esse problema. Essas inovações podem implicar o desenvolvimento de novas ideias, ou então simplesmente uma evolução ou melhoria de ideias existentes.

Processo de Inovação

Segundo Koen et al. (2001), o processo de inovação compreende três fases:

- *Front End of Innovation* (FEI), também conhecido por *Fuzzy Front End* (FFE). Estes dois termos são utilizados neste documento de forma intercambiável.
- *New Product and Process Development* (NPPD), ou *New Product Development* (NPD). Também estes termos são utilizados neste documento de forma intercambiável.
- Comercialização.

Uma nota relevante para evitar possíveis confusões na leitura desta secção, visto este documento tratar-se de uma dissertação no campo da informática, é que o *front-end* mencionado no FFE não se trata do mesmo *front-end* da área do desenvolvimento de *software*, isto é, da parte das aplicações com a qual os utilizadores interagem. Numa perspetiva de análise de valor e de inovação, tal como representado pela figura 3.1, Koen et al. (2001) definem *front-end* como as atividades que antecedem o NPPD, que é um processo mais formal e mais bem estruturado.

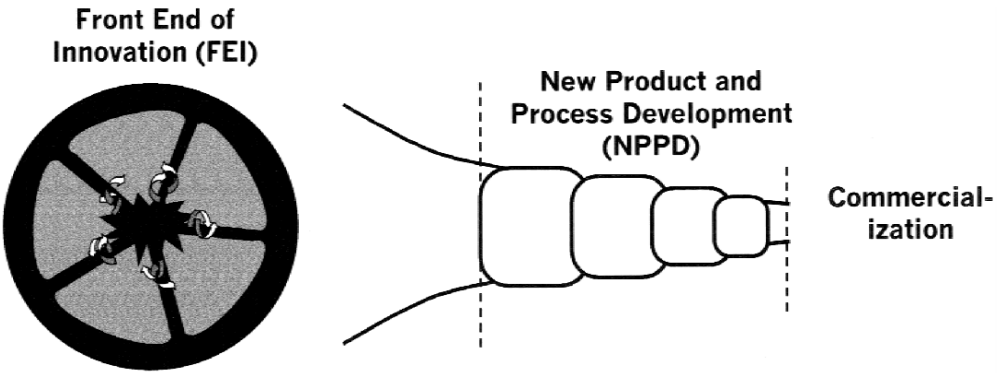

FIGURA 3.1: O processo de inovação, tal como definido por Koen et al. (2001).

A fase do FEI é a fase mais experimental e incerta do processo de inovação. Foi dessa incerteza que surgiu o termo "*fuzzy*" (difuso) em *Fuzzy Front End*. É a fase onde o futuro contexto de uma inovação está mais ambíguo e complexo, e onde é imperativo que a empresa tenha perceções e comentários dos seus clientes, bem como informações e dados orientados ao futuro, para que possam definir com mais confiança as decisões de *design*. Nesta fase são identificadas possíveis oportunidades para o problema em questão, sendo que a partir dessas oportunidades são geradas ideias que acrescentem valor. Dessas ideias são selecionadas as melhores, e é criado o conceito que será implementado na fase seguinte, a NPPD.

A fase do NPPD é mais estruturada do que a fase do FEI. Nesta fase é iniciado o processo de converter o conceito delineado na fase anterior num produto, desde a sua conceção, testes, iterações, remodelações, até, eventualmente, se chegar a um *design* final.

A tabela 3.1 resume as principais diferenças entre as fases FEI e NPPD (Koen et al. 2001).

Tabela 3.1: Diferenças entre o FEI e o NPPD.

	FEI	**NPPD**
Trabalho	Experimental, por vezes caótico.	Estruturado, disciplinado e orientado a objetivos.
Comercialização	Data não previsível.	Data definível.
Financiamento	Variável.	Orçamentado.
Receitas	Incertas.	Realistas.
Atividade	Individual ou em equipa.	Realizada em várias equipas de desenvolvimento de produtos.

Por fim, vem a fase da comercialização, que é a fase mais simples deste processo. Depois de existir uma versão final e aprovada do produto, o objetivo desta fase, tal como indicado pelo próprio nome, é comercializar o mesmo, vendendo-o aos seus clientes.

New Concept Development

Tal como mencionado na secção anterior, a fase do *front-end* no processo de inovação é a fase mais incerta, o que originou o termo *Fuzzy Front End*. No entanto, na opinião de Koen et al. (2001), o termo FFE sugeria que esta fase do processo de inovação era misteriosa, implicando assim demasiada ambiguidade. Foi com o objetivo de corrigir essa situação que surgiu o termo *Front End of Innovation*. Para além disso, numa tentativa de simplificar ainda mais a fase do FEI, Koen et al. (2001) desenvolveram um novo modelo, denominado *New Concept Development* (NCD).

A figura 3.2 representa uma ilustração do modelo NCD. Como se pode ver pela figura, este modelo é constituído por três partes principais:

- A parte interna do círculo, consistindo nos cinco elementos principais do FEI.
- O *engine*, ou motor, que é o que impulsiona os cinco elementos principais do FEI, e é alimentado pela liderança e cultura da organização.
- A parte mais exterior do círculo, representado os fatores de influência. Estes fatores incluem as capacidades organizacionais, a estratégia de negócio, o mundo exterior (clientes, concorrência, etc.), e as componentes científicas que vão ser utilizadas.

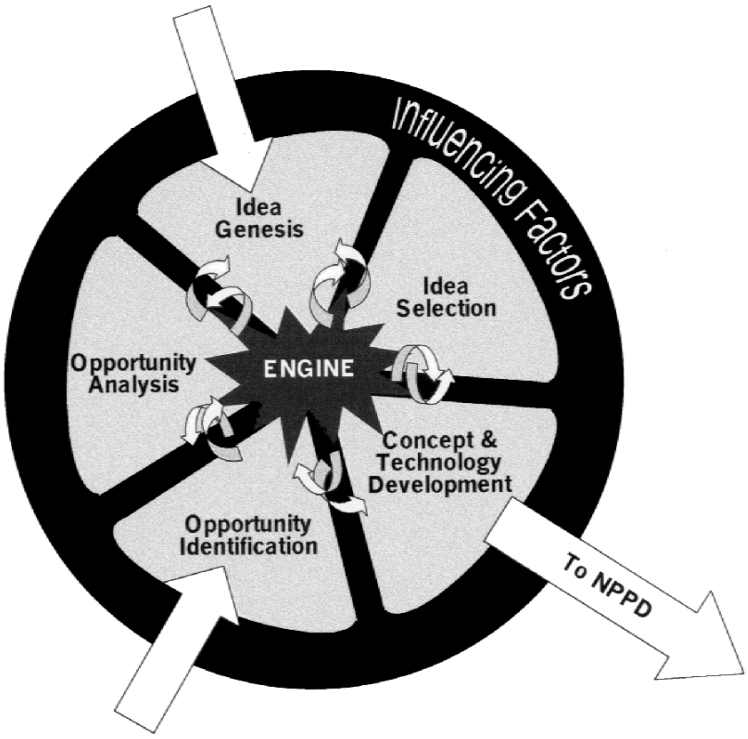

FIGURA 3.2: Ilustração do modelo *New Concept Development* (Koen et al. 2001).

O objetivo do modelo NCD é remover a confusão do *"Fuzzy" Front End*, criando uma estrutura comum e bem definida dos componentes principais do FEE, para assim evitar possíveis erros nesta fase.

De seguida é feita uma análise do problema em questão, utilizando para isso os cinco elementos principais do FEI (Koen et al. 2001):

1. **Identificação de oportunidade**

   Nesta fase é onde são identificadas as oportunidades que a organização quer explorar. Normalmente é uma fase direcionada pelos objetivos da organização em relação ao problema que querem solucionar. Essas oportunidades podem ter qualquer ordem de grandeza, podendo tanto surgir sob a forma de uma nova direção para o negócio, como uma pequena atualização para um produto existente.

   No caso desta dissertação, a principal oportunidade identificada foi a falta de automação nas tarefas iniciais de um projeto de *software*, tal como explorado na secção 1.3. A automação dessas tarefas teria o efeito de reduzir o tempo necessário para estabelecer uma base de um novo projeto, permitindo assim que as equipas de desenvolvimento comecem a produzir valor mais rapidamente.

2. **Análise da oportunidade**

   O objetivo desta fase é extrair informação adicional sobre a oportunidade identificada na fase anterior, com o objetivo de a transformar numa ideia de negócio específica. Para isso podem ser utilizadas técnicas como discussão em grupo, estudos de mercado e/ou experiências científicas. Contudo, o esforço gasto na obtenção dessa informação é proporcional à grandeza da oportunidade em si, e do desenvolvimento futuro de que vai necessitar.

Para o caso específico da solução descrita neste documento, não foi necessário uma análise particularmente extensa sobre a oportunidade, uma vez que visa solucionar um problema experienciado por todas as equipas da área de *Infotainment & Interactivity Services* da Critical TechWorks, no início dos seus respetivos projetos.

3. **Génese da ideia**

   Esta fase tem como objetivo o crescimento e a maturação da oportunidade identificada numa ideia concreta. Essa ideia pode passar por diversas iterações e mudanças à medida que é estudada, discutida e desenvolvida. A eficácia desta fase muitas vezes é aumentada se existir contacto com os futuros clientes ou utilizadores do produto final.

   Neste projeto, depois de terem sido identificadas várias oportunidades de melhoramento no começo de um novo projeto, surgiu assim a ideia de desenvolver uma solução que automatizasse todos esses processos. Essa ideia foi desenvolvida em conjunto com a Critical TechWorks, com base nas necessidades das suas equipas.

4. **Seleção da ideia**

   As organizações, dependendo do seu tamanho, podem ter várias ideias de produtos ou processos, e muitas vezes no mesmo período temporal. Dessa forma, o desafio torna-se em escolher quais as ideias que vão ser para avançar, isto se não for possível avançar com todas as ideias. Dessa forma, o propósito desta fase é fazer uma seleção das ideias que irão ser devidamente desenvolvidas.

   No caso deste projeto esta foi uma fase simples, uma vez que só havia uma ideia a ser considerada. Dessa forma, essa ideia foi automaticamente selecionada.

5. **Desenvolvimento do conceito e da tecnologia**

   Por fim, nesta fase a ideia é devidamente estruturada num conceito de negócio, completo com estimativas de potencial de mercado, necessidades dos clientes, requisitos de investimento, avaliações de concorrentes, incertezas tecnológicas, e risco geral do projeto. Contudo, o nível de formalidade vai variar com o caso de negócio.

   Para o caso desta solução, os fatores mais relevantes foram as necessidades dos clientes, sendo que foram essas necessidades que ditaram os requisitos da solução, que se encontram definidos na secção 3.1. As incertezas tecnológicas também foram importantes, algo que foi explorado no último capítulo. Fatores como o potencial de mercado e avaliações de concorrentes não foram considerados, visto tratar-se de uma solução interna para a empresa, e não um produto para ser comercializado. E em relação aos requisitos de investimento e risco geral do projeto, estes também não foram fatores relevantes uma vez que este projeto não tem qualquer tipo de risco para a empresa, visto que o seu desenvolvimento foi realizado num contexto académico fora do horário laboral.

3.2.2 Proposta de Valor

Considera-se que um produto ou serviço tem valor quando este possui um desempenho e um custo apropriados. Ou seja, se um produto tiver um mau desempenho ou um custo demasiado elevado, considera-se que não oferece bom valor. Partindo dessas premissas, chega-se a duas conclusões (Miles 1989):

1. O valor é sempre aumentado com a redução de custos, desde que se mantenha o mesmo desempenho.

2. O valor é aumentado se existirem melhorias no desempenho do produto, desde que o cliente esteja disposto a pagar por essas melhorias.

O desempenho de um produto ou serviço, numa perspetiva de valor, está associado a como este corresponde e serve os desejos dos seus clientes. O custo de produção do produto também tem de permitir que os clientes o possam adquirir a um preço competitivo, mas ao mesmo tempo assegurando que existe uma diferença adequada entre o custo de produção e o custo de venda, permitindo assim a continuidade do negócio (Miles 1989).

O conceito de valor percebido ou considerado[2] surgiu nos anos 90, e desde então que tem sigo sujeito a diversos estudos e pesquisas. No artigo "*The concept of perceived value: a systematic review of the research*", Sánchez-Fernández e Iniesta-Bonillo (2007) demonstram a ambiguidade da definição de valor percebido. Tanto existe literatura que defende que este conceito é uma construção unidimensional que pode ser mensurada ao questionar a clientes o valor que receberam quando adquiriram os produtos, como também existe literatura que defende que na verdade é um conceito multidimensional em que uma variedade de noções distintas devem ser consideradas (tais como preço, qualidade, benefícios, etc.).

Numa tentativa de facilitar a criação de um modelo de negócios, foi desenvolvido o modelo de negócios *canvas*[3], ilustrado na figura 3.3. Este modelo é constituído por nove componentes essenciais de um negócio, e permite obter rapidamente uma noção abrangente do plano de negócio (Muhtaroğlu et al. 2013).

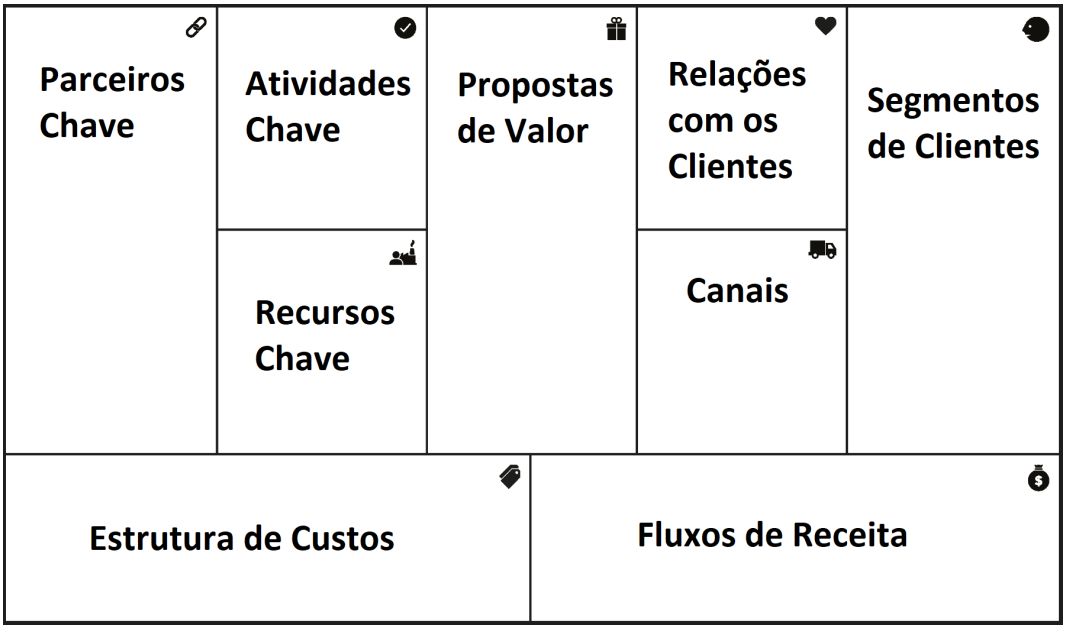

FIGURA 3.3: O modelo de negócios *canvas*.

Apesar de as propostas de valor serem uma das nove componentes do modelo de negócios *canvas*, existe um outro modelo *canvas* especificamente dedicado a este conceito. A figura 3.4 ilustra o modelo *canvas* de proposta de valor. A figura está dividida em duas secções: a secção da proposta de valor e a secção do segmento do cliente. A primeira inclui o valor que o produto ou serviço pode oferecer ao cliente. A última inclui as experiências que o cliente está a ter atualmente, e os problemas que este precisa que o produto resolva (Pokorná et al. 2015).

[2] *perceived value*
[3] *business model canvas*

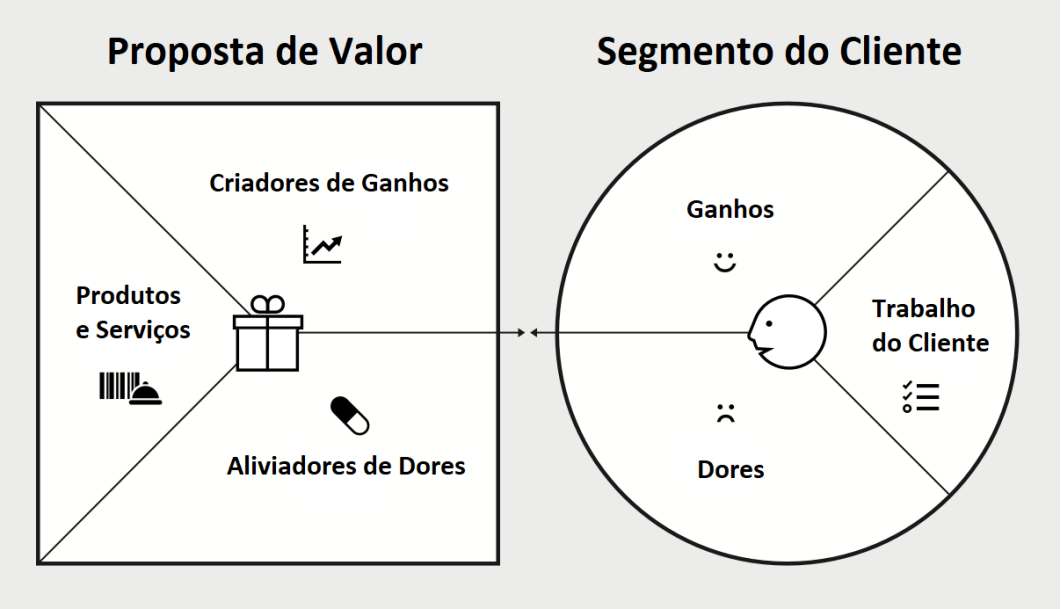

FIGURA 3.4: O modelo *canvas* de proposta de valor.

De forma mais detalhada, a secção da proposta de valor inclui:

- **Produtos e serviços:** Os produtos e serviços que vão ser fornecidos aos clientes para resolverem os seus problemas.
- **Criadores de ganhos:** Aquilo que o produto ou serviço pode oferecer para que os clientes consigam alcançar os seus ganhos.
- **Aliviadores de dores:** A forma como o produto ou serviço pode aliviar as "dores" que os clientes experienciam antes, durante ou depois do trabalho que é efetuado.

A secção do segmento do cliente inclui:

- **Trabalho do cliente:** O trabalho (ou tarefa) que é efetuado pelo cliente.
- **Ganhos:** Os benefícios que iam ajudar o cliente a efetuar o trabalho.
- **Dores:** Os problemas ou dificuldades que o cliente está a encontrar ao executar o trabalho.

Desta forma, a proposta de valor do projeto descrito neste documento é fornecer uma solução de automação das tarefas mais comuns no início de um novo projeto de *software*, sendo esse o trabalho que os clientes (neste caso, as equipas de desenvolvimento da Critical TechWorks) querem solucionar. As dores sentidas pelas equipas são o tempo que essas tarefas normalmente demoram a ser efetuadas, e o facto de serem executadas de forma manual. Assim, a automação dessas tarefas iriam oferecer ganhos às equipas, uma vez iam conseguir executar as tarefas de forma mais rápida e sem risco de erros humanos.

3.3 Análise de Alternativas

Nesta secção é realizada uma análise às tecnologias mencionadas na secção 2.4, com o objetivo de se efetuar uma seleção das tecnologias mais adequadas para o desenvolvimento da solução pretendida, de acordo com os requisitos estabelecidos.

As tecnologias mais importantes para qualquer projeto *full-stack* são as linguagens de programação e *frameworks* do *front-end* e do *back-end*. Enquanto que outras tecnologias, como tecnologias de *containerization* ou de infraestrutura, podem ser alteradas a meio de um projeto, mudar as linguagens de programação e *frameworks* do *front-end* ou *back-end* já é um processo consideravelmente mais complicado, especialmente se o projeto já estiver numa fase mais avançada. Dessa forma, é imperativo que seja feita uma seleção inicial destas tecnologias de forma ponderada, com base nos requisitos do projeto e dos clientes.

Para auxiliar esta fase de análise de alternativas e seleção das tecnologias, existem processos já estabelecidos que têm como objetivo assegurarem que os requisitos dos clientes são devidamente considerados e assegurados. Neste documento foram utilizados os processos *Quality Function Deployment* (QFD) e *Analytic Hierarchy Process* (AHP).

Hoje em dia o QFD é um método bastante utilizado por empresas por todo o mundo, sendo que as necessidades dos clientes são fatores mais relevantes do que nunca, considerando a competição e concorrência que existe em todos os setores do mercado (Zairi e Youssef 1995). O método AHP é um método de análise hierárquica no âmbito das decisões multicritério, cujas opções são priorizadas com base em fórmulas matemáticas (Forman e Gass 2001).

As secções seguintes fornecem uma explicação mais detalhada destes dois processos, bem como uma utilização prática de ambos.

3.3.1 *Front-end*

Para a análise das tecnologias da componente de *front-end*, as alternativas consideradas foram o React, o Angular e o Vue.js, e o método aplicado foi o QFD.

Quality Function Deployment

O QFD, tal como previamente mencionado, é um método que permite a análise de diferentes alternativas, tendo como base os requisitos do cliente. Para isso, este método considera os requisitos do cliente, também denominados como os "*Whats*", e as características que as tecnologias possuem, também denominados como os "*Hows*".

Para utilizar o processo do QFD, é usada a tabela *House of Quality* (HOQ), denominação que ganhou devido à sua forma análoga à de uma casa. Para construir esta tabela, são realizadas as seguintes tarefas:

1. Identificação dos requisitos do cliente ("*Whats*").
2. Identificação das características das tecnologias ("*Hows*").
3. Determinação do peso relativo de cada um dos requisitos do cliente. Este passo deve ser efetuado em parceria com o cliente, uma vez que é este que vai ditar a ordem de importância dos requisitos. Para isso, o cliente vai atribuir um valor a cada requisito de 1 a 5, em que o 1 significa um requisito pouco importante e o 5 um muito importante. Depois, esse valor é transformado em percentagem, obtendo assim o peso relativo.
4. Determinação do nível de correlação entre as características das tecnologias. Esta etapa é realizada no "telhado" triangular do HOQ, sendo que cada par de características pode ter uma correlação positiva, negativa ou pode não ter qualquer correlação.
5. Determinação da direção de melhoria de cada característica. A direção de melhoria indica se seria mais benéfico para o cliente se a característica fosse maximizada ou minimizada.

6. Determinação das relações entre os requisitos do cliente e as características das tecnologias, na parte central do HOQ. Esta etapa envolve determinar o nível de associação de cada par requisito do cliente / característica, sendo que pode ser uma associação forte, moderada ou fraca.
7. Por fim, na parte direita e na parte de baixo do HOQ, são colocadas as tecnologias que se quer comparar, e são atribuídos níveis de 1 a 5 a cada tecnologia com base no quanto corresponde a cada requisito do cliente e característica.

Os requisitos do cliente, uma vez que este projeto se trata de uma solução genérica que deve funcionar para todos os projetos de *software*, são também requisitos genéricos. Dessa forma, a solução deverá gerar um projeto *front-end* utilizando uma *framework* que cumpra os seguintes requisitos:

- Seja acessível, de forma a que seja mais fácil para todos os elementos de uma equipa trabalharem na componente *front-end*.
- Seja poderosa, de forma a que seja possível criar um *front-end* que satisfaça todos os requisitos do negócio.
- Seja rápida, para que o produto desenvolvido tenha um bom desempenho e ofereça uma boa experiência de utilização.

Assim, devem ser consideradas as seguintes características das *frameworks*, para que se consiga dar resposta às necessidades do cliente:

- A complexidade da sintaxe, no sentido em que quanto menos complexa for a sintaxe da *framework*, maiores serão os ganhos para as equipas.
- Tenha as funcionalidades necessárias de uma *framework front-end*, tais como manipulação do *Document Object Model* (DOM), gestão do *state*, *routing*, validação de formulários ou um cliente HTTP.
- Seja leve em termos de tamanho, para que possa ser mais facilmente incluída no processo de *containerization* e CI/CD.
- Seja popular, para assim haver melhores possibilidades de continuar a ser desenvolvida no futuro, e também para haver mais recursos disponíveis para essa tecnologia.

A figura 3.5 mostra a tabela HOQ do QFD das tecnologias de *front-end*. A legenda dos símbolos utilizados na figura 3.5 pode ser consultada na figura 3.6.

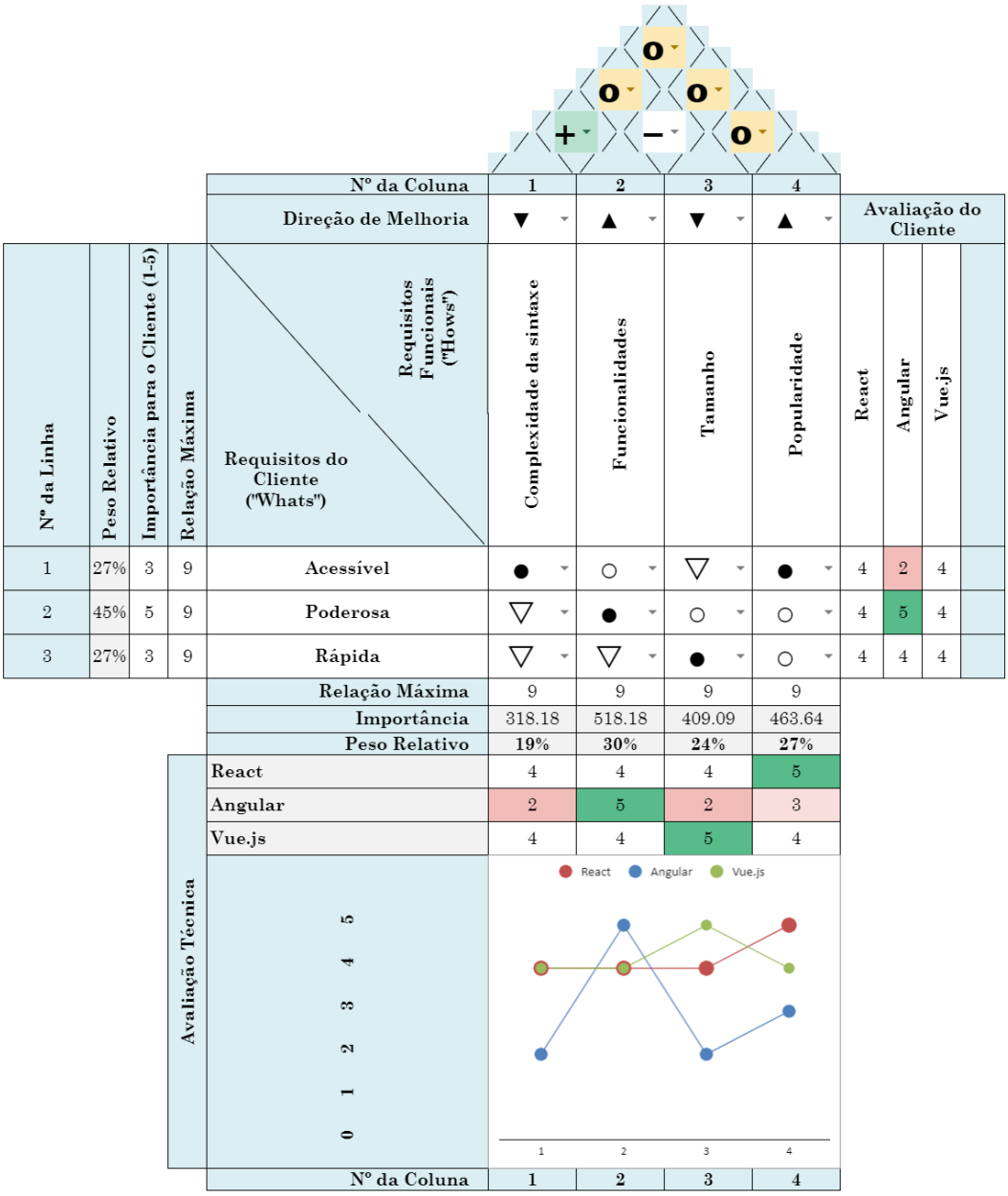

FIGURA 3.5: Tabela HOQ para as tecnologias *front-end*.

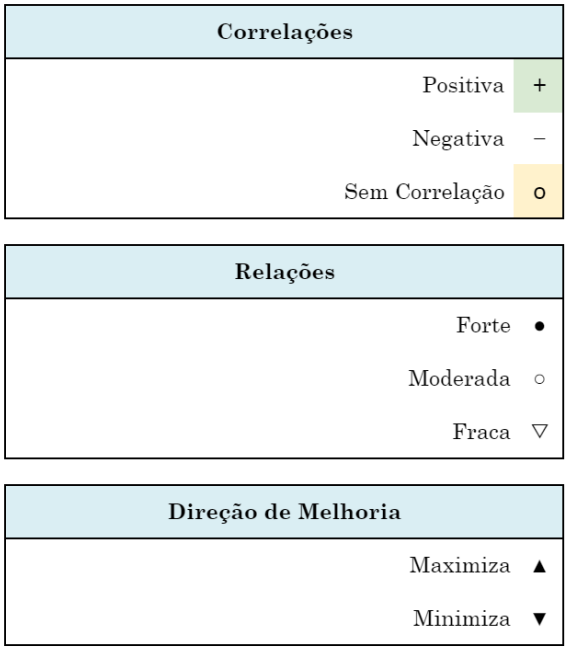

FIGURA 3.6: Legenda dos símbolos utilizados na tabela HOQ.

Relacionando as três *frameworks* com os requisitos do cliente:

- Em relação à acessibilidade, ambos o React e o Vue.js podem ser considerados igualmente acessíveis. Estas duas *frameworks* são de fácil instalação, tendo uma sintaxe simples baseada essencialmente em HTML e JavaScript. O Angular já é consideravelmente menos acessível devido à sua sintaxe mais complexa, e também pelo seu uso de TypeScript.
- Das três *frameworks*, o Angular é a *framework* mais poderosa, incluindo suporte nativo para mais funcionalidades. O React e o Vue.js também conseguem ser tão poderosos como o Angular, mas apenas com *plugins* ou bibliotecas externas.
- Por fim, em relação à rapidez, as três *frameworks* oferecem um desempenho semelhante.

Relacionando as três *frameworks* agora com as características:

- Em relação à sintaxe:
  - O React tem uma sintaxe baseada quase unicamente em JavaScript, através de uma "funcionalidade" do JavaScript denominada de JSX. O JSX é uma forma de se escrever HTML em JavaScript.

    O excerto 3.1 é um excerto simples de código com React, e na linha 2 consegue-se ver a atribuição de uma expressão semelhante a HTML a uma variável JavaScript. Essa expressão, apesar de parecida com HTML, está na verdade escrita em JSX, o que torna possível a inclusão de variáveis como o `{name}`.

    Depois, durante a fase de compilação, o React transforma o código JSX em código normal JavaScript, de forma a ser corretamente interpretado pelo navegador de internet.

```
1 const name = 'John';
2 const element = <h1>Hello, {name}</h1>;
3
4 ReactDOM.render(
5   element,
6   document.getElementById('root')
7 );
```

EXCERTO 3.1: Excerto de código com React.

– O Angular tem uma sintaxe mais particular e que tem uma curva de aprendizagem maior, como se pode verificar pelo excerto 3.2, fazendo uso de funcionalidades do TypeScript. Uma dessas funcionalidades são os *decorators*, tal como o `@Component`, que servem para ligar dados adicionais a classes normais. Com essa ligação, o programador só tem de se preocupar em criar os componentes, e o Angular trata automaticamente de manipular e gerir o DOM.

  O Angular, ao contrário do React, não inclui o *template* (HTML) e a lógica JavaScript no mesmo ficheiro, mas sim em ficheiros separados. No excerto 3.2, o *template* é incluído no componente na linha 5.

```
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-product-alerts',
5    templateUrl: './product-alerts.component.html',
6    styleUrls: ['./product-alerts.component.css']
7  })
8
9  export class ProductAlertsComponent implements OnInit {
10   constructor() {
11   }
12
13   ngOnInit() {
14   }
15
16 }
```

EXCERTO 3.2: Excerto de código com Angular.

– O Vue.js, tal como o React e o Angular, é também uma *framework* baseada em componentes. Na sintaxe dos seus componentes, o Vue.js utiliza JavaScript normal e HTML, sendo por isso uma *framework* acessível para novos programadores.

  Como se pode ver pelo excerto 3.3, o Vue.js é um misto entre React e Angular, no sentido em que usa só um ficheiro por cada componente, mas o *template* e o código JavaScript estão separados.

```
<template>
  <ul>
    <li v-for="user in users" :key="user.id" @click="selectUser(user.id)"
    >
      {{ user.name }}
    </li>
  </ul>
</template>

<script>
  export default {
    props: ['users'],
    methods: {
      selectUser: function(userId) {
        this.$emit(
          'selectUser',
          this.users.find(u => u.id === userId)
        )
      },
    },
  }
</script>
```

EXCERTO 3.3: Excerto de código com Vue.js.

Apesar de esta ser uma categoria algo subjetiva, deu-se uma vantagem ao React e ao Vue.js pela sua sintaxe relativamente simples e mais próxima de HTML e JavaScript. O Angular tem uma sintaxe mais particular, tendo assim um grau de aprendizagem relativamente maior.

- Em relação às funcionalidades, o Angular é a *framework* que tem mais funcionalidades "*out of the box*", isto é, de forma nativa. A tabela 3.2 mostra uma comparação entre algumas das principais funcionalidades de uma *framework front-end*.

TABELA 3.2: Funcionalidades nativas das *frameworks*.

	React	**Angular**	**Vue.js**
Manipulação do DOM	X	X	X
Gestão do *state*	X	X	X
Routing		X	X
Validação de formulários		X	
Cliente HTTP		X	

No entanto, com *plugins* e bibliotecas externas, as três *frameworks* acabam por conseguir ter as mesmas funcionalidades. Por exemplo, apesar de o React não ter capacidades nativas de *routing*, esta funcionalidade pode ser facilmente obtida com a biblioteca "react-router".

Desta forma, as três *frameworks* são equivalentes neste aspeto. Todavia, deu-se uma ligeira vantagem ao Angular por ser uma *framework* mais completa e de não necessitar de recursos externos.

- Em relação ao tamanho, no momento da instalação o Angular é a *framework* mais pesada, devido à quantidade de funcionalidades que oferece, seguida do React e depois do Vue.js, que é a *framework* mais leve (Daityari 2020).

  Comparando uma aplicação simples "Hello World", verifica-se o mesmo comportamento, sendo que a aplicação em Vue.js tende a ser a mais leve, seguida da aplicação React, e finalmente pela aplicação Angular (Schwarzmüller 2020).

  Contudo, comparando aplicações reais de dimensões maiores, as três *frameworks* acabam por produzir aplicações de tamanhos comparáveis.

- Por fim, em relação à popularidade, é difícil obter uma métrica fiável de popularidade, pois diferentes fontes fornecem diferentes resultados.

  Por exemplo, considerando o número de "*stars*" nos respetivos repositórios do GitHub, o Vue.js é a *framework* mais popular, seguido pelo React e depois pelo Angular. Por outro lado, considerando o número de *downloads* semanais no NPM de cada *framework*, o React está à frente com uma boa margem, seguido pelo Vue.js e finalmente pelo Angular.

  Sendo esta última métrica mais relevante para medir a popularidade das *frameworks*, uma vez que representa a utilização prática que cada *framework* está a ter, pode-se concluir que a *framework* mais popular é o React, seguido pelo Vue.js e depois pelo Angular.

Finalizando o processo de QFD, e considerando tanto os requisitos do cliente como as características de cada *framework*, chega-se à seguinte ordenação final, começando pela *framework* mais adequada para os clientes:

1. React.
2. Vue.js.
3. Angular.

3.3.2 *Back-end*

Para a análise das tecnologias da componente de *back-end*, as alternativas consideradas foram o Java, o Node.js e o Python, e o método utilizado foi o AHP.

Analytic Hierarchy Process

O modelo AHP é um método estruturado cuja finalidade é o apoio à tomada de decisão. Segundo Saaty (1994), o AHP cumpre essa finalidade ao partir o problema em partes mais pequenas, e depois agregar soluções para todos esses subproblemas. No AHP, o problema é estruturado como uma hierarquia, à qual é depois aplicado um processo de priorização.

O modelo AHP é um processo composto por várias fases:

- **Fase 1: Construção da árvore hierárquica de decisão**

  Esta fase envolve a definição do problema a resolver numa hierarquia composta por um objetivo, os critérios que serão considerados, e as alternativas que vão ser comparadas.

  Para este cenário em específico, a decisão a ser tomada é a priorização das tecnologias *back-end* mais úteis para o cliente. Os critérios do cliente para estas tecnologias são a usabilidade, o desempenho e as funcionalidades.

  Esta hierarquia encontra-se esquematizada na figura 3.7.

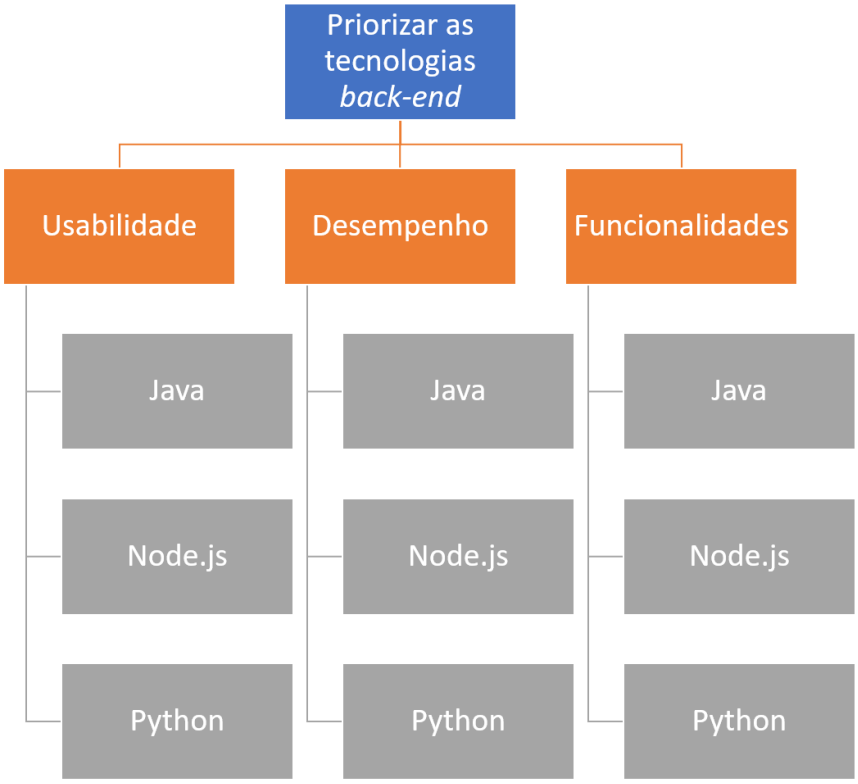

FIGURA 3.7: A árvore hierárquica de decisão com o objetivo, critérios e alternativas consideradas.

O critério da usabilidade envolve todos os aspetos da utilização das tecnologias, como a sua sintaxe e a legibilidade do código. O critério do desempenho diz respeito ao desempenho das tecnologias a executar certos tipos de tarefas. Por fim, o critério das funcionalidades abrange as funcionalidades e mecanismos que as tecnologias oferecem aos programadores para desenvolverem as suas aplicações.

- **Fase 2: Comparação das alternativas e critérios**

  Esta fase envolve a elaboração da matriz de comparação. O propósito desta matriz é estabelecer prioridades entre os elementos para cada nível da hierarquia. Para isso, os critérios são comparados entre si e para cada comparação é atribuído um valor da escala de fatores, que representa o nível de importância desse critério quando comparado com outro. No artigo "*How to Make a Decision: The Analytic Hierarchy Process*", Saaty (1994) propôs a uma escala fundamental de valores, descrita na tabela 3.3.

TABELA 3.3: Escala fundamental de valores.

Nível de Importância	Definição	Explicação
1	Importância igual	Ambos os critérios contribuem igualmente para o objetivo
3	Importância moderada	Um critério é levemente favorecido em relação a outro
5	Importância forte	Um critério é fortemente favorecido em relação a outro
7	Importância muito forte	Um critério é muito fortemente favorecido em relação a outro
9	Importância extrema	Um critério é favorecido em relação a outro com o mais alto grau de certeza
2, 4, 6, 8	Níveis intermédios	Uma condição de compromisso entre duas definições

A tabela 3.4 representa a matriz de comparação dos critérios definidos, seguindo os valores descritos na escala fundamental de Thomas Saaty. O critério das funcionalidades é favorecido em relação ao desempenho com um nível de importância ligeiramente abaixo de forte, e é favorecido em relação à usabilidade com um nível de importância ligeiramente abaixo do moderado. O critério da usabilidade é favorecido em relação ao desempenho com um nível abaixo do moderado.

TABELA 3.4: Matriz de comparação.

	Usabilidade	Desempenho	Funcionalidades
Usabilidade	1	2	1/2
Desempenho	1/2	1	1/4
Funcionalidades	2	4	1

- **Fase 3: Prioridade relativa de cada critério**

  Nesta fase é calculada a prioridade relativa de cada critério, obtendo assim o peso de cada um dos critérios. Para isso, em primeiro lugar, são calculadas as somas de cada coluna da tabela, originando assim a tabela 3.5.

TABELA 3.5: Matriz de comparação com a soma das colunas.

	Usabilidade	Desempenho	Funcionalidades
Usabilidade	1	2	1/2
Desempenho	1/2	1	1/4
Funcionalidades	2	4	1
Soma	**7/2**	**7**	**7/4**

De seguida, é necessário normalizar os valores da matriz, ou seja, igualar todos os critérios a uma mesma unidade. Para efetuar a normalização, cada valor da matriz é dividido pela soma da sua respetiva coluna, como representado pela tabela 3.6.

TABELA 3.6: Matriz de comparação normalizada.

	Usabilidade	**Desempenho**	**Funcionalidades**
Usabilidade	2/7	2/7	2/7
Desempenho	1/7	1/7	1/7
Funcionalidades	4/7	4/7	4/7

O passo seguinte é o cálculo da prioridade relativa de cada critério. Para isso é calculada a média aritmética de cada linha, cujos valores estão representados na tabela 3.7.

TABELA 3.7: Matriz com a prioridade relativa.

	Prioridade Relativa
Usabilidade	0,29
Desempenho	0,14
Funcionalidades	0,57

Os resultados desta fase permitem a ordenação dos critérios, com base na sua importância para o cliente. Para este caso específico, conclui-se que o critério das funcionalidades é o mais importante, seguido pelo critério da usabilidade e, finalmente, pelo critério do desempenho.

- **Fase 4: Avaliar a consistência das prioridades relativas**

  O objetivo desta fase é a validação dos resultados obtidos na fase anterior, para garantir que estes são valores consistentes. Esta consistência baseia-se no pressuposto de que o método AHP é racional. Quer isto dizer que se A é preferível a B e B é preferível a C, então A terá de ser preferível a C.

  Para realizar esta validação, o método AHP possui os conceitos de Índice de Consistência (IC) e Razão de Consistência (RC). Se a RC for superior a 0,1, isso significa que os resultados obtidos não são confiáveis e consistentes, devendo por isso ser revistos.

  Para calcular a RC, é necessário primeiro calcular o valor de λ_{max}, obtido a partir da seguinte fórmula:

$$Ax = \lambda_{max}x \tag{3.1}$$

  Na fórmula 3.1, A é a matriz de comparação (tabela 3.4), x é o vetor das prioridades relativas e λ_{max} é o valor próprio da matriz. Substituindo as variáveis conhecidas da fórmula 3.1, obtém-se a seguinte equação:

$$\begin{bmatrix} 1 & 2 & 0,50 \\ 0,50 & 1 & 0,25 \\ 2 & 4 & 1 \end{bmatrix} \begin{bmatrix} 0,29 \\ 0,14 \\ 0,57 \end{bmatrix} = \lambda_{max} \begin{bmatrix} 0,29 \\ 0,14 \\ 0,57 \end{bmatrix} \tag{3.2}$$

Resolvendo a primeira multiplicação de matrizes:

$$\begin{bmatrix} 0,855 \\ 0,4275 \\ 1,71 \end{bmatrix} = \lambda_{max} \begin{bmatrix} 0,29 \\ 0,14 \\ 0,57 \end{bmatrix} \tag{3.3}$$

Finalmente, pode-se obter o valor de λ_{max} da seguinte forma:

$$\lambda_{max} = \frac{(0,855/0,29) + (0,4275/0,14) + (1,71/0,57)}{3} = 3 \tag{3.4}$$

Tendo o valor de λ_{max}, é agora possível o cálculo do IC, através da seguinte fórmula, em que o n é o número de critérios considerados:

$$IC = \frac{\lambda_{max} - n}{n - 1} \tag{3.5}$$

Substituindo as variáveis conhecidas, obtém-se a seguinte equação:

$$IC = \frac{3 - 3}{3 - 1} = 0 \tag{3.6}$$

Sabendo o valor de IC, consegue-se agora calcular o RC através da seguinte fórmula:

$$RC = \frac{IC}{IR} \tag{3.7}$$

Na fórmula 3.7, o IR representa um índice aleatório para matrizes quadradas de n critérios, calculado pelo Laboratório Nacional de Oak Ridge, nos Estados Unidos da América. Como se pode verificar pela tabela 3.8, uma matriz de três critérios tem um IR de 0,58.

TABELA 3.8: Valores de IR para matrizes quadradas de ordem n.

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0,00	0,00	0,58	0,90	1,12	1,24	1,32	1,41	1,45	1,49

Assim, substituindo os valores da fórmula 3.7, chega-se ao seguinte valor de RC:

$$RC = \frac{0}{0,58} = 0 \tag{3.8}$$

Como a RC é inferior a 0,1, pode-se concluir que os valores das prioridades relativas são consistentes.

- **Fase 5: Construção da matriz de comparação paritária para cada critério, considerando cada uma das alternativas selecionadas**

  Nesta fase são repetidos os procedimentos da elaboração da matriz de comparação e da determinação da prioridade relativa, mas desta vez para as alternativas consideradas, e considerando cada um dos critérios.

  Uma vez que os procedimentos para a elaboração das tabelas são os mesmos, apenas vão ser incluídas as várias tabelas utilizadas para chegar à prioridade relativa, e não os cálculos intermédios para a determinação dos valores.

  - **Critério: Usabilidade**

    O excerto de código 3.4 é um exemplo escrito em Java, onde é inicializado um vetor, sendo que depois esse vetor é iterado e cada um dos seus valores é impresso na consola.

```java
public class Test {
    public static void main(String args[]) {
        String array[] = {"Hello", "World", "20"};
        for (String i : array) {
          System.out.println(i);
        }
    }
}
```

EXCERTO 3.4: Excerto de código em Java.

O excerto de código 3.5 é um exemplo semelhante ao anterior, mas em JavaScript.

```javascript
const array = ["Hello", "World", 20];
for (const i of array) {
    console.log(i);
}
```

EXCERTO 3.5: Excerto de código em JavaScript.

Por fim, o excerto 3.6 é também um exemplo semelhante, mas desta vez em Python.

```python
array = ["Hello", "World", 20]
for i in array:
    print(i)
```

EXCERTO 3.6: Excerto de código em Python.

Analisando os excertos anteriores, podem-se observar algumas diferenças. Primeiro, os excertos de JavaScript e Python podem ser executados da forma que estão, sendo que o excerto 3.5 pode ser executado em qualquer navegador de internet, e o excerto 3.6 pode ser executado como um script Python. No entanto, o Java *Virtual Machine* (JVM) necessita que o excerto Java tenha um ponto de entrada no código e um método denominado `main`, de forma a conseguir executar o código. Isso faz com que os ficheiros Java obriguem a ter mais código *boilerplate*, tornando assim a sintaxe mais pesada em termos de legibilidade.

Outro aspeto que se pode verificar é que tanto o JavaScript como o Python permitem a inicialização de vetores com valores de vários tipos. Em ambos os excertos, o vetor é inicializado com duas *strings* e um número inteiro. Com o Java isso já não

é possível, uma vez que é obrigatório declarar explicitamente o tipo do vetor, que neste caso é um vetor de *strings*. A razão para isso é porque o Java é uma linguagem tipada estaticamente[4], enquanto que o JavaScript e o Python são linguagens tipadas dinamicamente[5]. A diferença entre estes dois conceitos é que as linguagens tipadas estaticamente fazem a validação do tipo das variáveis no tempo de compilação, enquanto que as linguagens tipadas dinamicamente fazem essa validação no tempo de execução[6]. Isso torna as linguagens tipadas estaticamente mais seguras, uma vez que os possíveis erros são detetados mais cedo. O Node.js neste aspeto tem uma vantagem em relação ao Java e ao Python, pois oferece ambas as funcionalidades, uma vez que suporta tanto JavaScript como TypeScript. O TypeScript, de uma forma muito resumida, pode ser considerada como uma versão do JavaScript que é tipada estaticamente.

Em termos da utilização destas linguagens em implementações reais, estas também têm as duas diferenças. Por exemplo, a implementação de uma API REST, algo que é muito utilizado para as componentes *back-end* dos serviços desenvolvidos pela Critical TechWorks, é consideravelmente mais fácil em Node.js e Python do que em Java. *Frameworks* como o Express.js, discutida na secção 2.3.5, e o Flask, discutida na secção 2.3.7, permitem implementar facilmente uma API REST utilizando apenas algumas linhas de código, como demonstrado nos excertos de código 2.7 e 2.11. Em Java também é possível o desenvolvimento de APIs REST, utilizando ferramentas como o Spring Boot, por exemplo, que foi descrita na secção 2.3.2. No entanto, a implementação de uma API REST em Java é um pouco mais complexa, sendo necessário mais código do que em implementações semelhantes em Node.js ou Python.

Assim, com base no racional explicado nos parágrafos anteriores, chegou-se à matriz de comparação representada pela tabela 3.9.

TABELA 3.9: Matriz de comparação do critério "Usabilidade".

	Java	**Node.js**	**Python**
Java	1	1/5	1/5
Node.js	5	1	1
Python	5	1	1

[4] *static typing*
[5] *dynamic typing*
[6] *runtime*

A tabela 3.10 representa a tabela de comparação com as somas de cada coluna.

Tabela 3.10: Matriz de comparação do critério "Usabilidade" com a soma das colunas.

	Java	**Node.js**	**Python**
Java	1	1/5	1/5
Node.js	5	1	1
Python	5	1	1
Soma	**11**	**11/5**	**11/5**

A tabela 3.11 representa a tabela de comparação normalizada.

Tabela 3.11: Matriz de comparação normalizada do critério "Usabilidade".

	Java	**Node.js**	**Python**
Java	1/11	1/11	1/11
Node.js	5/11	5/11	5/11
Python	5/11	5/11	5/11

Por fim, a tabela 3.12 representa as prioridades relativas das alternativas.

Tabela 3.12: Matriz com a prioridade relativa do critério "Usabilidade".

	Prioridade Relativa
Java	0,09
Node.js	0,455
Python	0,455

– **Critério: Desempenho**

Em termos de desempenho, o Java é uma linguagem bastante eficiente, particularmente devido ao seu compilador *Just-In-Time* (JIT) e à sua arquitetura *multi-threaded*. Enquanto que o código Java é diretamente compilado, o código Python tem primeiro de ser interpretado (para se determinar os tipos de variáveis, por exemplo), o que causa alguma lentidão em tempo de execução. Por essa razão, o Java oferece um melhor desempenho do que o Python (Brihadiswaran 2020).

Tal como mencionado previamente neste documento, o Node.js utiliza um modelo de I/O não bloqueador, o que significa que o Node.js nunca irá ser bloqueado enquanto realiza operações de I/O. Isso faz com que os recursos de processamento e memória do servidor possam ser utilizados de forma eficiente. É essa característica que faz com que o Node.js consiga assegurar milhares de operações I/O apenas com uma única *thread*. Quando comparado com o Python, o Node.js oferece um desempenho superior em vários cenários de desenvolvimento *web* (Lei, Ma e Tan 2014).

Entre o Java e o Node.js, de forma geral, o Java oferece um desempenho superior. Embora ambas as linguagens sejam eficientes, as suas arquiteturas são bastante distintas. O Java tem uma arquitetura bloqueadora *multi-threaded*, enquanto que o Node.js tem uma arquitetura não bloqueadora *single-threaded*. Apesar de o Node.js ser muito eficiente a processar operações I/O, o Java acaba por conseguir um melhor desempenho devido à sua arquitetura *multi-threaded*. E, em poder computacional, o Java também tem a vantagem (Okuneu 2020). No entanto, para aplicações ou sistemas com grandes quantidades de operações I/O, é possível colmatar as limitações da arquitetura *single-threaded* do Node.js ao se utilizar um *cluster* Node.js, conseguindo dessa forma que vários processos possam ser executados simultaneamente.

Os serviços desenvolvidos pela Critical TechWorks, especialmente na área de *Infotainment & Interactivity Services*, habitualmente não são serviços com grande poder computacional, até porque são serviços que têm de ser executados nos veículos. Normalmente são serviços disponibilizados por APIs REST, que dependem mais de operações I/O. Dessa forma, tanto o Node.js como o Java oferecem um bom desempenho para este tipo de aplicações. No entanto, deu-se uma vantagem ao Java por ter um desempenho superior "*out of the box*".

A tabela 3.13 representa a matriz de comparação para o critério do desempenho, com base no racional apresentado.

TABELA 3.13: Matriz de comparação do critério "Desempenho".

	Java	**Node.js**	**Python**
Java	1	2	4
Node.js	1/2	1	3
Python	1/4	1/3	1

A tabela 3.14 representa a tabela de comparação com as somas de cada coluna.

TABELA 3.14: Matriz de comparação do critério "Desempenho" com a soma das colunas.

	Java	**Node.js**	**Python**
Java	1	2	4
Node.js	1/2	1	3
Python	1/4	1/3	1
Soma	**7/4**	**10/3**	**8**

A tabela 3.15 representa a tabela de comparação normalizada.

TABELA 3.15: Matriz de comparação normalizada do critério "Desempenho".

	Java	**Node.js**	**Python**
Java	4/7	3/5	1/2
Node.js	2/7	3/10	3/8
Python	1/7	1/10	1/8

A tabela 3.16 representa as prioridades relativas das alternativas.

TABELA 3.16: Matriz com a prioridade relativa do critério "Desempenho".

	Prioridade Relativa
Java	0,56
Node.js	0,32
Python	0,12

– **Critério: Funcionalidades**

Em termos de funcionalidades, as três linguagens podem ser consideradas relativamente equivalentes. Especialmente com os *plugins* e *frameworks* existentes hoje em dia, é possível fazer essencialmente tudo com qualquer uma das linguagens. No entanto, de forma geral, as três linguagens têm propósitos diferentes.

O Python é considerado como uma linguagem *general-purpose*, sendo utilizado para diferentes objetivos como *scripting*, automação, testes de *software* e desenvolvimento *web*. O Java, apesar de também ter certas aplicações na área do *front-end*, tem um foco claro no desenvolvimento *back-end*. O JavaScript nasceu exclusivamente como uma linguagem para ser utilizada no lado do cliente, contudo, graças ao Node.js, hoje em dia é também uma linguagem de *back-end* perfeitamente capaz.

Uma funcionalidade relevante a estas três linguagens é a gestão de pacotes. O Node.js tem o NPM, que é um dos gestores de pacotes mais populares no universo do desenvolvimento de *software*. O NPM não só permite a instalação de novos módulos, como também permite efetuar várias tarefas nos projetos, como fazer *build* e executar as aplicações ou servidores. Do lado do Python existe o pip, que é um gestor de pacotes para instalar pacotes Python. No entanto, não possui o mesmo nível de funcionalidades do NPM. Finalmente, no Java existem várias ferramentas diferentes, sendo o Maven a mais completa. O Maven permite gestão de dependências através de repositórios remotos, como o *central-repo*. Para além disso, é possível utilizar o Maven para efetuar uma variedade de tarefas diferentes, como lançar testes, fazer *build* do projeto, ou executar a aplicação ou servidor.

Para além disso, na análise dos critérios anteriores já foram enumeradas algumas diferenças de funcionalidades de cada uma das linguagens. Essas diferenças de funcionalidades incluem diferenças ao nível da compilação, de serem linguagens tipadas estaticamente ou dinamicamente, e ao nível da implementação ou desenvolvimento.

Concluindo, todas estas linguagens têm funcionalidades suficientes para serem uma boa solução como tecnologia de *back-end*. Todavia, uma vez que o Node.js e o Java foram tecnologias desenvolvidas especialmente para o *back-end*, estas acabam por ter algumas vantagens em relação ao Python.

A tabela 3.17 representa a matriz de comparação para o critério das funcionalidades.

TABELA 3.17: Matriz de comparação do critério "Funcionalidades".

	Java	**Node.js**	**Python**
Java	1	1	2
Node.js	1	1	2
Python	1/2	1/2	1

A tabela 3.18 representa a tabela de comparação com as somas de cada coluna.

TABELA 3.18: Matriz de comparação do critério "Funcionalidades" com a soma das colunas.

	Java	**Node.js**	**Python**
Java	1	1	2
Node.js	1	1	2
Python	1/2	1/2	1
Soma	**5/2**	**5/2**	**5**

A tabela 3.19 representa a tabela de comparação normalizada.

TABELA 3.19: Matriz de comparação normalizada do critério "Funcionalidades".

	Java	**Node.js**	**Python**
Java	2/5	2/5	2/5
Node.js	2/5	2/5	2/5
Python	1/5	1/5	1/5

A tabela 3.20 representa as prioridades relativas das alternativas.

TABELA 3.20: Matriz com a prioridade relativa do critério "Funcionalidades".

	Prioridade Relativa
Java	0,40
Node.js	0,40
Python	0,20

Efetuada a análise anterior, consegue-se agora construir a matriz de comparação geral de todos os critérios e todas as alternativas. A figura 3.8 representa a matriz de comparação paritária para cada critério considerando cada uma das alternativas selecionadas, sendo assim possível facilmente perceber o peso relativo de cada componente da análise realizada.

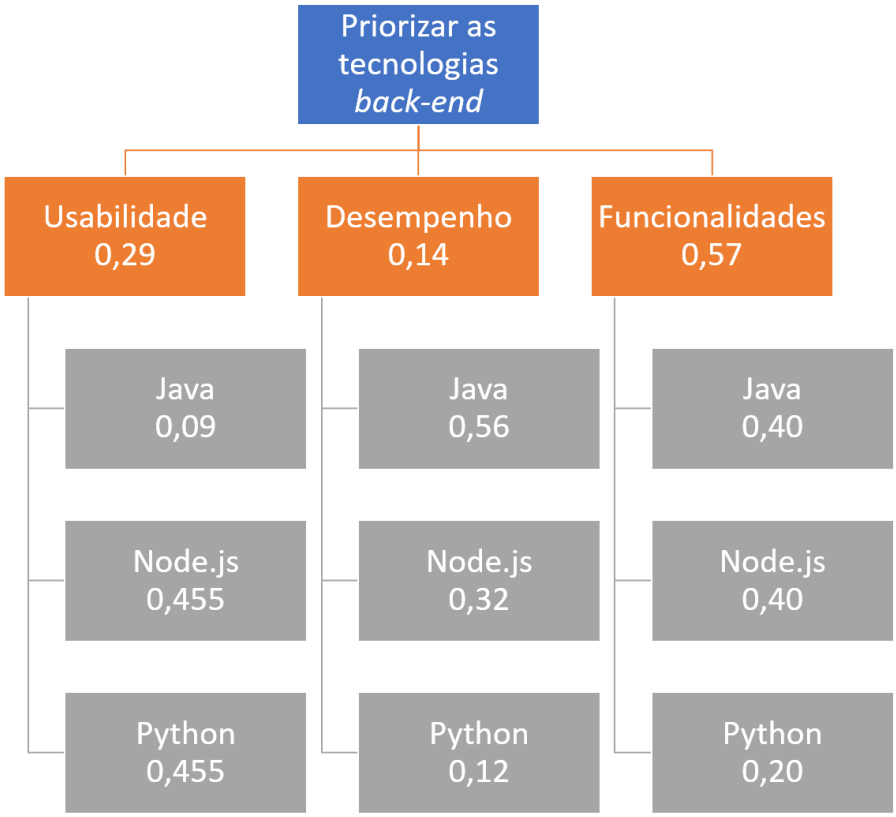

FIGURA 3.8: Matriz de comparação paritária das tecnologias *back-end*.

- **Fase 6: Obter a prioridade composta para as alternativas**

  Nesta fase é calculada a prioridade composta das alternativas, que vai ser o valor que vai tornar possível a ordenação das tecnologias *back-end*. Para calcular este valor, multiplica-se a matriz das prioridades das alternativas pela matriz do peso relativo dos critérios.

$$\begin{bmatrix} 0,09 & 0,56 & 0,40 \\ 0,455 & 0,32 & 0,40 \\ 0,455 & 0,12 & 0,20 \end{bmatrix} \begin{bmatrix} 0,29 \\ 0,14 \\ 0,57 \end{bmatrix} = \begin{bmatrix} 0,33 \\ 0,41 \\ 0,26 \end{bmatrix} \tag{3.9}$$

- **Fase 7: Escolha da alternativa**

  Esta é a fase mais simples de todo este processo, sendo que apenas requere uma análise da matriz calculada na fase anterior. Assim, com base nessa matriz, obtém-se a seguinte ordem para as tecnologias *back-end*, começando pela mais adequada para os clientes:

  1. Node.js, com um peso relativo de 41%.
  2. Java, com um peso relativo de 33%.

3. Python, com um peso relativo de 26%.

3.3.3 Restantes Componentes

Nesta secção é efetuada a seleção das tecnologias para as restantes componentes da solução. Estas componentes não são tão críticas como as componentes de *front-end* e *back-end*, pois apesar de serem também importantes, podem mais facilmente ser substituídas ou modificadas numa fase mais avançada do projeto.

Devido a isso, estas componentes não vão ser sujeitas a métodos tão formais de seleção e apoio à decisão. É por isso que estão todas nesta mesma secção, ao contrário do *front-end* e *back-end* que tiveram uma secção individual mais completa.

Assim, as escolhas para estas componentes foram realizadas maioritariamente devido à utilidade que terão para as equipas às quais esta solução se destina, tirando o máximo de proveito da *stack* tecnológica já existente na empresa.

Containerization

Em termos de *containerization*, o Docker é atualmente considerada a ferramenta padrão para aplicações comerciais. Para além disso, todas as atuais equipas da área de *Infotainment & Interactivity Services* da Critical TechWorks estão a utilizar o Docker nos seus projetos, havendo assim já conhecimento dessa ferramenta entre as equipas.

Dessa forma, a ferramenta de *containerization* selecionada para esta solução foi o Docker.

Orquestração de *Containers*

Tal como o Docker é a ferramenta padrão na área da *containerization*, também o Kubernetes é a ferramenta padrão na área de orquestração de *containers*. E tal como com o Docker, o Kubernetes é também a ferramenta utilizada na Critical TechWorks para os outros projetos, sendo por isso essa a ferramenta que mais se encontra alinhada com a *stack* tecnológica da empresa.

Assim, por essas razões, a ferramenta selecionada de orquestração de *containers* foi o Kubernetes.

Fornecedores *Cloud*

Considerando os fornecedores *cloud*, o Azure e a AWS podem ser considerados como equivalentes, enquanto que o Google Cloud está num nível ligeiramente diferente, especializando-se para tipos de projetos mais específicos. Assim, tanto o Azure como a AWS dariam excelentes escolhas para esta solução.

Desta forma, torna-se novamente relevante considerar a *stack* tecnológica da empresa, para garantir que esta solução se consegue lá inserir com o mínimo de trabalho possível. E, neste momento, todos os projetos da área de *Infotainment & Interactivity Services* encontram-se com uma infraestrutura alojada na AWS. Para além disso, não só cada projeto individual tem uma infraestrutura na AWS, como também existe uma infraestrutura comum a todos os projetos. Quer isto dizer que, quando for iniciado um novo projeto nesta área, idealmente esse projeto deveria também partilhar a infraestrutura já existente.

Assim, o fornecedor *cloud* selecionado para esta solução foi a AWS.

Gestão da Infraestrutura

Para o âmbito da gestão da infraestrutura, qualquer uma das ferramentas mencionadas na secção 2.4.6 serviriam para o propósito desta solução. Assim sendo, é novamente importante verificar qual a ferramenta que iria trazer mais valor às equipas.

Tal como mencionado na secção anterior, todas as equipas da área de *Infotainment & Interactivity Services* partilham uma infraestrutura comum, na qual os diversos serviços de cada equipa estão alojados. Toda essa infraestrutura comum, assim como toda a infraestrutura específica de cada serviço, é construída utilizando o Terraform. Ou seja, já existem diversos *scripts* Terraform para construir infraestrutura que podem ser aproveitados por todas as novas equipas irão surgir no futuro. Assim sendo, faz todo o sentido esta solução manter as mesmas tecnologias que já são utilizadas atualmente.

Devido a isso, a tecnologia de gestão da infraestrutura selecionada foi o Terraform.

CI/CD

Em relação às tecnologias de CI/CD, mais uma vez, qualquer ferramenta mencionada serviria bem os objetivos desta solução. As equipas da Critical TechWorks da área de *Infotainment & Interactivity Services* estão todas a utilizar o Jenkins como ferramenta de CI/CD. Ou seja, já existem *pipelines* e funções Groovy para o Jenkins, e que são utilizadas por todas as equipas.

Assim, de forma a esta solução ser mais facilmente integrada no fluxo de trabalho das equipas, a ferramenta de CI/CD selecionada foi o Jenkins.

3.4 Sumário

Neste capítulo foram realizadas várias análises de pontos-chave da solução, nomeadamente a análise de requisitos, a análise de valor e a análise de alternativas.

Na análise de requisitos foram enumerados e descritos os requisitos funcionais e não funcionais da solução. Estes são os requisitos que a solução final deve cumprir, sendo por isso um dado fundamental para as fases de *design* e implementação.

Na análise de valor foi feita uma análise mais detalhada do problema a ser resolvido, numa perspetiva de inovação. De seguida, foi descrita a proposta de valor da solução, onde foi explicado o valor que este projeto irá proporcionar aos seus utilizadores.

Por fim, foi realizada a análise de alternativas, onde as tecnologias identificadas no capítulo anterior foram comparadas. Concluído o processo de análise de alternativas, chegou-se à seguinte *stack* tecnológica para esta solução:

- ***Front-end*:** React
- ***Back-end*:** Node.js
- ***Containerization*:** Docker
- **Orquestração de *containers*:** Kubernetes
- **Fornecedor *cloud*:** AWS
- **Gestão da infraestrutura:** Terraform
- **CI/CD:** Jenkins

Capítulo 4

Design

Este capítulo explora alguns aspetos sobre o *design* da solução. Sendo este projeto, por natureza, um projeto genérico de desenvolvimento de *software*, e tendo como objetivo uma solução que consiga satisfazer as necessidades do maior número possível de equipas, existe aqui um balanço importante a ser mantido. Por um lado, é importante a utilização de um *design* ponderado e com um racional baseado em boas práticas da engenharia informática. Por outro, não pode ter um *design* muito aprofundado ou especializado, uma vez que tem de ser uma solução genérica que seja útil para todas as equipas. Desta forma, foram adotadas boas práticas numa perspetiva mais de alto nível, tendo como principal preocupação a geração de um projeto com bases sólidas em termos de engenharia informática, e que ofereça alguma flexibilidade às equipas de desenvolvimento, para dessa forma lhes proporcionar a melhor experiência possível.

O capítulo está dividido em cinco secções diferentes. A primeira secção aborda as principais componentes deste projeto, oferecendo assim uma visão geral sobre o mesmo. A segunda secção explora as alternativas consideradas para a geração dos repositórios do projeto. A terceira secção tem como objetivo uma explicação mais detalhada de todo o fluxo da solução. A quarta secção detalha a arquitetura das componentes mais importantes da solução, sendo essas a componente *front-end* e *back-end*, e a infraestrutura alojada na *cloud*. Finalmente, a última secção aborda alguns aspetos mais específicos sobre a API REST gerada para a componente *back-end*, e sobre princípios REST no geral.

4.1 Componentes Principais

O objetivo da solução, como previamente mencionado, é automatizar o processo de iniciar um novo projeto de desenvolvimento. Essa automação será assegurada por diversos *scripts* que irão conter toda a lógica da solução.

Existem dois tipos de *scripts* envolvidos no fluxo da solução: um *script* principal e os *scripts* secundários. O *script* principal é o único *script* com o qual o utilizador vai interagir. Os *scripts* secundários são os *scripts* responsáveis por gerar cada uma das componentes do projeto, e apenas são executados pelo *script* principal dependendo das escolhas do utilizador.

Dessa forma, tal como mostra a figura 4.1, a solução desenvolvida pode ser dividida em quatro componentes principais:

- ***Script* principal**: Este vai ser o *script* responsável por gerir todo o processo e, tal como mencionado, vai ser o único *script* executado pelo utilizador. Este *script* vai ser responsável por invocar todos os outros *scripts*.
- ***Scripts* secundários**:

- ***Script* do *back-end***: Este vai ser o *script* responsável por gerar o repositório da componente *back-end*. Este repositório contém o código de uma API REST que irá servir como o servidor do projeto *full-stack*.
- ***Script* do *front-end***: Este vai ser o *script* responsável por gerar o repositório da componente *front-end*. Este repositório contém o código de uma aplicação *web* que irá comunicar com a API REST, servindo assim como o cliente do projeto *full-stack*.
- ***Script* da infraestrutura**: Este vai ser o *script* responsável por gerar o repositório da componente de infraestrutura. Este repositório contém o código Terraform para provisionar toda a infraestrutura necessária para alojar todo este projeto. A arquitetura desta infraestrutura é detalhada na secção 4.4.

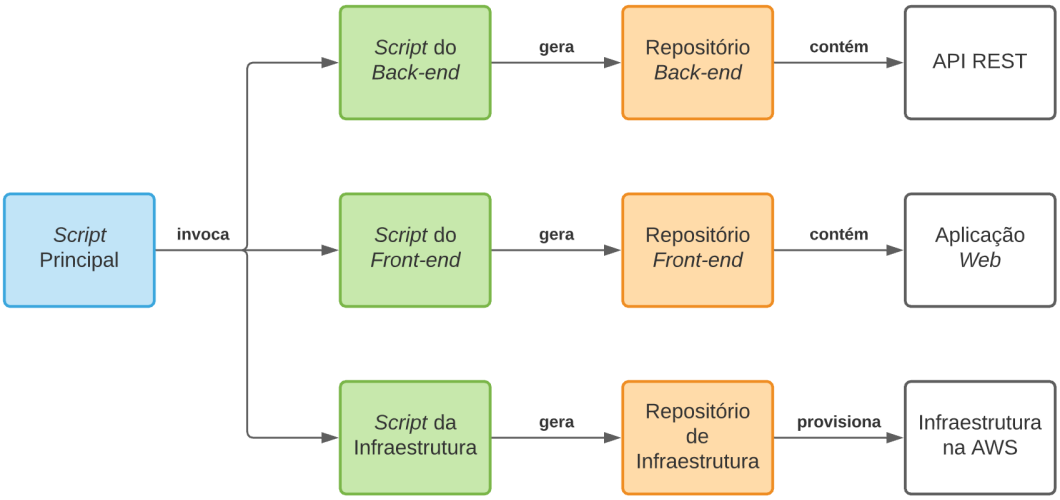

FIGURA 4.1: Componentes principais da solução.

4.2 Alternativas de Geração dos Repositórios

O propósito desta secção é mencionar as possíveis alternativas ou opções tecnológicas que foram consideradas para os *scripts* responsáveis por gerar as componentes *front-end* e *back-end*, mencionados na secção anterior.

Foram identificadas três possíveis alternativas para a implementação desses *scripts*:

- Criação de uma solução customizada, desenvolvida de raiz.
- Utilização do Yeoman, com *generators* já existentes.
- Utilização do Yeoman, com *generators* personalizados.

4.2.1 Solução Customizada

Esta opção iria envolver a criação de raiz de uma solução customizada que fosse capaz de gerar os repositórios *front-end* e *back-end*. Uma possível forma de fazer isso poderia ser gerar manualmente os repositórios e editá-los até à sua forma final, já com todos os ficheiros e testes necessários. De seguida, o *script* poderia clonar estes repositórios para a máquina local do utilizador.

Em termos de esforço de implementação, o principal desafio desta alternativa seria o desenvolvimento inicial dos repositórios, visto que teria de ser feito manualmente. Tendo os repositórios

finalizados, seria apenas uma questão de clonar os repositórios para a máquina local do utilizador e instalar as dependências dos projetos. Visto ambas as componentes *front-end* e *back-end* utilizarem o NPM para gestão de dependências, todas essas dependências já iriam estar nos ficheiros `package.json`, pelo que apenas bastaria executar o comando "`npm install`" para instalar todas as dependências.

No entanto, esta alternativa tem a vantagem de os repositórios gerados serem facilmente alterados pelas equipas, caso queiram, pois para isso bastaria editarem manualmente os repositórios que iriam servir de exemplo.

Em termos de linguagens de *scripting*, para este caso qualquer uma iria servir para o efeito. Assim, poderia-se utilizar Python, por exemplo, devido às suas funcionalidades e acessibilidade.

4.2.2 Yeoman com *Generators* Existentes

Esta opção iria envolver a utilização da ferramenta Yeoman. Tal como mencionado na secção 2.3.1, esta ferramenta utiliza *generators*, que são ficheiros com *templates* de *scaffolding*.

No *website* oficial do Yeoman é possível pesquisar toda a lista de *generators* atualmente existentes. Dessa lista, existem alguns que poderiam ser escolhas interessantes para este projeto:

- O *generator* "express" cria um projeto utilizando Node.js com Express.js. Permite uma estrutura de ficheiros básica ou baseada na arquitetura MVC. Tem suporte para várias bases de dados, como MongoDB ou PostgreSQL.
- O *generator* "rest" gera uma API REST utilizando Node.js, Express.js e MongoDB. É customizável, pois permite o utilizador escolher o que pretende instalar. E, para além disso, possui alguns extras que podem ser benéficos para o desenvolvimento de APIs REST, como um um validador do corpo de pedidos HTTP, utilizando a ferramenta "bodymen", e mensagens de erro padronizadas, utilizando as ferramentas "bodymen" e "querymen".
- Também existem alguns *generators* para gerar a componente *front-end*, como por exemplo os *generators* "react-webpack" e "react-scaffolder".

A vantagem clara desta alternativa é a facilidade de implementação, uma vez que bastaria executar o *generator* com o Yeoman. No entanto, as probabilidades de encontrar um *generator* que consiga dar resposta a todas as necessidades deste projeto sem qualquer alteração são relativamente baixas. Outra desvantagem é que depois as equipas estariam "presas" aos repositórios gerados, uma vez que qualquer alteração que as equipas queiram fazer aos repositórios iria envolver a criação de um novo *generator*.

4.2.3 Yeoman com *Generators* Personalizados

Esta alternativa seria uma mistura entre as duas alternativas anteriores. No fundo, iria também basear-se na utilização de *generators* do Yeoman. No entanto, ao contrário da alternativa anterior, esta alternativa iria envolver a criação de um *generator* personalizado que consiga satisfazer todos os requisitos do projeto.

A maneira mais fácil de implementar uma solução com esta alternativa seria provavelmente encontrar *generators* atualmente disponíveis que gerassem repositórios o mais próximos possível do resultado final esperado, e depois modificar esses *generators* de forma a produzirem os resultados expectados.

Dessa forma, em termos de implementação, esta alternativa seria mais fácil do que a primeira alternativa considerada, uma vez que não iria envolver a criação de todo o projeto de raiz. Para além disso, dá na mesma a flexibilidade às equipas de alterarem o que desejarem, de forma a produzir resultados alinhados às suas necessidades.

4.3 Fluxo da Solução

Esta secção vai explorar em mais detalhe o fluxo da solução desenvolvida. Das alternativas exploradas na secção anterior, aquela que acabou por ser selecionada foi a terceira opção - a utilização do Yeoman, com *generators* personalizados.

Na figura 4.2 está ilustrado um fluxograma de todo o processo da solução. Segue-se uma explicação detalhada do fluxo:

1. Para iniciar o processo, o utilizador executa o *script* principal, que será responsável por gerir todo o processo. Esse é o *script* que vai executar todos os outros *scripts*, e vai assegurar o correto funcionamento da solução. O *script* vai oferecer ao utilizador algumas escolhas e algumas configurações diferentes de funcionamento, ficando este *script* responsável por gerir todo este fluxo.

2. Após executar o *script* inicial, este vai questionar ao utilizador se vai querer gerar o repositório *back-end*. A razão desta escolha é simplesmente para oferecer mais flexibilidade aos utilizadores, uma vez que estes podem já ter um *back-end* existente, e podem apenas estar a necessitar de gerar um *front-end* para o seu projeto. Assim, se a equipa não estiver a precisar de um *back-end*, não vai ser forçada a gerar um. Caso o utilizador responda afirmativamente à questão de querer gerar o repositório *back-end*, o *script* vai pedir uma localização para esse repositório (isto é, a pasta no seu computador onde o repositório deve ser gerado) e vai de seguida invocar o *script* secundário da componente *back-end*.

   Este *script* secundário é, na verdade, um *generator* Yeoman. Este *generator* vai começar por questionar ao utilizador o nome que este quer dar ao projeto *back-end*, procedendo depois à geração de um repositório com o nome definido. Este repositório vai ser gerado na máquina local do utilizador, sendo este o responsável de posteriormente inicializar o Git (ou qualquer outro sistema de controlo de versões) no repositório e de fazer o respetivo *push* para a sua plataforma de escolha (GitHub, Bitbucket, etc.). Este repositório vai ser gerado com base num repositório *template* que existe no projeto do *generator*. No entanto, é possível adaptar esta solução para gerar literalmente qualquer tipo de projeto, e em qualquer linguagem de programação, bastando para isso apenas alterar os ficheiros que são usados como *template* para a geração do repositório. Os detalhes de implementação sobre o funcionamento deste *generator* são discutidos na secção 5.1.1.

   Se o utilizador tiver escolhido gerar um repositório *back-end*, então no final da geração do mesmo o *script* vai avançar para o próximo passo. Caso o utilizador tenha respondido de forma negativa, então o *script* vai simplesmente avançar para o próximo passo do fluxo, não fazendo qualquer ação neste passo.

3. O próximo passo é semelhante ao anterior, mas desta vez para a componente *front-end*. Ou seja, o *script* vai questionar se o utilizar quer gerar um repositório *front-end* e, caso queira, vai pedir a localização para esse repositório e depois vai invocar o *script* secundário da componente *front-end*.

Este *script* secundário, tal como o *script* do *back-end*, é um *generator* Yeoman, tendo um funcionamento muito semelhante ao *generator* do *back-end*. O *generator* vai começar por questionar ao utilizador o nome que quer dar ao projeto *front-end*, gerando depois o respetivo repositório. Tal como aconteceu com o *back-end*, este *generator* vai também gerar o repositório com base em ficheiros *template*, sendo também possível fazer qualquer tipo de alteração que seja necessária. Os detalhes de implementação sobre o funcionamento deste *generator* são discutidos na secção 5.1.3.

Caso o utilizador tenha escolhido gerar um repositório *front-end*, então no final da geração do mesmo o *script* vai avançar para o próximo passo. Caso contrário, o *script* vai avançar para o próximo passo do fluxo, não fazendo qualquer ação neste passo.

4. O próximo passo é bastante semelhante com os dois anteriores, mas desta vez para o repositório da infraestrutura. O *script* principal vai questionar se o utilizador quer gerar um repositório de infraestrutura. Em caso afirmativo, vai pedir a localização para o mesmo, e vai invocar o *script* secundário da componente de infraestrutura.

   Este *script* secundário é também um *generator* Yeoman, tendo um funcionamento semelhante aos anterior. Isto é, vai gerar o repositório com base em ficheiros *template*. Os detalhes de implementação sobre o funcionamento deste *generator* são discutidos na secção 5.1.3.

5. De seguida, o *script* vai questionar se o utilizador quer executar as aplicações localmente na sua máquina, ou se as quer instalar na *cloud*. Esta opção encontra-se aqui presente para o caso em que o utilizador queira realizar alguns testes localmente antes de começar a realizar ações na *cloud*. O utilizador pode primeiro querer visualizar a estrutura de ficheiros gerada, para depois eventualmente fazer algumas alterações se desejar. Assim, caso o utilizador escolha a opção de executar as aplicações localmente, então o *script* irá terminar, uma vez que nesta fase os repositórios já foram gerados na máquina local do utilizador, não tendo por isso o *script* nenhuma tarefa adicional para executar. Caso o utilizador escolha a opção de instalar as aplicações na *cloud*, então o *script* irá avançar para a fase seguinte.

6. Nesta fase, o *script* vai utilizar o Terraform para gerar a infraestrutura na AWS. Essa infraestrutura vai ser gerada com base em variáveis definidas nos ficheiros Terraform. Apesar de existir uma configuração já inicialmente preparada, o utilizador terá toda a flexibilidade de fazer as alterações que desejar nos ficheiros Terraform, de forma a gerar uma infraestrutura que cumpra todos os requisitos do seu projeto. O *design* inicial da infraestrutura da AWS é explicado na secção seguinte.

7. Por fim, o último passo é a instalação do Jenkins na infraestrutura da AWS, e a configuração das *pipelines* CI/CD. A instalação do Jenkins é tratada automaticamente pelos *scripts* Terraform que vão provisionar a infraestrutura. No entanto, a configuração das *pipelines* no Jenkins vai ter que ser feita de forma manual pelo utilizador, uma vez que as *pipelines* normalmente são executadas dentro de um repositório alojado algures na *cloud*, e o código não tem como saber a localização desses repositórios. Todavia, os repositórios *back-end* e *front-end* são gerados com *pipelines* de exemplo para tratar do *deploy* das aplicações. Essas *pipelines* de *deploy* vão encapsular as aplicações em *containers* Docker, instalando posteriormente esses *containers* no cluster Kubernetes da infraestrutura da AWS.

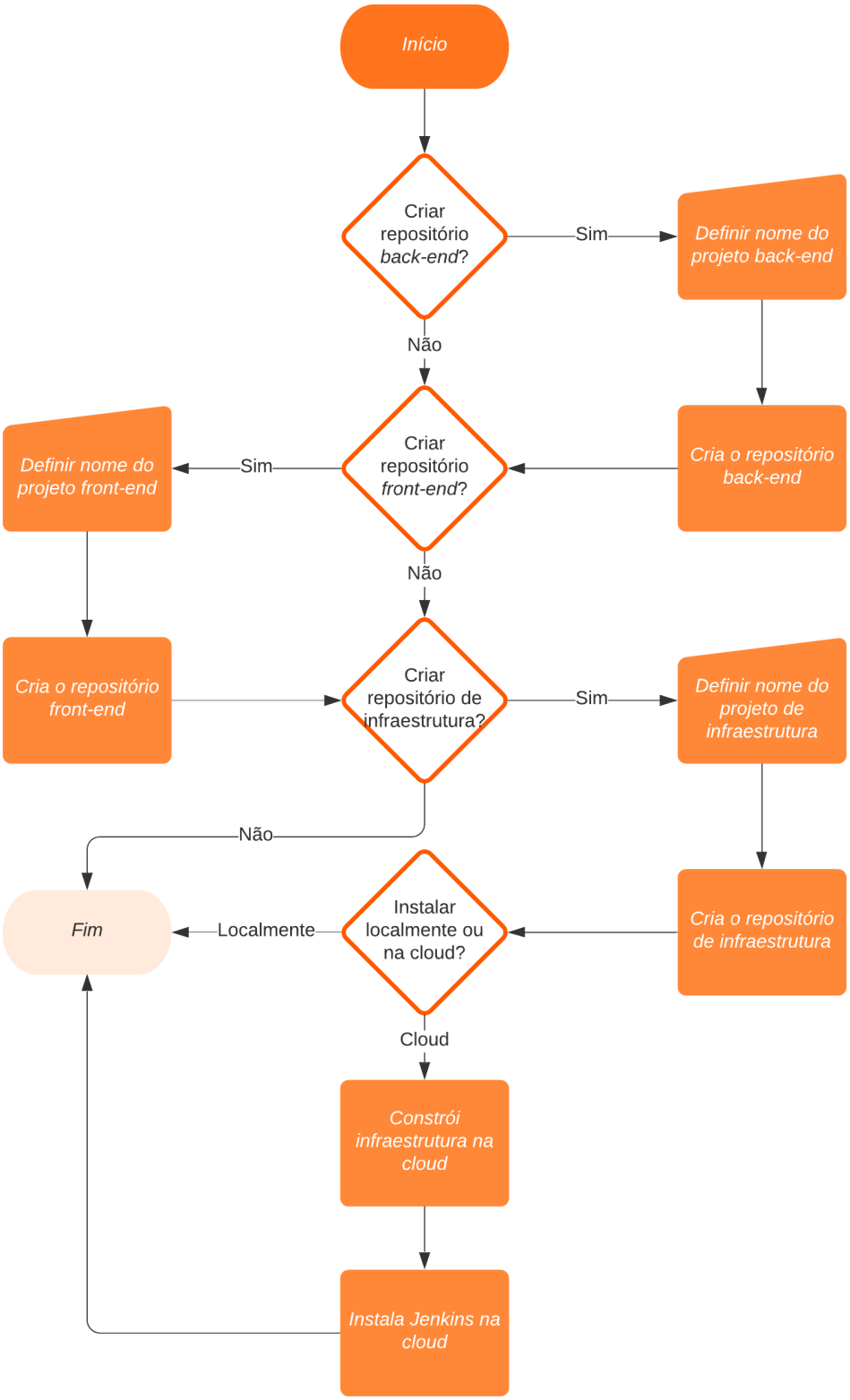

FIGURA 4.2: O fluxograma da solução desenvolvida.

4.4 Arquitetura

Esta secção aborda alguns aspetos da arquitetura do sistema, particularmente da arquitetura do *front-end* e o *back-end*, e da infraestrutura que será gerada na AWS pelos *scripts*.

4.4.1 *Front-end* e *Back-end*

A figura 4.3 representa a arquitetura que é gerada pela solução, sendo esta uma arquitetura cliente-servidor. Esta arquitetura é extremamente comum nos dias de hoje, uma vez que a própria internet, no geral, é um sistema baseado em clientes e servidores. O princípio da separação de conceitos[1] é um tema central desta arquitetura, pois os clientes e servidores têm funções distintas a desempenhar (Masse 2011).

Num sistema implementado com esta arquitetura, o cliente e o servidor comunicam através de pedidos e respostas HTTP, em que habitualmente o cliente requisita informação ao servidor, e o servidor responde com dados que serão consumidos pelo cliente.

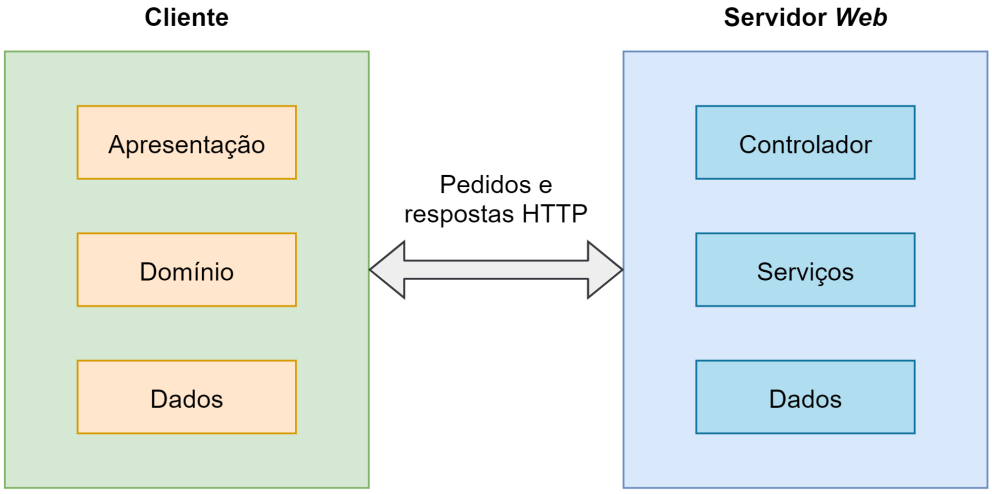

FIGURA 4.3: Arquitetura cliente-servidor do projeto gerado.

No projeto gerado pela solução, o cliente terá uma arquitetura baseada em três camadas:

- **Camada de apresentação:** É responsável por mostrar dados e informação na página *web*, e por gerir as interações do utilizador.
- **Camada de domínio:** É a camada responsável por conter a lógica dos casos de uso da aplicação *front-end*.
- **Camada de dados:** É a camada responsável por estabelecer a comunicação entre o *front-end* e o *back-end*, e por gerir todos os dados que são enviados e recebidos.

Em projetos *full-stack*, o *front-end* normalmente é a componente mais simples. Idealmente, em termos de implementação, o *front-end* apenas tem de enviar e receber dados do *back-end*, não devendo conter muita (ou mesmo nenhuma) lógica de negócio.

O *back-end* será uma API REST, e terá também uma arquitetura baseada em três camadas:

[1] *separation of concerns*

- **Camada do controlador:** É a camada que irá receber os pedidos do cliente, e que terá a responsabilidade de invocar os serviços certos para processar esses pedidos.
- **Camada de serviços:** É a camada que contém toda a lógica de negócio. Esta camada processa os pedidos do cliente, e também tem a função de efetuar chamadas à camada de dados.
- **Camada de dados:** É a camada responsável por interagir com o sistema de persistência de dados da aplicação, como uma base de dados, por exemplo.

Distribuir a implementação de um projeto em camadas, especialmente um *back-end*, é particularmente importante, novamente devido ao princípio da separação de conceitos. Este princípio é um pouco difícil de definir, uma vez que se trata de uma abstração. No entanto, existem boas razões para ser implementado. No fundo, é uma forma de organizar logicamente a aplicação, atribuindo a cada camada uma responsabilidade diferente. Isso vai acabar por ter benefícios em termos de desenvolvimento, não só pelas questões organizacionais, mas também pelo facto de que normalmente é possível que estas camadas possam ser alteradas sem que isso tenha efeito nas outras camadas. Por exemplo, se uma API quiser alterar a sua lógica de negócio, desde que os dados recebidos pelo cliente sejam os mesmos, não terá de haver qualquer tipo de alteração feita na camada do controlador.

4.4.2 Infraestrutura na AWS

Tal como mencionado na secção 2.1, todas as equipas da área de *Infotainment & Interactivity Services* partilham a mesma infraestrutura base, infraestrutura essa que é mantida e desenvolvida por todas as equipas. Já existem atualmente processos automatizados para a criação, manutenção e destruição dessa infraestrutura partilhada, pelo que esta solução não necessita de criar componentes já existentes nessa infraestrutura. Essa infraestrutura partilhada inclui componentes na AWS como *Network Load Balancer*s (NLBs), *Route* 53, *subnets*, entre outros. Ou seja, os novos projetos não têm de criar essa infraestrutura, apenas têm de criar a infraestrutura específica para os seus serviços.

Este projeto precisou essencialmente de quatro componentes principais:

- **Armazenamento de imagens Docker**

  Primeiro, é necessário um local na *cloud* onde seja possível armazenar as imagens Docker da aplicação. Para isso, pode-se usar o Docker Hub, que é a solução de armazenamento de imagens do próprio Docker, ou então um serviço como o Amazon *Elastic Container Registry* (ECR), que é um serviço da AWS que permite o armazenamento, gestão, partilha e *deploy* de imagens de *containers*. Este projeto utilizou o Docker Hub, principalmente por questões de custos durante esta fase de *Proof of Concept* (PoC) e também porque é uma solução semelhante em termos de implementação. Contudo, aquando da sua implementação na empresa, este projeto irá utilizar o Amazon ECR como o serviço que irá alojar as imagens Docker das aplicações geradas pela solução. Este projeto, por defeito, vai gerar duas imagens Docker, uma para o *back-end* e outra para o *front-end*.

- **Amazon *Elastic Kubernetes Service* (EKS)**

  O Amazon EKS é um serviço que permite a criação de *clusters* Kubernetes, de forma a permitir de uma forma fácil a execução e escalonamento de aplicações encapsuladas em *containers*. A solução vai gerar um *cluster* Kubernetes onde tanto o *front-end* como o *back-end* serão alojados.

- **Amazon *Virtual Private Cloud* (VPC)**

  Uma VPC é um seviço que permite a execução de recursos AWS numa rede isolada que pode ser completamente configurada pelo utilizador. É necessário a solução criar a sua própria VPC, uma vez que um *cluster* EKS tem como requisito ter uma VPC e *security group* dedicados.

- **Amazon *Elastic Compute Cloud* (EC2)**

  Por fim, foi ainda gerada uma instância Amazon EC2, que é essencialmente uma máquina virtual, para alojar o Jenkins.

Com base no que foi mencionado, a figura 4.4 mostra a arquitetura que será gerada pelo solução na AWS. Esta infraestrutura é gerada com base no princípio de infraestrutura como código, utilizando para isso *scripts* Terraform para se conseguir automatizar a criação e destruição da toda a infraestrutura.

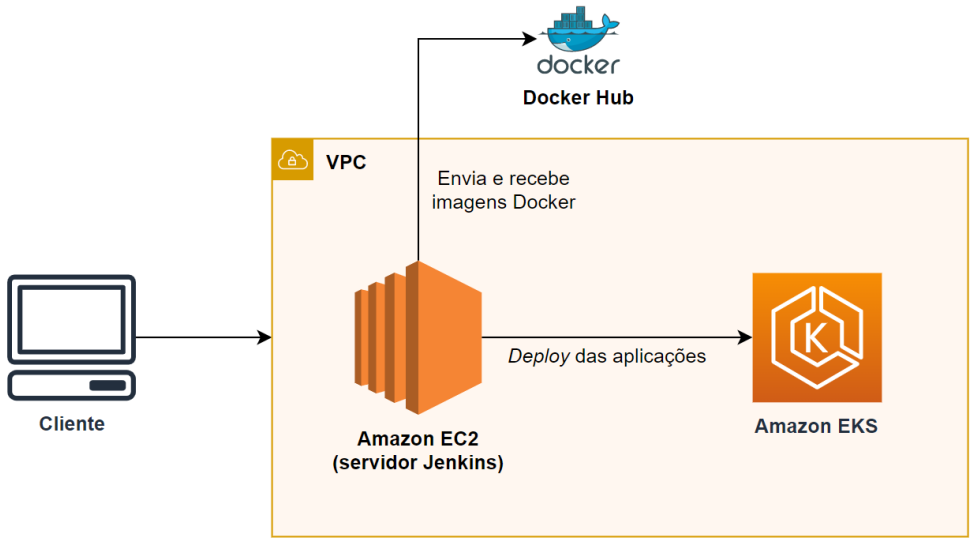

FIGURA 4.4: Infraestrutura do projeto gerado na AWS.

4.5 Especificação da API

Segundo os princípios REST, as operações permitidas por uma API devem ser expostas por um método HTTP e um *Uniform Resource Identifier* (URI). Para além disso, o propósito da operação não deve contradizer o método HTTP utilizado (Bojinov 2015). Ou seja, não deve ser usado o método `GET` para realizar uma atualização a um recurso, por exemplo.

A tabela 4.1 resume os métodos HTTP que habitualmente são mais utilizados, e os fins para os quais são usados.

Tabela 4.1: Explicação dos métodos HTTP.

Método	Descrição
`GET`	Obter informação de um recurso.
`POST`	Criar um novo recurso.
`PUT`	Atualizar a informação de um recurso.
`DELETE`	Apagar um recurso.

Estes são apenas os métodos habitualmente mais usados, e não a totalidade de métodos HTTP existentes. Existem ainda métodos como o `TRACE` ou o `OPTIONS`, por exemplo, mas esses não são utilizados para interagir com dados.

Também é relevante falar da utilização do método `PUT` para a atualização de dados, em vez do `PATCH`, que também é frequentemente usado para o mesmo efeito. A principal diferença entre estes dois métodos é que o `PUT` efetua uma substituição completa, enquanto que o `PATCH` efetua uma atualização parcial. Por outras palavras, o método `PUT` espera que o cliente envie todos os dados do recurso no pedido, fazendo assim uma substituição integral do recurso existente pelo recurso enviado no pedido do cliente. O `PATCH`, por outro lado, apenas espera que o cliente envie no pedido os campos que devem ser atualizados. Ou seja, o servidor faz uma atualização apenas aos campos enviados pelo cliente, deixando os restantes campos do recurso inalterados. Não há necessariamente uma escolha melhor do que outra, pois ambas têm as suas vantagens e desvantagens. Tudo depende da implementação da API, e do comportamento esperado pelo *back-end*. As equipas da Critical TechWorks tendem a preferir a utilização do `PUT`, daí ter sido essa a opção utilizada nesta solução.

Como esta solução apenas tem como objetivo criar código genérico para servir de *template* às equipas, o conteúdo da API não é particularmente relevante, uma vez que este vai ser substituído pelas equipas. O mais importante para as equipas será a implementação e a forma como a API se encontra desenhada. No entanto, para testar todas as funcionalidades da aplicação, era importante criar uma API que permitisse as operações *Create, Read, Update and Delete* (CRUD), para que as equipas possam ter um exemplo completo que auxilie o desenvolvimento dos seus serviços.

Dessa forma, foi idealizada uma API *dummy* para gerir carros da BMW. Essa API irá permitir obter uma lista de vários modelos de carro da BMW, adicionar um novo modelo, atualizar a informação de modelos existentes, e apagar um determinado modelo. A tabela 4.2 representa a especificação das operações permitidas pela API.

Tabela 4.2: Especificação da API.

Método	URI	Descrição
`GET`	`/cars`	Obtém todos os modelos de carro existentes.
`POST`	`/cars`	Cria um novo modelo de carro.
`GET`	`/cars/:carId`	Obtém um único modelo de carro pelo seu ID.
`PUT`	`/cars/:carId`	Atualiza um único modelo de carro pelo seu ID.
`DELETE`	`/cars/:carId`	Elimina um único modelo de carro pelo seu ID.

Tendo a especificação das operações permitidas pela API, definiu-se de seguida o formato utilizado para os dados enviados na comunicação entre as componentes *front-end* e *back-end*. Para isso, foi selecionado o formato JavaScript *Object Notation* (JSON). É um formato suportado nativamente pelo JavaScript, é de fácil leitura por humanos, e permite facilmente a manipulação dos dados. O excerto 4.1 é um exemplo da representação de um carro na API, em formato JSON.

```
{
    "carId": "1",
    "carModel": "3 Series",
    "engineType": "Petrol"
}
```

EXCERTO 4.1: Representação de um carro na API.

4.6 Sumário

Na solução gerada, todo o processo e fluxo das diversas operações é assegurado por *scripts*, que irão garantir o correto funcionamento da solução. Adicionalmente, esses *scripts* vão garantir que a solução tem comportamentos diferentes, com base nas escolhas definidas pelos utilizadores. Dessa forma, oferece-se alguma flexibilidade às equipas de desenvolvimento, garantindo que esta solução consiga mais facilmente atingir os seus objetivos.

A comunicação entre *front-end* e *back-end* irá basear-se na comum arquitetura cliente-servidor, onde a comunicação entre estas duas componentes será realizada por pedidos e respostas HTTP. Em termos de desenvolvimento de *software*, tanto o *front-end* como o *back-end* irão ter uma arquitetura baseada em camadas, adotando assim a boa prática do princípio da separação de conceitos e responsabilidades.

O *back-end* será constituído por uma API REST, que será apenas gerada para servir de exemplo e *template* para as equipas. Essa API *dummy* terá como recurso principal uma lista de modelos de carros da BMW, permitindo a realização de operações CRUD sobre esse recurso.

Capítulo 5

Implementação

Este capítulo explora a implementação e o desenvolvimento da solução em mais detalhe. Tal como mencionado previamente neste documento, esta solução funciona à base de *scripts*. Existe o *script* principal, que é o único *script* que o utilizador terá de invocar, e depois existem os *scripts* secundários que serão invocados pelo *script* principal dependendo das escolhas do utilizador, sendo assim completamente transparentes para o mesmo.

O capítulo é dividido em três secções. A primeira secção aborda os *scripts* secundários e detalha cada um deles individualmente. A segunda secção descreve o *script* principal, e o processo de como este junta todos os *scripts* secundários para formar a solução final. A terceira e última secção detalha as pipelines CI/CD que foram desenvolvidas, bem como a configuração do Jenkins.

5.1 *Scripts* Secundários

A implementação da solução começou com o desenvolvimento dos *scripts* secundários para gerar os repositórios individuais. No total eram necessários três repositórios: um para a componente *back-end*, outro para a componente *front-end* e, finalmente, um para a o código da infraestrutura necessária. Dessa forma, foram desenvolvidos três *generators* Yeoman, um para gerar cada repositório.

5.1.1 *Back-end*

Desenvolvimento da API

O desenvolvimento do repositório *back-end* começou pelo desenvolvimento da API, cuja especificação se encontra na secção 4.5 deste documento. Também como já mencionado, a componente do *back-end* foi desenvolvida utilizando Node.js, mais especificamente a *framework* Express.

Tal como abordado na secção 4.4, o *back-end* iria ser baseado numa arquitetura em três camadas:

- **Camada do controlador**

  Esta é a camada que irá receber os pedidos do cliente, e que terá a responsabilidade de invocar os serviços certos para processar esses pedidos. O excerto 5.1 é uma parte do ficheiro principal do *back-end*, aquele que é o ponto de entrada para a API.

```
1 import express from "express";
2 import carsRoutes from "./api/routes/cars.js";
3
4 const app = express();
5
6 app.use("/cars", carsRoutes);
```

EXCERTO 5.1: Código parcial do ficheiro de entrada da API.

A linha 4 está a inicializar uma nova aplicação Express. A linha 6 está a definir um *endpoint* para esta aplicação: o *endpoint* `/cars`. O segundo argumento da linha 6 é o ficheiro `carsRoutes`, importado na linha 2, que contém todas as rotas do *endpoint* `/cars`.

O excerto 5.2 é o conteúdo do ficheiro `carsRoutes`, utilizado no excerto anterior.

```
1 import express from "express";
2 import { createCar, deleteCar, getCar, getCars, updateCar } from "../
      services/cars.js";
3
4 const router = express.Router();
5
6 router.get("/", getCars);
7 router.post("/", createCar);
8 router.get("/:id", getCar);
9 router.put("/:id", updateCar);
10 router.delete("/:id", deleteCar);
11
12 export default router;
```

EXCERTO 5.2: Código das rotas do *endpoint* `/cars`.

Como se pode ver, é um ficheiro simples e de fácil compreensão, algo que demonstra as vantagens da arquitetura utilizada. Desta forma, esta camada não necessita de ter qualquer lógica de negócio. A única responsabilidade desta camada é definir as rotas disponíveis, e invocar os serviços respetivos para cada rota. Na linha 2 estão a ser importados cinco métodos de um ficheiro na camada de serviços. Depois, da linha 6 à linha 10, são criadas as rotas disponíveis para o *endpoint* `/cars`, tendo como segundo argumento o método que vai ser executado com base na rota invocada. Por exemplo, um pedido do tipo `DELETE` ao *endpoint* `/cars/123`, vai executar o método `deleteCar()` e vai passar o ID "123" como parâmetro de entrada para esse método.

- **Camada de serviços**

  Esta é a camada que contém toda a lógica de negócio. Esta camada processa os pedidos do cliente, e também tem a função de efetuar chamadas à camada de dados.

  O excerto 5.3 mostra o método `getCars()` do ficheiro da camada de serviços, importado e utilizado no excerto anterior.

```
1 export const getCars = (req, res) => {
2     return res.send(cars);
3 };
```

EXCERTO 5.3: Método `getCars()` da camada de serviços.

Este é um método bastante simples que simplesmente retorna uma resposta com o vetor `cars`, sendo este vetor constituído por todos os carros que estão guardados. Este método é executado quando a API recebe um pedido `GET` no *endpoint* `/cars`.

- **Camada de dados**

  Esta é a camada responsável por interagir com o sistema de persistência de dados da aplicação. No caso desta API, devido a razões já mencionadas neste documento, não foi utilizada uma base de dados uma vez que o objetivo deste projeto é fornecer uma solução genérica que possa ser utilizado e iterado por qualquer equipa, e a escolha de uma base de dados é uma decisão que vai depender das necessidades específicas de cada equipa. Dessa forma, a persistência de dados está a ser feita na memória da aplicação, uma vez que isso serve bem os requisitos da solução.

  Devido a isso, para o exemplo desta API, a camada de dados é essencialmente inexistente. O excerto 5.4 representa os dados iniciais da API quando a aplicação é inicializada.

```
export let cars = [
    {
        id: "2c208cc8-978c-4c79-bc2d-f02b99cf5a54",
        model: "1 Series",
        color: "Blue",
        engineType: "Petrol"
    },
    {
        id: "61154091-6482-409b-9650-308255c58185",
        model: "3 Series",
        color: "Red",
        engineType: "Diesel"
    }
];
```

EXCERTO 5.4: Dados iniciais da API.

No entanto, os métodos da camada de serviços vão manusear este vetor, tornando assim possível todas as operações CRUD.

Apesar de não haver uma persistência de dados real numa base de dados, a API possui validação de entidades para garantir que apenas objetos válidos são aceites. O excerto 5.5 mostra a validação implementada, utilizando para isso a biblioteca Joi.

```
import Joi from "joi";

const carSchema = Joi.object({
    model: Joi.string().required(),
    color: Joi.string().required(),
    engineType: Joi.string().required()
});

export const validateCar = (car) => {
    return carSchema.validate(car);
};
```

EXCERTO 5.5: Validação de um carro.

Como se pode analisar, apenas são aceites na API entidades que cumpram o *schema* definido. Ou seja, entidades que tenham um `model`, uma `color` e um `engineType`, todos do tipo `string`.

Para além do código desenvolvido para o funcionamento da API, foram também desenvolvidos testes para garantir o correto funcionamento de todos os tipos de pedidos. Para a realização dos testes foram utilizadas as bibliotecas Mocha e Chai, que são duas bibliotecas de testes muito utilizadas no universo JavaScript.

O excerto 5.6 é um exemplo de um dos testes que foi desenvolvido. Neste caso, é um teste de insucesso para um pedido `POST`. O Chai está a ser utilizado para efetuar um pedido à API, sendo que nesse pedido está a ser enviado um objeto inválido no *body*, representado pela variável `invalidCarBody`. Este objeto inválido é um objeto que apenas possui as propriedades `model` e `engineType`, faltando por isso a propriedade `color`. Dessa forma, da linha 6 à linha 12 está indicado que o teste espera receber uma resposta de erro com um código HTTP 400 (ou seja, um código `Bad Request`), e com uma mensagem de erro a referir que a propriedade `color` é obrigatória.

```
1  it("should not create a new car with an invalid body", (done) => {
2      chai.request(app)
3          .post(carsApiEndpoint)
4          .send(invalidCarBody)
5          .end((err, res) => {
6              res.should.have.status(400);
7              res.body.should.be.a("object");
8              res.body.should.have.property("error");
9              res.body.should.have.nested.property("error.status").eq(400);
10             res.body.should.have.nested
11                 .property("error.message")
12                 .eq(ERROR_MESSAGES.COLOR_IS_REQUIRED);
13             done();
14         });
15 });
```

EXCERTO 5.6: Exemplo de um teste da API.

A figura 5.1 é uma execução real de todos os testes desenvolvidos, sendo que todos foram completados com sucesso.

```
Cars API
  GET /cars
    √ should get all the cars
  POST /cars
    √ should create a new car with a valid body
    √ should not create a new car with an invalid body
  GET /cars/:id
    √ should get a car if the ID exists
    √ should not get a car if the ID doesn't exist
  PUT /cars/:id
    √ should update a car with a valid ID and valid body
    √ should not update a car with a valid ID and invalid body
    √ should not update a car with an invalid ID
  DELETE /cars/:id
    √ should delete a car if the ID exists
    √ should not delete a car if the ID doesn't exist


10 passing (108ms)
```

FIGURA 5.1: Testes da API.

Desenvolvimento dos Ficheiros Kubernetes

O objetivo desta solução é gerar um projeto cujas aplicações estejam a ser executadas num *cluster* Kubernetes na AWS. Dessa forma, após a implementação da aplicação em si, iniciou-se o desenvolvimento dos ficheiros Kubernetes.

Em termos de componentes Kubernetes, existem três componentes importantes para a implementação desta aplicação:

1. ***Pod*:** O *pod* é o componente Kubernetes onde as aplicações estão realmente a ser executadas. Cada *pod* contém um ou mais *containers*, sendo que no caso desta aplicação cada *pod* apenas terá um *container*.
2. ***Deployment*:** Um *deployment* é o componente Kubernetes responsável por provisionar e gerir os *pods* de um serviço. Por exemplo, caso algo aconteça a um *pod* que o faça terminar inadvertidamente, o *deployment* irá automaticamente provisionar um novo *pod* para substituir o que foi terminado.
3. ***Service*:** Um *service* é um componente Kubernetes que tem um endereço IP estático e funciona como um *load balancer* para os *pods*. Cada *pod* tem o seu endereço IP privado, mas não é boa prática utilizar este endereço para comunicar com a aplicação pois os *pods* podem ser terminados e provisionados várias vezes, sendo que o seu endereço IP ia estar constantemente a mudar. O *service* existe para resolver esse mesmo problema, sendo que está uma camada acima dos *pods* e tem um endereço estático. Ou seja, caso uma aplicação *front-end* queira comunicar com a API do *back-end*, essa aplicação *front-end* usaria o endereço IP do *service*, e depois o *service* iria tratar de reencaminhar esses pedidos para todos os *pods* associados a esse serviço.

Assim, para posteriormente ser possível fazer o *deploy* da aplicação *back-end* para um *cluster* Kubernetes, seria necessário criar um *service* e um *deployment*, sendo que depois o *deployment* iria tratar de provisionar os *pods*. Os componentes Kubernetes são configurados em ficheiros do tipo

YAML *Ain't Markup Language* (YAML), e é possível juntar vários componentes no mesmo ficheiro. O excerto 5.7 mostra o conteúdo do ficheiro YAML com a configuração Kubernetes da aplicação *back-end*.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: fullstack-archetype-backend
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: fullstack-archetype-backend
10   template:
11     metadata:
12       labels:
13         app: fullstack-archetype-backend
14     spec:
15       containers:
16         - name: fullstack-archetype-backend
17           image: telmof/fullstack-archetype-backend
18           ports:
19             - containerPort: 4000
20 ---
21 apiVersion: v1
22 kind: Service
23 metadata:
24   name: fullstack-archetype-backend
25 spec:
26   selector:
27     app: fullstack-archetype-backend
28   ports:
29     - port: 4000
30       targetPort: 4000
```

EXCERTO 5.7: Configuração Kubernetes do *back-end*.

Este ficheiro contém a configuração de um *deployment* e de um *service*, sendo que as duas configurações estão divididas pela linha 20. A partir da linha 5, na secção `spec`, são definidas as especificações dos *pods* que vão ser gerados a partir deste *deployment*. A linha 6 indica o número de *pods* que vão ser gerados. A linha 9 é a *label* que vai associar o *deployment* ao *service*, sendo que a *label* da linha 9 tem de ser igual à da linha 27. A partir da linha 15, na secção `containers`, são definidos os *containers* que vão estar em cada *pod*. Neste caso vai ser apenas um *container* com a imagem Docker da aplicação desenvolvida, que vai estar a correr na porta 4000. Finalmente, a partir da linha 21 é definida a configuração do *service*, que vai ter um endereço IP estático e vai ser executado na porta 4000, tal como indicado pela linha 29. O `targetPort` da linha 30 é a porta do *container* associado ao *service*, que deverá sempre ser igual à porta da linha 19.

Desenvolvimento do *Generator*

Um *generator* Yeoman é essencialmente um módulo Node.js. Para o desenvolvimento do *generator* propriamente dito, o Yeoman oferece uma classe `Generator` que contém já algumas das funcionalidades necessárias. Assim, toda a implementação do *generator* estará dentro de uma classe que estende esta classe base, tal como demonstrado pelo exemplo do excerto 5.8.

```
const Generator = require("yeoman-generator");

module.exports = class extends Generator {
  method1() {
    this.log('method 1 just ran');
  }

  method2() {
    this.log('method 2 just ran');
  }
};
```

EXCERTO 5.8: Exemplo de um *generator*.

Para garantir o correto funcionamento dos seus *generators*, o Yeoman utiliza um ciclo de execução, ou *run loop*. Este ciclo de execução é um sistema de filas ordenado por prioridades. As prioridades são definidas no código através dos nomes dos métodos utilizados. Se o nome de um método corresponder a uma das prioridades, então esse método é enviado para uma certa posição na fila do ciclo de execução. Se o nome do método não corresponder a nenhuma das prioridades pré-definidas, então esse método é incluído no grupo `default`. Estas são as prioridades pré-definidas pelo Yeoman, já devidamente ordenadas:

1. `initializing`: Os métodos de inicialização. Por exemplo: verificar o estado do projeto atual, inicializar configurações, etc.
2. `prompting`: Aqui fazem-se as interações com os utilizadores.
3. `configuring`: Guardar ou alterar configurações do projeto
4. `default`: Se um método não for igual a nenhuma das outras prioridades, então esse método é incluído neste grupo.
5. `writing`: Onde são escritos os ficheiros gerados pelo *generator*.
6. `conflicts`: Onde os conflitos são geridos e tratados.
7. `install`: Aqui fazem-se as instalações necessárias, como instalações do NPM, por exemplo.
8. `end`: O último grupo a ser chamado. Aqui fazem-se as últimas ações do *generator*, como limpar ficheiros, por exemplo.

Ou seja, caso um *generator* tenha um método chamado `prompting()` e outro método chamado `configuring()`, o método `prompting()` vai ser ser executado primeiro, pois possui uma ordem maior de prioridade.

Apesar de ser possível implementar bastante lógica num *generator* Yeoman, o objetivo deste projeto era desenvolver algo simples, algo que depois as equipas de desenvolvimento consigam alterar facilmente sem primeiro terem de estudar as especificidades do Yeoman. Com isso em mente, foi desenvolvido um *generator* constituído apenas por dois métodos principais: o método `prompting()` para interagir com o utilizador no início da execução do *generator*, e o método `writing()` para fazer o *scaffolding* da aplicação.

O método `prompting()`, no caso deste *generator*, apenas necessita de questionar o utilizador sobre o nome que este deseja dar ao seu projeto. O excerto 5.9 mostra o código deste método.

```
async prompting() {
    this.log(
        yosay("Welcome to the " + chalk.red("fullstack-archetype") + "
    backend generator!")
    );

    this.answers = await this.prompt([
        {
            type: "input",
            name: "name",
            message: "What is your project name?",
            default: this.appname
        }
    ]);
}
```

EXCERTO 5.9: Método `prompting()` do *generator*.

O método começa com um bloco `this.log()` onde aparece uma mensagem a cumprimentar o utilizador. Depois, na linha 6, é utilizado o método `this.prompt()` para questionar algo ao utilizador, sendo que depois a resposta é guardada na variável `this.answers.name`. Neste caso, o programa questiona ao utilizador o nome que este quer dar ao projeto *back-end* que vai gerar. Esse nome é depois utilizado no método `writing()`, que é próximo passo do *generator*.

O excerto 5.10 mostra o código do método `writing()`.

```
writing() {
    this.destinationRoot(this.answers.name);
    this.appname = this.answers.name;

    this.copyTemplate("_package.json", "package.json", {name: this.answers.name});
    this.copyTemplate("_package-lock.json", "package-lock.json", {name: this.
    answers.name});
    this.copyTemplate("_prettierrc.json", "prettierrc.json");
    this.copyTemplate("Dockerfile", "Dockerfile");
    this.copyTemplate("gitignore", ".gitignore");
    this.copyTemplate("prettierignore", ".prettierignore");
    this.copyTemplate("README.md", "README.md", {name: this.answers.name});
    this.copyTemplate("src", "src");
    this.copyTemplate("test", "test");
    this.copyTemplate("pipelines", "pipelines", {name: this.answers.name});
    this.copyTemplate("kubernetes", "kubernetes", {name: this.answers.name});
}
```

EXCERTO 5.10: Método `writing()` do *generator*.

As linhas 2 e 3 criam uma pasta com o nome definido pelo utilizador no método anterior, que irá conter todo o código gerado pelo *generator*. Depois, da linha 5 à linha 15, são copiados os ficheiros de *template*. Esta é a estrutura de ficheiros do repositório deste *generator*:

```
package.json
package-lock.json
README.md
app/
    index.js
    templates/
        kubernetes/
        pipelines/
        src/
        test/
        _package.json
        _package-lock.json
        Dockerfile
        gitignore
        README.md
```

O método `this.copyTemplate()`, utilizado no excerto 5.10, vai automaticamente procurar ficheiros na pasta `templates/`. Esse método tem três argumentos: o primeiro é o nome do ficheiro ou pasta a ser copiado; o segundo é o nome que o ficheiro ou pasta deve ter depois de ser copiado; e o último é um argumento opcional onde podem ser passadas algumas opções. A pasta de destino para onde este método vai copiar os ficheiros é definida pela variável `this.destinationRoot`, que contém o nome selecionado pelo utilizador no excerto 5.9.

Dando um exemplo da utilização do método `this.copyTemplate()`, na linha 5 este método vai selecionar o ficheiro `templates/_package.json` (que está com o *underscore* para não criar conflitos com o verdadeiro `package.json` do próprio *generator*), e vai copiá-lo para a pasta de destino com o nome `package.json`. Para além disso, neste caso é passada a variável `name` como terceiro parâmetro, que vai ter o valor guardado na variável `this.answers.name`. Isso vai informar o Yeoman que deve substituir a variável `name` no conteúdo do ficheiro pelo nome que está guardado na variável `this.answers.name`. Depois, o Yeoman vai procurar no ficheiro pela expressão `<%= name %>` e vai substituí-la pelo valor de `this.answers.name`. Se tivesse sido passada a expressão `{info: this.answers.name}`, por exemplo, como terceiro parâmetro, então o Yeoman ia procurar a expressão `<%= info %>` para substituir pelo valor da variável `this.answers.name`.

O excerto 5.11 mostra o conteúdo parcial do ficheiro `_package.json`. A linha 2 mostra a variável que vai ser substituída pelo conteúdo da variável `this.answers.name`.

```
1 {
2   "name": "<%= name %>",
3   "version": "0.0.1",
4   "main": "src/app.js",
5   "type": "module",
6   (...)
7 }
```

EXCERTO 5.11: Conteúdo parcial do ficheiro `_package.json`.

Todo o código da componente *back-end*, isto é, todo o código que vai ser gerado pelo *generator*, foi pensado e desenvolvido de forma a ser o mais genérico e simples possível, mas mesmo assim tendo todas as funcionalidades necessárias de uma API REST. É por essa razão que o nome do projeto é a única variável definida pelo utilizador, sendo que toda a lógica da aplicação é gerada tal como se encontra nos ficheiros de *template*, de forma a proporcionar uma experiência de

utilização fácil e acessível. Caso o *generator* fosse mais complicado e tivesse várias opções de utilização, uma nova equipa poderia sentir-se sobrecarregada pela complexidade da ferramenta e isso poderia desincentivar a utilização deste *generator* ao invés de uma solução mais tradicional.

A execução deste *generator*, bem como o resultado obtido, será discutido em mais detalhe na secção 5.2 deste documento.

5.1.2 *Front-end*

Desenvolvimento da Aplicação *Web*

O desenvolvimento da componente *front-end* do projeto envolveu o desenvolvimento de uma aplicação *web* que conseguisse comunicar com a API do *back-end*, tanto para obter dados como também para fazer alterações nesses dados e criar dados novos. A tecnologia selecionada para o desenvolvimento da componente *front-end*, tal como já foi mencionado neste documento, foi a biblioteca React.

Na secção 4.4 foi mencionado que a componente *front-end* iria ser baseada numa arquitetura organizada em três camadas: a camada de apresentação, a camada de domínio e a camada de dados. Contudo, não é obrigatório que estas camadas estejam isoladas em ficheiros diferentes. Numa perspetiva de implementação em React, por vezes, faz todo o sentido unir as camadas de apresentação e domínio nos mesmos ficheiros, uma vez que dessa forma torna-se possível manter toda a lógica de um componente no mesmo ficheiro, diminuindo assim a complexidade da estrutura de ficheiros do repositório, ao invés de dividir a parte de apresentação e a lógica de um componente em ficheiros diferentes.

- **Camada de apresentação e camada de domínio**

  A camada de apresentação é responsável pela apresentação da página *web*, e por gerir as interações do utilizador. A camada de domínio é responsável por conter a lógica dos casos de uso da aplicação. Tal como indicado no parágrafo anterior, durante a implementação do código *front-end* optou-se por uma estruturação do projeto por componentes, sendo que cada componente terá apenas um ficheiro que inclui tanto a parte da apresentação como toda a sua lógica.

  Considere-se, por exemplo, o componente `Car`, que representa um carro. Tal como já mencionado, a entidade do carro na API é um modelo JSON composto por três atributos: `model`, `color` e `engineType`. O excerto 5.12 representa o código do componente `Car` da aplicação.

```
1  import { FaTimes } from "react-icons/fa";
2
3  const Car = ({ car, onDelete, onGet }) => {
4      return (
5          <div id={`car-${car.id}`}
6              data-testid={`car-${car.id}`}
7              className="car"
8              onDoubleClick={() => onGet(car.id)}>
9              <h3>
10                 {car.model}
11                 <FaTimes id={`delete-icon-${car.id}`}
12                     data-testid={`delete-icon-${car.id}`}
13                     onClick={() => onDelete(car.id)}/>
14             </h3>
15             <p>{car.color}</p>
16             <p>{car.engineType}</p>
17         </div>
18     );
19 };
20
21 export default Car;
```

EXCERTO 5.12: Componente `Car`.

Como se pode ver pelo código, a parte visual deste componente é um elemento HTML `<div>`, que por sua vez contém três sub-elementos: um elemento `<h3>` com o modelo do carro, um elemento `<p>` com a cor do carro, e um elemento `<p>` com o tipo de motor do carro. Para além disso, o componente também inclui alguma lógica, nomeadamente a lógica para eliminar esse carro. Na linha 11 é utilizado o elemento `<FaTimes>`, que é importado na linha 1, e este elemento é um ícone de uma cruz que vai servir como um botão para eliminar carros. Depois, na linha 13, é utilizado o evento do React `onClick` que vai executar o método `onDelete()` cada vez que o utilizador pressionar nesse ícone, que por sua vez vai eliminar esse carro do *back-end*. A figura 5.2 mostra um exemplo deste componente na aplicação *web*.

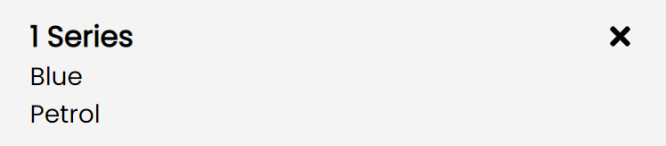

FIGURA 5.2: Componente `Car`.

- **Camada de dados**

  A camada de dados, tal como já indicado, é a camada responsável por estabelecer a comunicação entre o *front-end* e o *back-end*, bem como gerir todos os dados que são enviados e recebidos.

  Continuando o exemplo anterior do componente `Car`, depois de o utilizador pressionar no botão de eliminar um carro, a aplicação *web* vai enviar esse pedido para o *back-end*. Em termos de estruturação de código, a camada de dados foi colocada num ficheiro separado dos componentes. A razão para essa decisão é que esta camada iria conter todos os métodos para comunicação com a API, métodos esses que poderiam ser reutilizados por vários componentes.

Tal como mostrado pela linha 13 do excerto 5.12, quando o utilizador pressiona no ícone de eliminar um carro, é executado o método `onDelete()`. Este método, por sua vez, vai executar um método chamado `deleteOne()` que se encontra no ficheiro da camada de dados. O excerto 5.13 mostra o conteúdo parcial deste ficheiro, incluindo o método `deleteOne()`.

```
1  import axios from "axios";
2  import { handleError, handleSuccess } from "../utils/ApiUtils";
3
4  const carsApiPath = "/api/cars/";
5
6  export const deleteOne = async (id) => {
7      return await axios
8          .delete(carsApiPath + id)
9          .then((response) => handleSuccess(response))
10         .catch((error) => handleError(error));
11 };
```

EXCERTO 5.13: Conteúdo parcial da camada de dados.

A linha 1 está a importar o `axios`, que é uma biblioteca popular de JavaScript para se efetuar pedidos HTTP. A linha 2 está a importar dois métodos criados numa outra classe, com lógica para se o pedido for um sucesso e para se o pedido originar qualquer tipo de erro. A linha 4 explicita qual o *endpoint* que será invocado. E depois, finalmente, na linha 6 começa o método `deleteOne()`, recebendo como parâmetro o ID do carro a ser eliminado. Esse método simplesmente vai invocar o método `axios.delete()`, significando assim que irá fazer um pedido HTTP do tipo `DELETE` para a API do *back-end*, sendo que depois vai utilizar os dois métodos previamente descritos para o caso de sucesso ou erro.

Esta metodologia foi aplicada para o resto do desenvolvimento da aplicação *web*. Cada componente teria o respetivo código da sua parte visual e da lógica interna desse componente, sendo que depois todas as comunicações com o *back-end* passariam pelo ficheiro da camada de dados.

Passando agora para uma apresentação da aplicação *web* desenvolvida, bem como as suas funcionalidades, a figura 5.3 mostra a página inicial da aplicação. Como se pode verificar, é uma página bastante simples, constituída pelo título da aplicação, um botão para adicionar um novo carro, e a lista dos carros existentes.

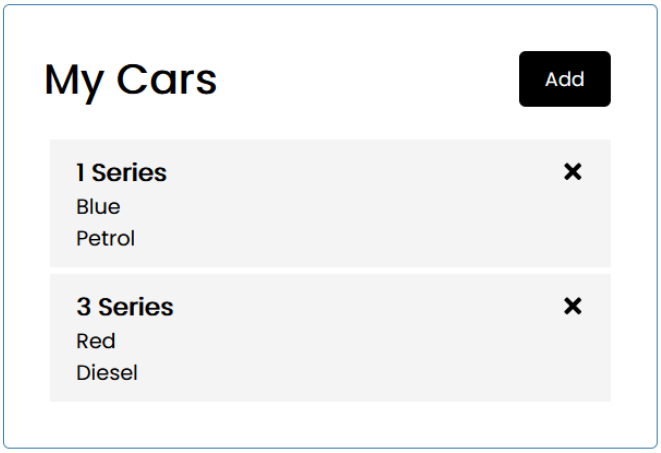

FIGURA 5.3: Página inicial da aplicação.

Pressionando o botão "Add", aparece na página principal um formulário para adicionar um novo carro à coleção, tal como mostra a figura 5.4.

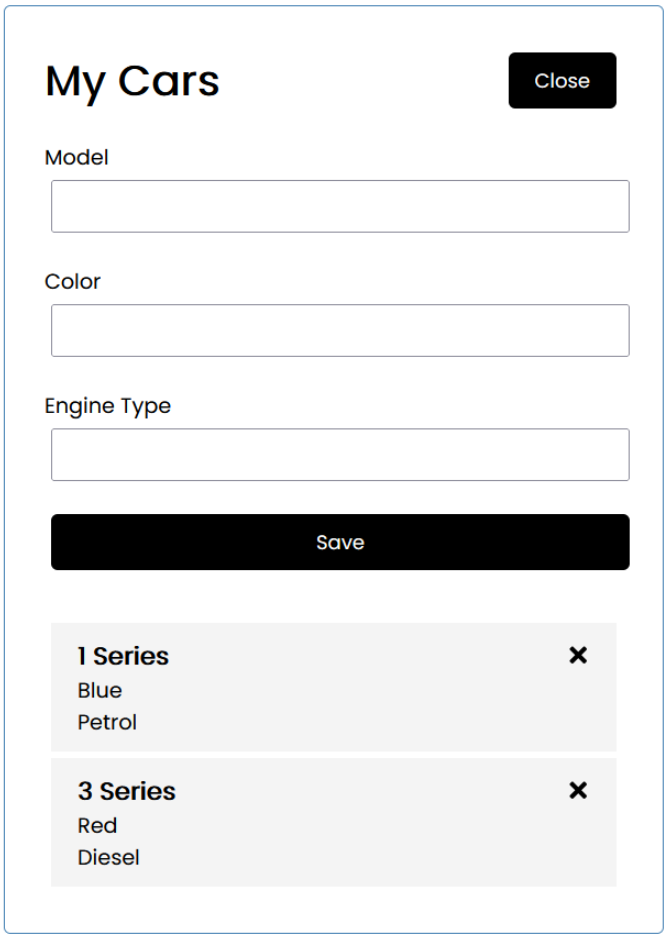

FIGURA 5.4: Formulário para adicionar um novo carro.

Quando o formulário está aberto, o texto do botão "Add" é transformado para "Close", de modo a indicar ao utilizador que pode pressionar o mesmo botão para fechar o formulário. O utilizador pode digitar os dados do novo carro e depois pressionar o botão "Save" para adicionar esse carro à lista.

A aplicação tem ainda a funcionalidade de editar um carro. Para isso, foi reutilizado o mesmo formulário para adicionar um novo carro, mas a forma de abrir o formulário é distinta. Para adicionar um novo carro, o utilizador pressiona o botão "Add" e o formulário abre em branco. Para editar um carro existente, o utilizador terá de fazer um duplo clique sobre o carro que quer editar, e o formulário é aberto já pré-preenchido com os dados atuais do carro. A figura 5.5 foi o resultado de um duplo clique no primeiro carro da lista.

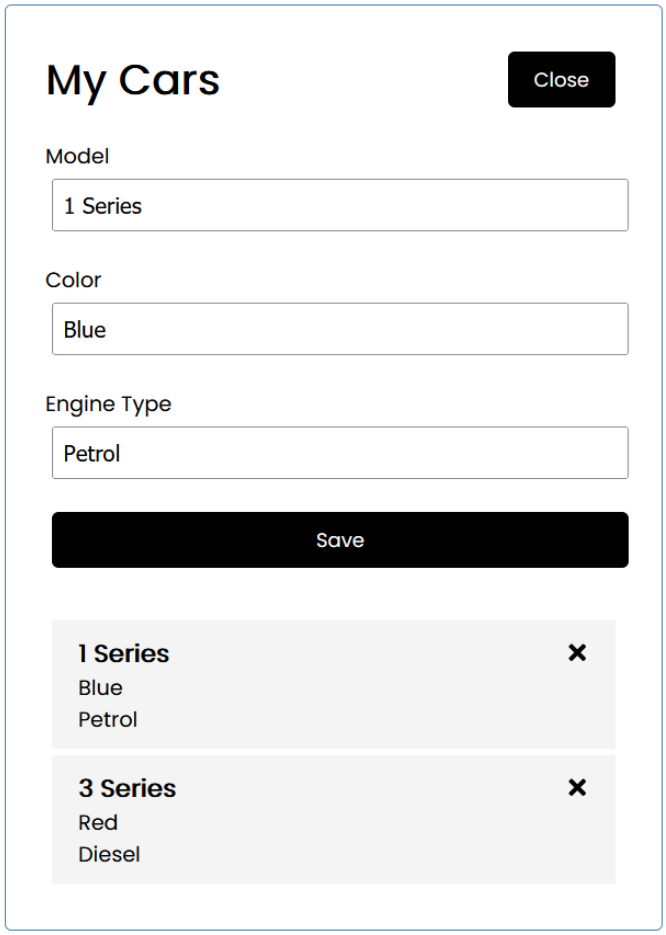

FIGURA 5.5: Formulário para editar um carro existente.

O utilizador poderá depois editar os campos que desejar, e pressionar o botão "Save" para guardar as alterações.

Finalmente, a aplicação tem ainda a função de apagar um carro, que já foi discutida previamente.

Tal como com a componente *back-end*, foram também desenvolvidos testes para garantir o correto funcionamento de todas as funcionalidades da aplicação *web*. Para a realização dos testes unitários foi utilizada a biblioteca de testes Jest, que foi desenvolvida pela empresa Facebook propositadamente para ser utilizada juntamente com o React.

O excerto 5.14 mostra apenas alguns dos testes que foram desenvolvidos para o componente `Car`. No primeiro teste, por exemplo, é utilizado o método `render()` para construir o componente, e depois é testado se o ícone de eliminar o carro está presente no componente. Os dois testes seguintes testam se os métodos corretos são chamados quando o utilizador efetua as ações necessárias.

```
it("should render the delete icon", () => {
    render(<Car car={car} />);
    const icon = screen.getByTestId("delete-icon-1");
    expect(icon).toBeInTheDocument();
});

it("should call the onDelete function one time when clicking the delete icon", ()
    => {
    render(<Car car={car} onDelete={onDeleteMock} />);
    const icon = screen.getByTestId("delete-icon-1");
    expect(icon).toBeInTheDocument();
    userEvent.click(icon);
    expect(onDeleteMock.mock.calls.length).toBe(1);
});

it("should call the onGet function one time when double clicking the car", () => {
    render(<Car car={car} onGet={onGetMock} />);
    const element = screen.getByTestId("car-1");
    userEvent.dblClick(element);
    expect(onGetMock.mock.calls.length).toBe(1);
});
```

EXCERTO 5.14: Exemplo de testes ao componente `Car`.

No total foram desenvolvidos 21 testes unitários para testar todos os componentes desenvolvidos. A figura 5.6 é uma execução real de todos os testes unitários desenvolvidos, sendo que todos foram executados com sucesso.

```
 PASS  src/components/Button/Button.UT.test.js
 PASS  src/components/Car/Car.UT.test.js
 PASS  src/components/CarForm/CarForm.UT.test.js
 PASS  src/components/App/App.UT.test.js
 PASS  src/components/Header/Header.UT.test.js
 PASS  src/components/Cars/Cars.UT.test.js

Test Suites: 6 passed, 6 total
Tests:       21 passed, 21 total
Snapshots:   0 total
Time:        4.063 s
Ran all test suites matching /UT.test.js/i.
```

FIGURA 5.6: Testes unitários da aplicação *web*.

Para além dos testes unitários, foram também criados testes funcionais para fazer testes E2E e para testar a UI da aplicação *web*. Para a realização destes testes funcionais foi utilizada a ferramenta Puppeteer, que é uma biblioteca Node capaz de emular um navegador de internet Chrome ou Chromium. Em termos práticos, é uma biblioteca capaz de abrir uma página num navegador de internet e interagir com esta página da mesma forma que um humano o faria, interagindo com os elementos de UI da página (como pressionar botões, por exemplo).

Foram desenvolvidos quatro testes funcionais, um para cada operação CRUD. Estes testes vão abrir a página da aplicação *web* e vão efetuar pedidos reais à API do *back-end*, tal como um utilizador verdadeiro faria. O excerto 5.15 mostra um exemplo de um dos testes funcionais que foram desenvolvidos, nomeadamente o teste para criar um novo carro.

```
1  describe("Functional tests", () => {
2      beforeAll(async () => {
3          const url = "http://localhost:3000/";
4          page = await initializeBrowser(url);
5      });
6
7      it("should create a new car", async () => {
8          await clickButton("#add-button");
9          await typeInput("#model-input", "4 Series");
10         await typeInput("#color-input", "Green");
11         await typeInput("#engine-type-input", "Petrol");
12         await clickButton("#form-submit-button");
13
14         await page.waitForSelector("#cars-component");
15         const carRowsAfterCreation = await countRows(page);
16
17         expect(carRowsAfterCreation).toBe(startingCarRows + 1);
18     });
19
20     afterAll(async () => {
21         closeBrowser();
22     });
23 });
```

EXCERTO 5.15: Exemplo de um teste funcional.

O excerto começa pelo método `beforeAll()`, que é um método que é executado antes de todos os testes. Neste caso, o método vai abrir o navegador de internet na página onde a aplicação *web* está a correr localmente. Depois, na linha 7, começa o teste propriamente dito. Na linha 8 é utilizado o método `clickButton()`, que vai fazer pressionar o botão que for passado como parâmetro, sendo que neste caso está a ser passado o ID do botão "Add" que já foi mencionado. De seguida, nas linhas 9, 10 e 11, é utilizado o método `typeInput()`, que é um método que vai escrever no campo passado no primeiro parâmetro o conteúdo que lhe for passado no segundo parâmetro. Por exemplo, na linha 9 este método vai escrever "4 Series" no campo de texto com o ID "`#model-input`", que corresponde ao campo de texto do modelo do carro. Na linha 12 é novamente utilizado o método `clickButton()` para pressionar no botão de "Save", tal como um utilizador real faria. Finalmente, o teste vai esperar que o componente `Cars` seja atualizado com o novo carro, vai contar os carros existentes e depois vai validar se o número de carros existentes tem mais um carro do que o número de carros iniciais. Para finalizar o teste é executado o método `afterAll()`, que apenas é executado depois de todos os testes terem completado. Este método vai informar o Puppeteer que pode fechar o navegador de internet.

A figura 5.7 é uma execução real dos testes funcionais desenvolvidos, sendo que todos foram completados com sucesso.

```
PASS src/components/App/App.FT.test.js
  App component functional tests
    √ should have more than zero cars initially (19 ms)
    √ should create a new car (293 ms)
    √ should edit a car (360 ms)
    √ should delete a car (52 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        4.885 s, estimated 5 s
Ran all test suites matching /FT.test.js/i.
```

FIGURA 5.7: Testes funcionais da aplicação *web*.

Desenvolvimento dos Ficheiros Kubernetes

Semelhante ao que se fez para a componente *back-end*, foi também criado um ficheiro de configuração Kubernetes para a aplicação *front-end*. O excerto 5.16 mostra essa configuração.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fullstack-archetype-frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: fullstack-archetype-frontend
  template:
    metadata:
      labels:
        app: fullstack-archetype-frontend
    spec:
      containers:
        - name: fullstack-archetype-frontend
          image: telmof/fullstack-archetype-frontend
          ports:
            - containerPort: 3000
---
apiVersion: v1
kind: Service
metadata:
  name: fullstack-archetype-frontend
spec:
  selector:
    app: fullstack-archetype-frontend
  ports:
    - port: 3000
      targetPort: 3000
```

EXCERTO 5.16: Configuração Kubernetes do *front-end*.

Como se pode ver pelo código, é uma configuração idêntica à configuração do *back-end*, sendo que as principais diferenças são os nomes dados aos componentes, a imagem Docker a ser utilizada pelo *container* do *pod*, e a porta onde esse *container* vai disponibilizar a aplicação.

Desenvolvimento do *Generator*

O *generator* da componente *front-end* desta solução foi desenvolvida de uma forma bastante semelhante à do *generator* da componente *back-end*, descrita na secção 5.1.1. Dessa forma, esta secção não irá entrar em tantos detalhes como a secção mencionada.

Tal como o *generator* do *back-end*, este *generator* também apenas utiliza dois métodos principais: o método `prompting()` e o método `writing()`. O excerto 5.17 mostra o conteúdo do método `prompting()`.

```
async prompting() {
    this.log(
        yosay("Welcome to the " + chalk.cyan("fullstack-archetype") + " frontend
    generator!")
    );

    this.answers = await this.prompt([
        {
            type: "input",
            name: "name",
            message: "What is your project name?",
            default: this.appname
        }
    ]);
}
```

EXCERTO 5.17: Método `prompting()` do *generator* do *front-end*.

Como se pode ver pelo excerto, este método é quase idêntico ao método do *generator* do *back-end*. Começa por cumprimentar o utilizador, e depois questiona ao mesmo o nome que este quer dar ao projeto *front-end*.

De seguida é executado o método `writing()`, cujo conteúdo é o código mostrado no excerto 5.18.

```
writing() {
    this.destinationRoot(this.answers.name);
    this.appname = this.answers.name;

    this.copyTemplate("_package.json", "package.json", {name: this.answers.
    name});
    this.copyTemplate("_package-lock.json", "package-lock.json", {name:
    this.answers.name});
    this.copyTemplate("_prettierrc.json", "prettierrc.json");
    this.copyTemplate("Dockerfile", "Dockerfile");
    this.copyTemplate("gitignore", ".gitignore");
    this.copyTemplate("prettierignore", ".prettierignore");
    this.copyTemplate("README.md", "README.md", {name: this.answers.name});
    this.copyTemplate("src", "src", {name: this.answers.name});
    this.copyTemplate("public", "public", {name: this.answers.name});
    this.copyTemplate("pipelines", "pipelines", {name: this.answers.name});
    this.copyTemplate("nginx", "nginx");
    this.copyTemplate("kubernetes", "kubernetes", {name: this.answers.name
    });
}
```

EXCERTO 5.18: Método `writing()` do *generator* do *front-end*.

Tal como o método `prompting()`, também este método `writing()` é bastante semelhante com o do *generator* do *back-end*. Este método começa por criar a pasta de destino com o nome definido pelo utilizador, sendo que depois vai gerar nessa pasta os ficheiros que estão na pasta `template/`. Em alguns ficheiros a variável `<%= name %>` vai ser substituída pelo nome do projeto definido pelo utilizador, tal como também acontece no *generator* do *back-end*.

5.1.3 Infraestrutura

Desenvolvimento dos *Scripts* Terraform

Como já descrito em secções anteriores deste documento, a solução desenvolvida deverá também criar toda a infraestrutura necessária na *cloud* para o alojamento das aplicações. Essa geração da infraestrutura, bem como a sua posterior manutenção e destruição, vai ser assegurada pelo Terraform.

Os *scripts* Terraform foram divididos por quatro ficheiros principais:

1. `vpc.tf`: Este ficheiro vai assegurar a criação da VPC que vai conter toda a infraestrutura na AWS, assim como uma *subnet* pública e uma *subnet* privada. Para este ficheiro foi utilizado um módulo Terraform, neste caso o módulo oficial da AWS para VPC. No Terraform, um módulo é um conjunto de recursos que são utilizados juntos ou que estão relacionados. Existem módulos já pré-desenvolvidos, tanto pela equipa oficial do Terraform como pela comunidade, no Terraform Registry, que funciona como um repositório de módulos Terraform assim como o Docker Hub funciona para imagens Docker.

2. `eks-cluster.tf`: Este ficheiro vai conter os recursos necessários para a criação do *cluster* EKS. Para a criação deste *cluster* foi utilizado o módulo da AWS para EKS, o que facilitou bastante o processo. Com a utlização do módulo apenas é necessário fornecer alguns dados, como por exemplo o nome e a versão do *cluster*, e o Terraform cria tudo o que é necessário automaticamente.

3. `ec2-jenkins.tf`: Este ficheiro, como indicado pelo nome, vai ser responsável por provisionar uma instância EC2 e por instalar o Jenkins nessa máquina virtual. Para isso, é utilizado um bloco `data` para encontrar a Amazon *Machine Image* (AMI) mais recente do tipo Amazon Linux 2. Depois, é utilizado um bloco `resource` do tipo "`aws_instance`" para criar uma instância EC2 com a AMI encontrada pelo bloco `data`, tal como mostra o excerto 5.19.

```
1  resource "aws_instance" "jenkins-instance" {
2    ami             = data.aws_ami.amazon_linux.id
3    instance_type   = "t2.micro"
4    key_name        = var.key_pair_name
5    security_groups = [aws_security_group.jenkins_sg.name]
6    user_data       = file("scripts/install_tools.sh")
7    tags = {
8      "Name" = var.jenkins_server_name
9    }
10 }
```

EXCERTO 5.19: Código Terraform para criar uma instância EC2.

Na linha 6 deste bloco é utilizado a propriedade `user_data` para executar um *script*. O objetivo deste *script* é instalar o Jenkins e todas as suas dependência, cujo conteúdo está representado no excerto 5.20.

```
#!/bin/bash

sudo yum -y update
sudo yum -y install curl
curl -sL https://rpm.nodesource.com/setup_14.x | sudo bash -
sudo wget -O /etc/yum.repos.d/jenkins.repo https://pkg.jenkins.io/
    redhat-stable/jenkins.repo
sudo rpm --import https://pkg.jenkins.io/redhat-stable/jenkins.io.key
sudo yum upgrade
sudo yum install jenkins java-1.8.0-openjdk-devel git nodejs gcc-c++
    make docker -y
sudo chkconfig docker on
sudo usermod -aG docker jenkins
sudo systemctl daemon-reload
sudo systemctl start jenkins
sudo systemctl status jenkins
sudo service docker start
```

EXCERTO 5.20: *Shell script* para instalar o Jenkins.

No entanto, como já foi discutido previamente, os *scripts* vão tratar da instalação do Jenkins automaticamente, mas a configuração das *pipelines* tem de ser feita manualmente pelo utilizador. Todavia, o utilizar pode facilmente utilizar as *pipelines* que são geradas nos repositórios do *back-end* e *front-end*.

4. `security-groups.tf`: Este ficheiro vai garantir a criação dos *security groups* utilizados pelo cluster EKS e pela instância EC2. Na AWS, os *security groups* contêm as regras de acesso aos diversos recursos. Por exemplo, este *script* cria um *security group* que abre a porta 22 da instância EC2, permitindo assim que seja possível aceder a esta máquina via SSH.

Para além destes quatro ficheiros principais, existem ainda dois ficheiros de apoio ao *scripts*:

1. `variables.tf`: Este ficheiro contém as variáveis que são usadas nos outros ficheiros. Este ficheiro é opcional, pois tal como em linguagens de programação tradicionais é perfeitamente possível colocar o valor das variáveis diretamente nos blocos Terraform. No entanto, definir as variáveis num ficheiro à parte ajuda com a organização e reutilização das variáveis.

2. `outputs.tf`: Este ficheiro vai ser responsável por imprimir na consola alguns valores de *output* no final da execução dos *scripts*. Um exemplo desses valores é o endereço IP da instância EC2 onde o Jenkins foi instalado, ou então valores do *cluster* EKS que foi criado. Os dados devolvidos no *output* de uma execução Terraform são valores que são úteis para o utilizador, ou então valores que serão usados por *scripts* que serão executados de seguida.

Para executar os *scripts*, primeiro é necessário executar o comando `terraform init` para inicializar o Terraform e todas as suas dependências, tais como quaisquer módulos utilizados. A figura 5.8 mostra a execução deste comando.

```
→  terraform git:(develop) x terraform init
Initializing modules...

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Reusing previous version of hashicorp/kubernetes from the dependency lock file
- Reusing previous version of terraform-aws-modules/http from the dependency lock file
- Reusing previous version of hashicorp/local from the dependency lock file
- Reusing previous version of hashicorp/cloudinit from the dependency lock file
- Using previously-installed terraform-aws-modules/http v2.4.1
- Using previously-installed hashicorp/local v2.1.0
- Using previously-installed hashicorp/cloudinit v2.2.0
- Using previously-installed hashicorp/aws v3.46.0
- Using previously-installed hashicorp/kubernetes v2.3.2

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

FIGURA 5.8: Execução do comando `terraform init`.

De seguida, o utilizador poderá executar o comando `terraform plan` para ver todas as alterações que a execução dos *scripts* irá fazer. Este comando é seguro de se executar pois o comando em si não vai fazer qualquer alteração, apenas vai listar as alterações que serão feitas. A figura 5.9 mostra o resultado parcial da execução deste comando. Este comando vai listar todos os recursos na AWS que serão criados/modificados/destruídos, pelo que não foi possível apanhar todo o seu *output* na imagem. O `terraform plan` também informa o utilizador do número de recursos que serão criados/modificados/destruídos, e para este projeto numa primeira execução dos *scripts* vão ser criados 48 recursos, como se pode ver pela imagem.

```
      + tags_all                          = {
          + "Name"                                    = "fullstack-archetype-vpc"
          + "kubernetes.io/cluster/fullstack-archetype" = "shared"
        }
    }

Plan: 48 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + cluster_endpoint          = (known after apply)
  + cluster_name              = "fullstack-archetype"
  + cluster_security_group_id = (known after apply)
  + config_map_aws_auth       = [
```

FIGURA 5.9: Execução do comando `terraform plan`.

Caso o utilizador queira avançar para a criação desses recursos, pode simplesmente executar o comando `terraform apply` e o Terraform irá tratar de provisionar todos esses componentes automaticamente.

O Terraform vai guardar o estado da infraestrutura criada por estes *scripts* num *bucket* S3, na AWS, num ficheiro do tipo `.tfstate`. E sempre que for executado um comando `terraform plan`,

`terraform apply` ou `terraform destroy` (para destruir a infraestrutura gerada), o Terraform vai comparar o resultado desse *plan* com o estado atual da infraestrutura no *bucket* S3. Caso existam diferenças entre o estado atual e o resultado do *plan*, apenas essas alterações serão realizadas.

A execução destes *scripts* Terraform vai ser assegurada pelo *script* principal, que será discutido em mais detalhe na secção 5.2.

Desenvolvimento dos Ficheiros Kubernetes

Embora esta componente de infraestrutura não precise de nenhum ficheiro Kubernetes, pois esta parte do projeto apenas vai gerar a infraestrutura necessária, acabou por conter um ficheiro de configuração para gerar um componente de Kubernetes denominado de *ingress*, que é algo que não pertence necessariamente ao *back-end* ou ao *front-end*. Um *ingress* é um componente Kubernetes semelhante ao *service*, na medida em que também vai agir como um *load balancer*. A diferença é que o *ingress* está uma camada acima dos *services*, e vai reencaminhar pedidos para os *services* definidos nas suas regras, enquanto que os *services* reencaminham pedidos para *deployments*.

O excerto 5.21 mostra a configuração do *ingress*, bem como as regras que foram definidas para o reencaminhamento de pedidos.

```
1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: fullstack-archetype-ingress
5    annotations:
6      kubernetes.io/ingress.class: "nginx"
7      nginx.ingress.kubernetes.io/use-regex: "true"
8      nginx.ingress.kubernetes.io/rewrite-target: /$1
9  spec:
10   rules:
11     - http:
12         paths:
13           - path: /?(.*)
14             pathType: Prefix
15             backend:
16               service:
17                 name: fullstack-archetype-frontend
18                 port:
19                   number: 3000
20           - path: /api/?(.*)
21             pathType: Prefix
22             backend:
23               service:
24                 name: fullstack-archetype-backend
25                 port:
26                   number: 4000
```

EXCERTO 5.21: Configuração Kubernetes do *ingress*.

As regras definidas, que começam na linha 10, significam o seguinte:

- A primeira regra, na linha 13, diz que qualquer pedido que seja feito à rota `/(...)` do endereço IP do *ingress* vai ser reencaminhado para o endereço IP do *service* da aplicação *front-end*, na porta 3000. Ou seja, em termos mais simples, se o utilizador aceder ao endereço IP do *ingress* vai obter a página *web* da aplicação *front-end*.

- A segunda regra, na linha 20, define que pedidos enviados para a rota `/api/(...)` do endereço IP do *ingress* vão ser reencaminhados para o serviço do *back-end*. A aplicação *front-end* vai enviar os seus pedidos para o *endpoint* `/api/cars`, sendo que o *ingress* vai encaminhar esses pedidos para o *back-end*, garantindo assim comunicação entre o *front-end* e o *back-end*.

Desenvolvimento do *Generator*

O *generator* para gerar o repositório da componente de infraestrutura manteve uma estrutura semelhante à aplicada nos *generators* do *back-end* e do *front-end*. Ou seja, foram também utilizados os métodos `prompting()` e `writing()`.

Começando com o método *prompting()*, representado pelo excerto 5.22, este apresenta uma ligeira diferença em relação aos métodos utilizados nos *generators* do *back-end* e do *front-end*.

```
1  async prompting() {
2      this.log(yosay("Welcome to the " + chalk.green("fullstack-archetype") +
        "infrastructure generator!"));
3
4      this.answers = await this.prompt([
5          {
6              type: "input",
7              name: "name",
8              message: "What is your project name?",
9              default: this.appname
10         },
11         {
12             type: "input",
13             name: "backend",
14             message: "What is the name of your backend repository?",
15         },
16         {
17             type: "input",
18             name: "frontend",
19             message: "What is the name of your frontend repository?",
20         }
21     ]);
22 }
```

EXCERTO 5.22: Método `prompting()` do *generator* de infraestrutura.

Como se pode ver, o método `prompting()` da componente de infraestrutura coloca mais algumas questões ao utilizador. Para além do nome do projeto de infraestrutura, o *generator* questiona também os nomes que o utilizador deu aos repositórios de *back-end* e *front-end*. A razão deste *generator* necessitar dessa informação é devido ao *ingress* do Kubernetes que vai ser gerado, uma vez que a configuração YAML do *ingress* tem os nomes dos *services* do *back-end* e do *front-end*.

O método `writing()`, representado pelo excerto 5.23, mostra onde estes valores obtidos no método `prompting()` vão ser aplicados, que é precisamente no ficheiro de configuração Kubernetes.

```
writing() {
    this.destinationRoot(this.answers.name);
    this.appname = this.answers.name;

    this.copyTemplate("README.md", "README.md");
    this.copyTemplate("gitignore", ".gitignore");
    this.copyTemplate("terraform", "terraform");
    this.copyTemplate("kubernetes", "kubernetes", {frontend: this.answers.
    frontend, backend: this.answers.backend});
}
```

EXCERTO 5.23: Método `writing()` do *generator* de infraestrutura.

Contudo, o *generator* da componente de infraestrutura apresenta ainda mais uma diferença em relação aos outros dois *generators*, na medida em que faz uso de um método adicional - o método `end()`. Relembrando as prioridades pré-definidas do Yeoman, o método `end()` é o último da lista, daí ter esse nome, uma vez que é o método que vai terminar as ações do *generator*. O excerto 5.24 mostra o conteúdo do método `end()`.

```
end() {
    const done = this.async();
    this.spawnCommand(`python terraform\\scripts\\run_terraform_scripts.py
    `).on('close', done);
}
```

EXCERTO 5.24: Método `end()` do *generator* de infraestrutura.

Este método vai utilizar o método `spawnCommand()`, que é um método da API do Yeoman para executar um comando. Este comando, como se pode ver pelo excerto, vai executar o *script* Python `run_terraform_scripts.py` depois do repositório da infraestrutura ter sido gerado. O excerto 5.25 representa o conteúdo do *script*.

```
import os
import sys

os.chdir("terraform")
os.system("terraform init")
os.system("terraform apply")
os.system("aws eks --region $(terraform output -raw region) update-
    kubeconfig --name $(terraform output -raw cluster_name)")
os.system("kubectl create clusterrolebinding cluster-system-anonymous --
    clusterrole=cluster-admin --user=system:anonymous")
os.system("kubectl apply -f https://raw.githubusercontent.com/kubernetes/
    ingress-nginx/controller-v0.48.1/deploy/static/provider/aws/deploy.yaml
    ")
os.chdir("..\\kubernetes\\templates")
os.system("kubectl apply -f ingress-service.yaml")
```

EXCERTO 5.25: Conteúdo do *script* `run_terraform_scripts.py`.

O *script* vai começar por entrar na pasta onde estão os ficheiros Terraform, sendo que depois vai executar o comando `terraform init` para inicializar os módulos Terraform e o comando `terraform apply` para mostrar ao utilizador tudo o que será gerado, sendo que depois este terá de confirmar para iniciar a execução dos *scripts*. Depois do Terraform ter terminado o aprovisionamento da infraestrutura, vai ser executado o comando da linha 7, que vai utilizar algumas variáveis do *output* do Terraform para criar uma ligação na máquina local do utilizador com o

cluster EKS que foi gerado na AWS. De seguida, vai ser executado o comando da linha 8 para criar as permissões necessárias no *cluster* para posteriormente ser possível fazer o *deploy* das aplicações via Jenkins. O comando da linha 9 vai instalar o NGINX Ingress Controller, que vai ser a aplicação que vai gerir as regras definidas no *ingress*. E, finalmente, o comando da linha 11 vai instalar o *ingress* previamente mencionado.

5.2 *Script* Principal

O *script* principal é o *script* mestre que vai gerir todo o processo da solução desenvolvida. Este é o único *script* com o qual o utilizador vai necessitar de interagir, uma vez que este *script* tem a capacidade de invocar todos os outros mencionados na secção anterior.

O repositório que contém o *script* principal tem a seguinte estrutura de ficheiros:

```
helpers.py
main-script.py
.gitignore
README.md
backend/
    __init__.py
    main.py
frontend/
    __init__.py
    main.py
infrastructure/
    __init__.py
    main.py
```

O ficheiro que o utilizador vai executar é o `main-script.py`. Para além deste ficheiro, o repositório conta ainda com o ficheiro `helpers.py`, que contém várias funções que são utilizadas ao longo do script, e três módulos Python, um para cada componente (*back-end*, *front-end* e infraestrutura).

O excerto 5.26 mostra o conteúdo do ficheiro `main-script.py`. A estruturação dos ficheiros em módulos resulta num ficheiro com uma complexidade bastante reduzida, tendo apenas 42 linhas de código. Dessa forma, mesmo sem precisar de analisar o código dos restantes ficheiros, uma equipa de desenvolvimento poderá facilmente perceber o código do *script* principal apenas através do fluxo da solução e dos nomes das funções.

```python
import backend.main
import frontend.main
import helpers
import infrastructure.main

yo_installed = False

helpers.print_project_title()

helpers.print_normal("This script will help you generate a fullstack project.")

# ---- Backend ----

backend_input = helpers.input_normal("\nDo you want to generate a backend
    repository? (Y/n) ")

if backend_input == "Y" or backend_input == "y":
    backend.main.generate_backend(yo_installed)
    yo_installed = True
else:
    helpers.print_normal("\nBackend will not be generated.")

# ---- Frontend ----

frontend_input = helpers.input_normal("\nDo you want to generate a frontend
    repository? (Y/n) ")

if frontend_input == "Y" or frontend_input == "y":
    frontend.main.generate_frontend(yo_installed)
    yo_installed = True
else:
    helpers.print_normal("\nFrontend will not be generated.")

# ---- Infrastructure ----

infrastructure_input = helpers.input_normal("\nDo you want to generate a cloud
    infrastructure? (Y/n) ")

if infrastructure_input == "Y" or infrastructure_input == "y":
    infrastructure.main.generate_infrastructure(yo_installed)
    yo_installed = True
else:
    helpers.print_normal("\nInfrastructure will not be generated.")

helpers.print_success("\nYou have successfully generated a fullstack project!")
```

EXCERTO 5.26: Ficheiro `main-script.py`.

De seguida vai ser feita uma descrição mais detalhada do funcionamento deste *script*. Os três módulos Python funcionam essencialmente da mesma maneira, e o fluxo é idêntico em todos os módulos, pelo que não há necessidade de se descrever os três módulos neste documento. Dessa forma, vai ser apenas descrito o funcionamento do módulo do *back-end*, sendo que essa descrição também se aplica aos restantes dois módulos.

O *script* começa por inicializar a variável `yo_installed` com o valor *False*. Esta variável vai ser utilizada posteriormente na geração dos repositórios via Yeoman, pelo que se vai detalhar mais esta variável nesse momento. De seguida o *script* vai mostrar o título da solução e uma breve descrição do que vai fazer. O fluxo que se segue já foi descrito em mais detalhe na secção 4.3, pelo que aqui vai ser mostrado o código que resulta no comportamento descrito.

A execução do módulo do *back-end* começa com uma pergunta que questiona ao utilizador se este quer gerar um repositório *back-end*, tal como mostra a figura 5.10.

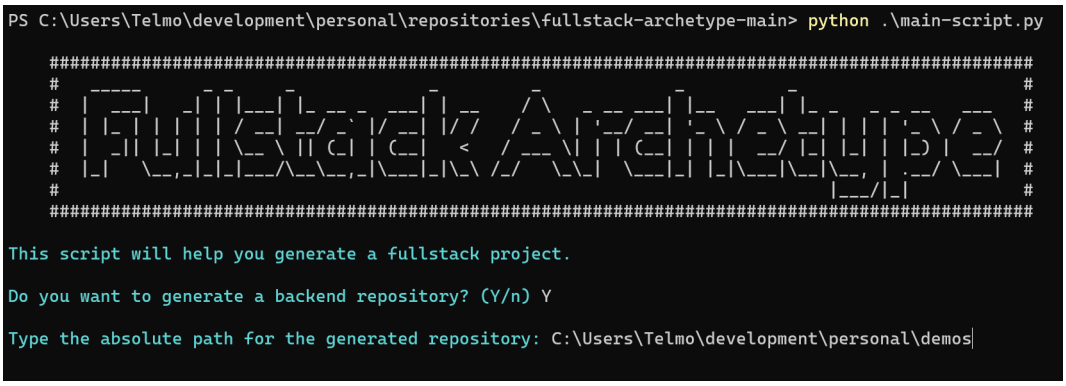

FIGURA 5.10: Execução do *script* principal.

Caso o utilizador não queira, o programa vai imprimir uma mensagem a informar que o repositório não será gerado. Caso responda de forma positiva, então vai ser invocada a função `generate_backend()` do módulo do *back-end*, passando-lhe a variável `yo_installed` como parâmetro. O excerto 5.27 mostra o conteúdo do ficheiro `main.py` do módulo do *back-end*.

```
import helpers


def generate_backend(yo_installed):
    repository_type = "backend"
    generator_name = f"fullstack-archetype-{repository_type}"
    repository_name = f"generator-{generator_name}"
    path = helpers.get_repository_path()

    helpers.generate(repository_name, repository_type, generator_name, path,
    yo_installed)
```

EXCERTO 5.27: Ficheiro `main.py` do módulo do *back-end*.

Como se pode ver, este ficheiro em si é também bastante simples. Apenas define algumas variáveis, questiona ao utilizador a pasta onde deverá ser gerado o repositório, através da função `get_repository_path()`, e depois invoca a função `generate()` do ficheiro `helpers.py`. A função `generate()` é uma função genérica que é utilizada por todos os módulos para a geração dos repositórios. Dessa forma, os ficheiros `main.py` dos módulos do *front-end* e infraestrutura são muito semelhantes a este, tendo apenas variáveis diferentes.

O excerto 5.28 mostra o código da função `generate()` do ficheiro `helpers.py`.

```
1  def generate(repository_name, repository_type, generator_name, path, yo_installed):
2      if yo_installed:
3          print_normal("\n'yo' has already been installed. Skipping this step...")
4      else:
5          install_yo()
6
7      os.chdir(path)
8
9      clone_repository(path, repository_name, repository_type)
10     npm_link()
11
12     os.chdir(path)
13
14     run_generator(generator_name, repository_type)
15     delete_repository(path, repository_name, repository_type)
16
17     print_success(f"\nDone! The {repository_type} repository was generated
       successfully.")
```

EXCERTO 5.28: Função `generate()` do ficheiro `helpers.py`.

Esta função começa por instalar o yo, dependendo do conteúdo da variável `yo_installed`. Recordando a secção 2.3.1, o Yeoman utiliza a ferramenta yo para fazer o *scaffolding* de aplicações. A variável `yo_installed` é inicializada como *False* no início do *script* principal, pelo que numa primeira execução o programa vai entrar na condição `else` e vai executar a função `install_yo()`, tal como mostra a figura 5.11.

```
Type the absolute path for the generated repository: C:\Users\Telmo\development\personal\demos

Installing 'yo' globally...
npm WARN deprecated har-validator@5.1.5: this library is no longer supported
npm WARN deprecated uuid@3.4.0: Please upgrade  to version 7 or higher.  Older versions may use Math.r
andom() in certain circumstances, which is known to be problematic.  See https://v8.dev/blog/math-rand
om for details.
npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/request/reques
t/issues/3142

changed 716 packages, and audited 717 packages in 14s

51 packages are looking for funding
  run `npm fund` for details

2 high severity vulnerabilities

To address all issues, run:
  npm audit fix

Run `npm audit` for details.
```

FIGURA 5.11: Execução da função `install_yo()`.

O excerto 5.29 mostra o código da função `install_yo()`.

```
1  def install_yo():
2      print_normal("\nInstalling 'yo' globally...")
3      os.system("npm install --global yo")
```

EXCERTO 5.29: Função `install_yo()` do ficheiro `helpers.py`.

No próximo passo da função `generate()`, na linha 7, o programa entra dentro da pasta definida na variável `path`, que foi preenchida no excerto 5.27 pela função `get_repository_path()` do

ficheiro `helpers.py`. O excerto 5.30 mostra o código desta função, que simplesmente pede ao utilizador para este digitar o caminho até à pasta onde este quer gerar o repositório *back-end*.

```
def get_repository_path():
    path_to_generate_repository = input_normal("\nType the absolute path for the
    generated repository: ")
    return path_to_generate_repository
```

EXCERTO 5.30: Função `get_repository_path()` do ficheiro `helpers.py`.

Continuando o fluxo do excerto 5.28, e já estando dentro da pasta onde vai ser gerado o repositório, o *script* vai executar a função `clone_repository()`, responsável por clonar o repositório do *generator* para a máquina local do utilizador. O excerto 5.31 mostra o código desta função. Neste caso, como esta solução se trata de um PoC, a função está a clonar um repositório pessoal. Contudo, no futuro, esta função irá clonar um repositório privado alojado na *cloud* da empresa. A função vai clonar o repositório do *generator*, cujo nome foi definido no módulo do *back-end*.

```
def clone_repository(path, repository_name, repository_type):
    print_normal(f"\nCloning the {repository_type} generator repository...")
    os.system(f"git clone https://TelmoF@bitbucket.org/TelmoF/{repository_name}.git
    ")
    os.chdir(f"{path}\\{repository_name}")
    os.system("git pull")
```

EXCERTO 5.31: Função `clone_repository()` do ficheiro `helpers.py`.

A figura 5.12 mostra esta função em funcionamento.

```
2 high severity vulnerabilities

To address all issues, run:
  npm audit fix

Run `npm audit` for details.

Cloning the backend generator repository...
Cloning into 'generator-fullstack-archetype-backend'...
Receiving objects: 100% (103/103), 80.52 KiB | 1.13 MiB/s, done.
Resolving deltas: 100% (34/34), done.
Already up to date.
```

FIGURA 5.12: Execução da função `clone_repository()`.

O próximo passo do fluxo é executar a função `npm_link()`, cujo código é o do excerto 5.32. Esta é uma função muito simples, que apenas vai executar os comandos `npm start` e `npm link`. Estes comandos vão instalar as dependências do *generator* nos módulos globais NPM do utilizador, permitindo assim que depois seja possível executar o comando `yo <nome-do-generator>` em qualquer pasta do computador.

```
def npm_link():
    print_normal("\nExecuting 'npm link'...")
    os.system("npm install")
    os.system("npm link")
```

EXCERTO 5.32: Função `npm_link()` do ficheiro `helpers.py`.

A figura 5.13 mostra a execução desta função.

```
Cloning the backend generator repository...
Cloning into 'generator-fullstack-archetype-backend'...
Receiving objects: 100% (103/103), 80.52 KiB | 1.13 MiB/s, done.
Resolving deltas: 100% (34/34), done.
Already up to date.

Executing 'npm link'...

added 131 packages, and audited 132 packages in 2s

12 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

up to date, audited 3 packages in 854ms

found 0 vulnerabilities
```

FIGURA 5.13: Execução da função `npm_link()`.

Depois, o programa vai navegar de volta para a pasta o repositório deve ser gerado e vai iniciar o *generator* com a função `run_generator()`. O excerto 5.33 mostra o código desta função, que vai simplesmente iniciar o *generator* passado por parâmetro, cujo nome foi definido no módulo do *back-end*. O que o utilizador verá depois de iniciar o *generator* é o funcionamento já descrito na secção 5.1.1.

```
1 def run_generator(generator_name, repository_type):
2     print_normal(f"\nStarting the {repository_type} generator...")
3     os.system(f"yo {generator_name}")
```

EXCERTO 5.33: Função `run_generator()` do ficheiro `helpers.py`.

A figura 5.14 mostra a execução do *generator* que vai gerar o repositório *back-end* com o nome definido pelo utilizador.

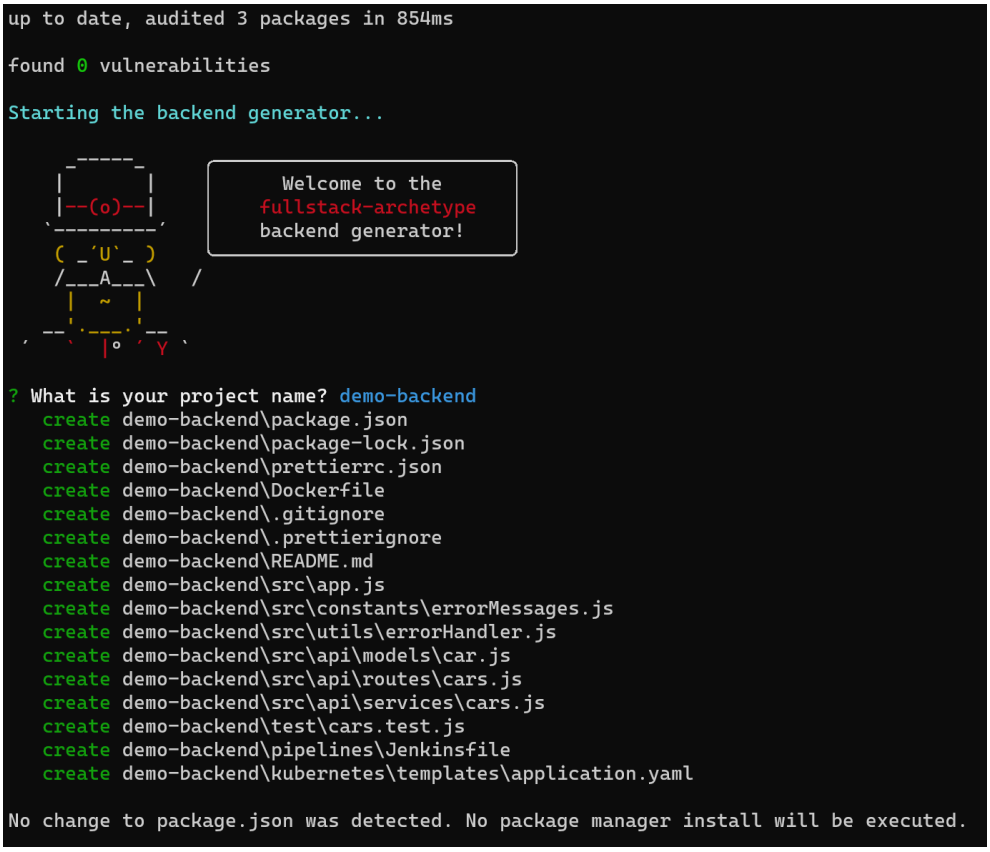

FIGURA 5.14: Execução da função `run_generator()`.

Depois da geração do repositório pela parte do *generator* ter terminado, o programa vai então invocar a função `delete_repository()` para eliminar o repositório do *generator* que clonou previamente, deixando assim o utilizador com apenas o repositório que foi gerado. O excerto 5.34 mostra o código desta função.

```
1 def delete_repository(path, repository_name, repository_type):
2     print_normal(f"\nDeleting the {repository_type} generator repository...")
3     os.system('rmdir /S /Q "{}"'.format(f"{path}\\{repository_name}"))
```

EXCERTO 5.34: Função `delete_repository()` do ficheiro `helpers.py`.

Finalmente, o programa vai executar a função `print_success()`, informando o utilizador de que o repositório *back-end* foi gerado com sucesso.

A figura 5.15 mostra a execução das funções `delete_repository()` e `print_success()`.

```
   create demo-backend\src\api\services\cars.js
   create demo-backend\test\cars.test.js
   create demo-backend\pipelines\Jenkinsfile
   create demo-backend\kubernetes\templates\application.yaml

No change to package.json was detected. No package manager install will be executed.

Deleting the backend generator repository...

Done! The backend repository was generated successfully.

Do you want to generate a frontend repository? (Y/n)
```

FIGURA 5.15: Finalização da geração do repositório *back-end*.

Como também se pode verificar na figura 5.15, depois de concluído o processo de geração do repositório *back-end*, é iniciado o módulo do *front-end*, repetindo-se o mesmo processo, mas desta vez para gerar o repositório *front-end*. Após este ter sido gerado, inicia-se um último ciclo para gerar o repositório da infraestrutura.

Concluída a geração dos repositórios de *back-end*, *front-end* e infraestrutura, o programa mostra ao utilizador uma mensagem de sucesso, informando-o de que gerou um projeto *full-stack*.

5.3 *Pipelines* de CI/CD e Configuração do Jenkins

Depois da finalização do *script* principal, o utilizador irá ter um repositório *back-end* com uma API REST, um repositório *front-end* com uma aplicação *web* em React e uma infraestrutura construída na *cloud* preparada para alojar ambas as aplicações e para permitir comunicação entre elas. Dessa forma, a única coisa que falta para cumprir todos os requisitos levantados são *pipelines* de CI/CD para permitir o *deploy* das aplicações na infraestrutura montada.

Os repositórios que foram gerados na máquina local do utilizador já têm as *pipelines* desenvolvidas. Por exemplo, analisando novamente a figura 5.14, consegue-se ver que foi gerada uma pasta chamada `pipelines` com um ficheiro `Jenkinsfile`, sendo este o nome que se dá a ficheiros de *pipelines* do Jenkins. Ambos os projetos *back-end* e *front-end* têm *pipelines* já desenvolvidas para o *deploy* das respetivas aplicações.

O excerto 5.35 mostra o código do `Jenkinsfile` do *back-end*.

```
pipeline {
    environment {
        DOCKERHUB_CREDENTIALS = credentials("dockerhub")
    }

    agent any

    stages {
        stage("Install Dependencies") {
            steps {
                sh script: "npm ci",
                label: "Install the project's dependencies in CI mode"
            }
        }
        stage("Tests") {
            steps {
                sh script: "npm run test",
                label: "Run the project's tests"
            }
        }
        stage("Docker Build") {
            steps {
                sh script: "docker build -t telmof/<%= name %>:latest .",
                label: "Build the Docker image"
            }
        }
        stage("Docker Login") {
            steps {
                sh script: "echo $DOCKERHUB_CREDENTIALS_PSW | docker login -u
    $DOCKERHUB_CREDENTIALS_USR --password-stdin",
                label: "Login into Docker Hub"
            }
        }
        stage("Docker Push") {
            steps {
                sh script: "docker push telmof/<%= name %>:latest",
                label: "Push the Docker image to Docker Hub"
            }
        }
        stage("Kubernetes Deploy") {
            steps {
                kubernetesDeploy(
                    configs: "kubernetes/templates/application.yaml",
                    kubeconfigId: "k8s",
                    enableConfigSubstitution: true
                )
            }
        }
    }

    post {
        always {
            sh script: "docker logout",
            label: "Logout from Docker Hub"
        }
    }
}
```

EXCERTO 5.35: `Jenkinsfile` do repositório *back-end*.

A *pipeline* do repositório de *front-end* é praticamente idêntica, tendo os mesmos *stages*, uma vez que também se trata de um projeto desenvolvido em JavaScript e construído com o NPM.

A *pipeline* vai efetuar as seguintes ações:

1. Instalar todas as dependências necessárias do projeto. Aqui foi usado o comando `npm ci` em vez do tradicional `npm install`, uma vez que o primeiro comando foi criado especialmente para *pipelines* de CI/CD, tendo uma utilização mais rápida.
2. Executar os testes desenvolvidos para a aplicação. Estes testes já foram descritos em mais detalhe nas secções respetivas da implementação do *back-end* e *front-end*.
3. Construir a imagem Docker da aplicação. De relembrar aqui que a sintaxe `<%= name %>` significa que este campo é substituído pelo nome do projeto definido pelo utilizador durante a geração do repositório.
4. Autenticar-se no Docker para ter acesso ao repositório que irá alojar a imagem criada no *stage* anterior. Para isso está a ser utilizada uma credencial que terá de ser configurada no Jenkins, que é a credencial obtida na linha 3. Para este exemplo, foi configurada uma credencial com o ID "dockerhub", constituída pelo nome de utilizador e respetiva *password* de uma conta pessoal do Docker.
5. Transferir a imagem gerada para o repositório no Docker Hub. Este passo só é possível uma vez que o Jenkins já vai estar autenticado no Docker.
6. Fazer o *deploy* da aplicação desenvolvida, através de um ficheiro Kubernetes que por sua vez vai utilizar a imagem que foi transferida para o Docker Hub. Para fazer este *deploy* está a ser utilizada a função `kubernetesDeploy()`, que não é uma função nativa do Jenkins. Para ser possível utilizar esta função, tem que ser primeiro instalado no Jenkins um *plugin* chamado "Kubernetes Continuous Deploy". Depois da instalação deste *plugin*, é também necessário configurar uma credencial de acesso ao *cluster* para onde se quer fazer o *deploy*. Este *plugin* adiciona um novo tipo de credencial ao Jenkins chamado "Kubeconfig", onde o utilizador poderá colar o conteúdo do ficheiro `%USERPROFILE%/.kube/config` da sua máquina local.
7. Finalmente, num bloco `post`, a *pipeline* vai fazer o terminar a sessão no Docker.

Após a execução desta *pipeline*, o utilizador terá feito *deploy* da API do *back-end* para o *cluster* EKS. Ao executar também a *pipeline* do projeto *front-end*, vai também fazer *deploy* da aplicação *web* do *front-end*. Graças ao *ingress* que já se encontra configurado no *cluster*, graças aos *scripts* Terraform, as componentes *back-end* e *front-end* vão poder comunicar, tendo assim o utilizador gerado um projeto verdadeiramente *full-stack*.

5.4 Sumário

Devido à magnitude do projeto desenvolvido, com todas as suas respetivas componentes, não foi possível abordar todas os desenvolvimentos e decisões que foram tomadas durante o processo de implementação, sendo apenas possível realçar alguns dos aspetos mais importantes e relevantes da implementação da solução desenvolvida. Dessa forma, este capítulo serviu para tentar demonstrar alguns dos princípios de engenharia informática que foram seguidos para a resolução do problema estabelecido no início do documento, bem como algumas das principais decisões que levaram ao *design* atual da solução.

Em suma, este capítulo descreveu uma solução capaz de cumprir todos os requisitos levantados para o problema em mãos, nomeadamente uma solução que fosse capaz de:

- Gerar um repositório para uma componente *back-end* e outro para uma componente *front-end*, através de *generators* Yeoman. Estes repositórios estão completos com todas as funcionalidades esperadas, assim como um conjunto de testes para testar todas essas funcionalidades.
- Estabelecer comunicação entre as componentes *back-end* e *front-end*, através de um *ingress* Kubernetes.
- Criar um ficheiro `Dockerfile`, tanto para o *back-end* como para o *front-end*, de forma a ser possível encapsular estas aplicações em *containers* Docker.
- Gerar ficheiros de configuração Kubernetes para criar *deployments* e *services*, de forma a utilizar as imagens Docker geradas para alojar as aplicações num *cluster* Kubernetes, com todas as vantagens que esta ferramenta proporciona.
- Gerar um repositório para uma componente de infraestrutura, novamente através de um *generator* Yeoman. Este repositório vai ter *scripts* Terraform para gerar toda a infraestrutura necessária para alojar esta solução na AWS.
- Gerar *pipelines* Jenkins para ser possível fazer *deploy* dos serviços. Estas *pipelines* vão construir as aplicações, executar todos os testes, construir as imagens Docker e fazer *deploy* das aplicações para o *cluster* Kubernetes.

É também importante realçar que o objetivo desta solução não era apenas criar uma ferramenta de geração de código, mas sim criar uma base sólida com bons princípios de engenharia de *software* para as equipas de desenvolvimento da Critical TechWorks, daí o nome de *archetype* no título deste documento. Dessa forma, a ferramenta de geração de código é importante, mas o código gerado é até ainda mais importante, pois é sobre este código que as equipas de desenvolvimento vão iterar e começar a desenvolver os seus projetos.

Capítulo 6

Experimentação e Avaliação

Este capítulo aborda a fase de experimentação e avaliação da solução. Ou seja, neste capítulo são discutidos os aspetos da solução que foram avaliados, e a forma de como foram avaliados.

O capítulo encontra-se dividido em três secções. Na primeira secção são definidas as hipóteses de investigação que foram avaliadas para determinar a qualidade da solução desenvolvida. A segunda secção descreve as metodologias que foram utilizadas para efetuar a avaliação da solução. A terceira e última secção explora os resultados obtidos no processo de avaliação.

6.1 Hipóteses de Investigação

Como mencionado anteriormente, o problema que esta solução visa resolver, ou pelo menos remediar, é o tempo perdido no início de um novo projeto de *software* devido a tarefas rotineiras que as equipas de desenvolvimento têm de efetuar manualmente. Dessa forma, este projeto tem como principal objetivo o desenvolvimento de uma solução que seja capaz de automatizar essas tarefas efetuadas no início de um novo projeto, poupando assim tempo às equipas de desenvolvimento e impedindo a ocorrência de erros humanos que possam ser cometidos durante essas tarefas. Para além disso, visa também gerar código que seja útil às equipas para começarem os seus desenvolvimentos, código que foi pensado para esse efeito e baseado em princípios sólidos da engenharia de *software*.

Para testar a solução, foram formuladas duas hipóteses que serão utilizadas para avaliar a qualidade da solução desenvolvida:

- **Hipótese de insucesso (H0):** A solução não cumpre com os requisitos e objetivos propostos, ou não fornece valor adequado às equipas de desenvolvimento.
- **Hipótese de sucesso (H1):** A solução cumpre com os requisitos e objetivos propostos, e fornece valor adequado às equipas de desenvolvimento.

A hipótese de insucesso será verificada caso aconteça algum destes casos:

- A solução não cumpre com todos os requisitos e objetivos propostos.
- As equipas de desenvolvimento consideram que a solução não proporciona valor adequado.
- As equipas de desenvolvimento consideram que a solução não oferece as funcionalidades necessárias para resolver o problema.

A hipótese de sucesso será verificada caso se verifiquem todos estes casos:

- A solução cumpre com todos os requisitos e objetivos propostos.

- As equipas de desenvolvimento consideram que a solução proporciona valor adequado.
- As equipas de desenvolvimento consideram que a solução oferece as funcionalidades necessárias para resolver o problema.

6.2 Metodologias de Avaliação

Para avaliar a solução desenvolvida, foram utilizadas duas métricas: **funcionalidade** e **usabilidade**.

6.2.1 Funcionalidade

Para testar a funcionalidade da solução desenvolvida foi feita uma análise aos requisitos funcionais que foram levantados no início juntamente com a empresa, e foi feita uma validação à solução desenvolvida para avaliar se estes requisitos foram implementados. Adicionalmente, foram utilizados testes automáticos para garantir a qualidade do código desenvolvido, sendo que a cobertura dessa código também vai ser utilizada para avaliar a funcionalidade da solução final.

No que toca às funcionalidades que podem ser testadas via testes automáticos, não existe propriamente um valor definido para uma boa cobertura de testes, uma vez que os testes necessários podem variar bastante consoante o tipo de projeto. Aliás, utilizar apenas o valor da cobertura de testes como métrica para a qualidade dos testes realizados pode por vezes ser um erro (Marick et al. 1999). Todavia, quanto maior for a cobertura de testes, menor tende a ser a probabilidade de existirem *bugs* não testados no código. De forma geral, uma cobertura de testes acima de 80% já produz resultados considerados aceitáveis (Malaiya et al. 1994).

6.2.2 Usabilidade

Para testar a usabilidade da solução, uma vez que esta é uma métrica mais subjetiva, foi utilizado um questionário que foi disponibilizado a alguns elementos das equipas da Critical TechWorks. No entanto, sendo esta uma métrica subjetiva, para ser possível validar a usabilidade geral da solução desenvolvida é importante existir uma metodologia que torne possível uma avaliação objetiva e sistemática da mesma. Assim, o método selecionado para avaliar a usabilidade da solução desenvolvida foi o *System Usability Scale* (SUS).

O SUS é um sistema que foi desenvolvido pelo John Brooke em 1986. De acordo com o autor, é um sistema rápido que permite que um utilizador consiga facilmente avaliar a usabilidade de um produto ou serviço (Brooke et al. 1996). As vantagens deste sistema incluem:

- Ser fácil de administrar aos participantes.
- Poder ser utilizado com uma amostra reduzida de participantes e com resultados fiáveis.
- Poder efetivamente diferenciar entre produtos/serviços usáveis e inutilizáveis.

O questionário é composto por dez afirmações, sendo que o participante tem de responder a essas afirmações com um valor de 1 a 5, sendo que estes valores têm o seguinte significado:

- **Valor 1:** Discordo totalmente.
- **Valor 2:** Discordo.
- **Valor 3:** Indiferente.
- **Valor 4:** Concordo.

- **Valor 5:** Concordo totalmente.

Relativamente ao questionário, as afirmações utilizadas pelo sistema SUS são padronizadas, podendo ser usadas para qualquer tipo de produto ou serviço. A tabela 6.1 mostra as afirmações utilizadas no questionário.

TABELA 6.1: Questionário sobre a usabilidade da solução.

ID	Afirmação
1	Eu gostaria de utilizar esta solução frequentemente.
2	Eu considerei a solução desnecessariamente complexa.
3	Eu considerei a solução fácil de utilizar.
4	Eu acho que iria precisar da ajuda de uma pessoa técnica para poder usar esta solução.
5	Eu considerei que as várias funções da solução estavam bem integradas.
6	Eu achei a solução demasiado inconsistente.
7	Eu acho que a maior parte das pessoas iriam aprender rapidamente a usar esta solução.
8	Eu considerei a solução incómoda de utilizar.
9	Eu estava muito confiante ao utilizar a solução.
10	Eu precisei de aprender muitos conceitos antes de poder utilizar a solução.

Estas afirmações têm uma particularidade que é importante na determinação da usabilidade, que é o facto de que as afirmações com números ímpares terem uma conotação positiva, enquanto que as com números pares têm uma conotação negativa.

Estes são os passos para obter a classificação final do SUS:

1. Para afirmações ímpares: subtrair 1 à resposta do participante.
2. Para afirmações pares: subtrair a resposta do participante a 5.
3. Isso vai normalizar todos os valores para um intervalo entre 0 e 4, sendo 4 a resposta mais positiva.
4. Somar os valores normalizados de todas as afirmações (que vai dar um resultado numa escala de 0 a 40), e depois multiplicar esse número por 2,5 (tendo assim um resultado numa escala de 0 a 100).

Segundo Jeff Sauro, a média das pontuações SUS de 500 estudos é 68. Ou seja, uma pontuação superior a 68 seria considerada acima da média, enquanto que uma pontuação inferior a 68 seria considerada abaixo da média (Sauro 2011). Sauro também formulou uma escala com vários níveis de pontuações SUS, sendo que a escala vai de F (a classificação mais baixa) até A (a classificação mais alta). A tabela 6.2 mostra as classificações para vários intervalos de pontuações SUS.

TABELA 6.2: Classificações das pontuações SUS.

Classificação	Pontuação SUS
A	> 80,3
B	68 - 80,3
C	68
D	51 - 68
F	< 51

6.3 Resultados

Nesta secção são discutidos os resultados que foram obtidos para as métricas levantadas anteriormente, utilizando para isso as metodologias já detalhadas.

6.3.1 Funcionalidade

Em termos dos testes desenvolvidos para a componente *front-end*, foi utilizada a biblioteca Jest. Esta biblioteca já tem nativamente a capacidade de avaliar a cobertura de testes, tendo sido por isso uma métrica fácil de obter. A figura 6.1 mostra o resultado obtido, revelando uma cobertura de testes de 100%.

```
-------------|---------|----------|---------|---------|-------------------
File         | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------|---------|----------|---------|---------|-------------------
All files    |     100 |      100 |     100 |     100 |
 App         |     100 |      100 |     100 |     100 |
  App.js     |     100 |      100 |     100 |     100 |
 Button      |     100 |      100 |     100 |     100 |
  Button.js  |     100 |      100 |     100 |     100 |
 Car         |     100 |      100 |     100 |     100 |
  Car.js     |     100 |      100 |     100 |     100 |
 CarForm     |     100 |      100 |     100 |     100 |
  CarForm.js |     100 |      100 |     100 |     100 |
 Cars        |     100 |      100 |     100 |     100 |
  Cars.js    |     100 |      100 |     100 |     100 |
 Header      |     100 |      100 |     100 |     100 |
  Header.js  |     100 |      100 |     100 |     100 |
-------------|---------|----------|---------|---------|-------------------

Test Suites: 6 passed, 6 total
Tests:       21 passed, 21 total
Snapshots:   0 total
Time:        5.644 s
```

FIGURA 6.1: Cobertura de testes da componente *front-end*.

Este é o melhor valor que se pode obter para esta métrica, pois significa que todo o código existente no projeto está coberto por testes.

Em relação aos testes desenvolvidos para a componente *back-end*, infelizmente não foi possível obter esta métrica. A razão para isso foi devido à biblioteca de testes utilizada, que foi a biblioteca

Mocha. Esta biblioteca, ao contrário do Jest, não tem a habilidade nativa de avaliar a cobertura de testes. Segundo a comunidade *online*, a melhor forma atualmente de obter a cobertura de testes com o Mocha é com a ferramenta Istanbul, que é uma ferramenta de cobertura de testes para JavaScript e que pode ser utilizada com algumas bibliotecas de testes. Com o Mocha, a melhor forma de utilizar o Istanbul é através da ferramenta nyc, que é uma ferramenta que proporciona uma interface em linha de comandos para o Istanbul. No entanto, após instalar esta ferramenta e executar os testes com o modo de cobertura ativo, a ferramenta diz que a cobertura de testes da componente *back-end* está em 0%, tal como mostra a figura 6.2. Este resultado está claramente errado, uma vez que esta componente tem testes para todos os *endpoints* da API. Uma possível solução para este problema seria uma migração dos testes desenvolvidos para Jest. Contudo, por questões de tempo, esta medida não foi possível de ser implementada a tempo da submissão desta dissertação.

```
  Cars API
    GET /cars
      √ should get all the cars
    POST /cars
      √ should create a new car with a valid body
      √ should not create a new car with an invalid body
    GET /cars/:id
      √ should get a car if the ID exists
      √ should not get a car if the ID doesn't exist
    PUT /cars/:id
      √ should update a car with a valid ID and valid body
      √ should not update a car with a valid ID and invalid body
      √ should not update a car with an invalid ID
    DELETE /cars/:id
      √ should delete a car if the ID exists
      √ should not delete a car if the ID doesn't exist


  10 passing (78ms)

----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |       0 |        0 |       0 |       0 |
----------|---------|----------|---------|---------|-------------------
```

FIGURA 6.2: Cobertura de testes da componente *back-end*.

Finalmente, para além da cobertura de testes, foi também realizada uma auto-validação de todos os requisitos funcionais levantados na secção 3.1 deste documento, para garantir que foram implementadas todas as funcionalidades necessárias. Esta validação foi assegurada pelos testes automáticos desenvolvidos para o código gerado, assim como por testes manuais realizados aos *scripts* e aos seus resultados. A tabela 6.3 mostra o resultado dessa validação.

TABELA 6.3: Validação dos requisitos funcionais.

Requisito Funcional	**Implementado?**
Criar as componentes *back-end* e *front-end*	Sim
Estabelecer comunicação entre as componentes *back-end* e *front-end*	Sim
Criar *templates* para testes	Sim
Utilizar tecnologias de *containerization* das aplicações	Sim
Utilizar ferramentas de orquestração de *containers*	Sim
Utilizar ferramentas de gestão da infraestrutura	Sim
Utilizar mecanismos de CI/CD	Sim

Como se pode ver, todos os requisitos funcionais foram devidamente implementados na solução final. Para além disso, os testes desenvolvidos oferecem uma cobertura suficiente para validar a qualidade do código gerado pela solução. Dessa forma, pode-se considerar que a solução final obtém assim uma boa classificação na métrica de funcionalidade.

6.3.2 Usabilidade

Para a métrica da usabilidade, como já indicado, foi utilizado o sistema SUS. No total, foram respondidos dez questionários por alguns elementos das equipas da Critical TechWorks sobre a usabilidade da solução desenvolvida. Tal como já mencionado, o sistema SUS produz resultados válidos mesmo com um número reduzido de participantes. Dessa forma, o número de participantes envolvidos é suficiente para julgar a usabilidade da solução.

As figuras 6.3, 6.4, 6.5, 6.6 e 6.7 mostram os resultados dos questionários.

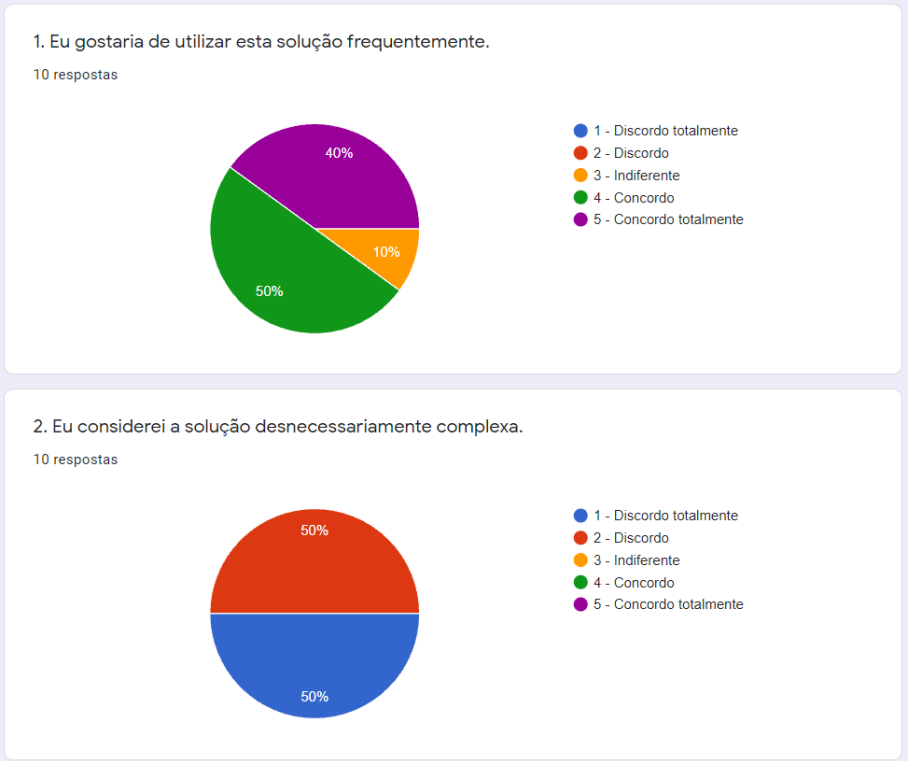

FIGURA 6.3: Respostas das afirmações 1 e 2.

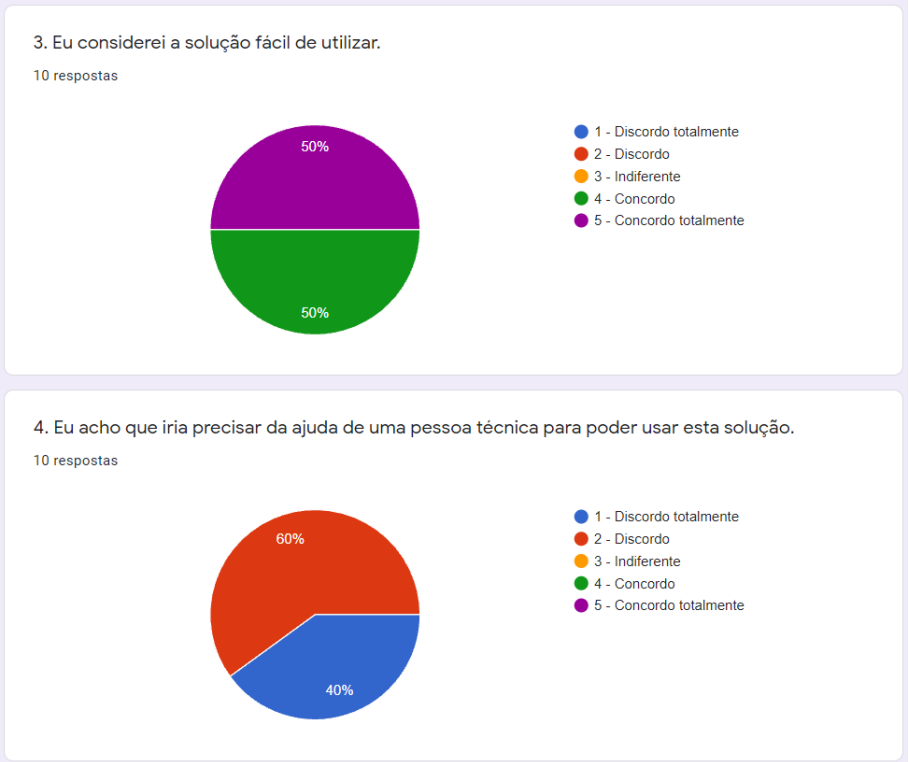

FIGURA 6.4: Respostas das afirmações 3 e 4.

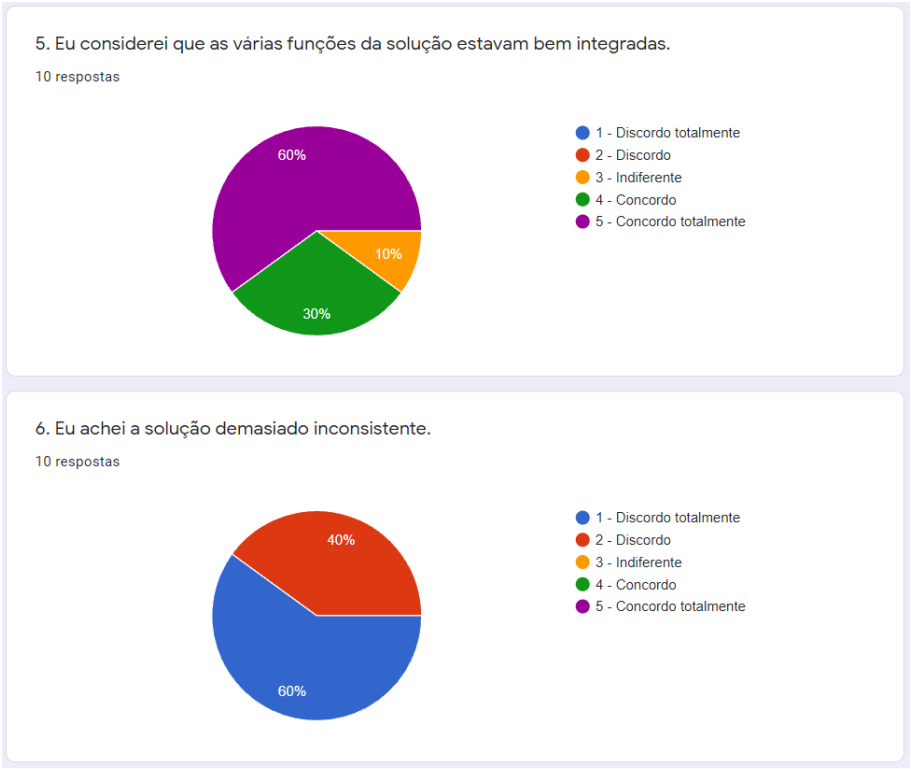

FIGURA 6.5: Respostas das afirmações 5 e 6.

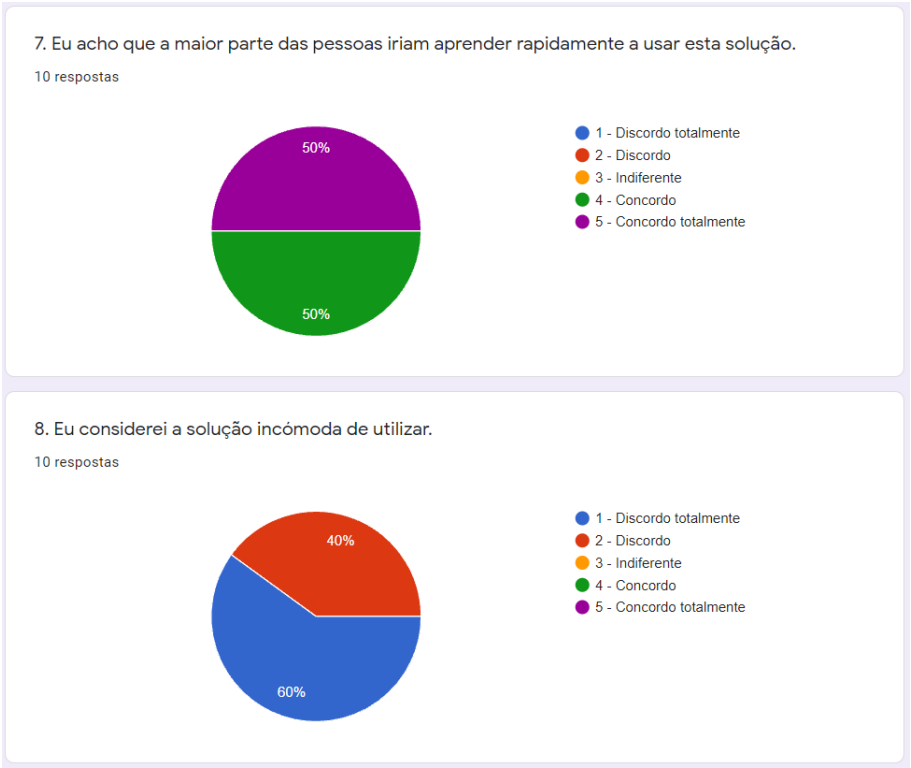

FIGURA 6.6: Respostas das afirmações 7 e 8.

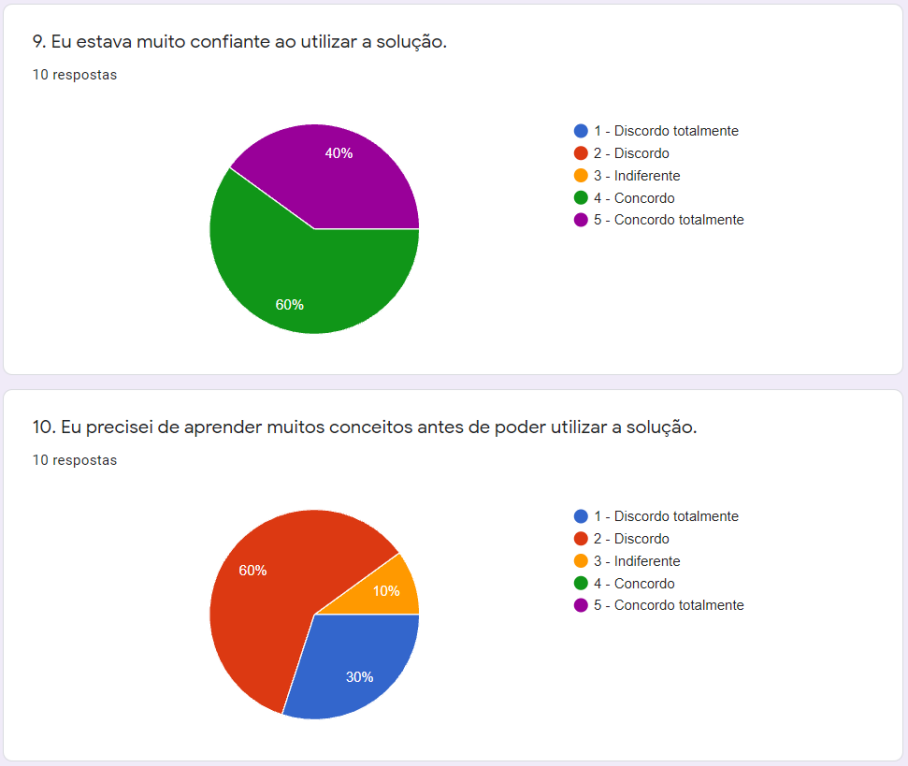

FIGURA 6.7: Respostas das afirmações 9 e 10.

A tabela 6.4 mostra um resumo das repostas de todos os participantes, para cada uma das afirmações do questionário.

TABELA 6.4: Respostas do questionário.

	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10
Participante 1	5	1	5	1	5	1	5	1	5	1
Participante 2	4	2	5	2	5	2	5	2	5	3
Participante 3	4	2	4	2	5	2	5	1	4	2
Participante 4	4	1	4	2	5	1	5	1	5	2
Participante 5	4	2	5	2	4	1	4	1	4	2
Participante 6	5	1	5	1	5	1	5	1	5	1
Participante 7	3	2	4	2	4	1	4	2	4	2
Participante 8	4	2	4	2	4	2	4	2	4	2
Participante 9	5	1	4	1	5	2	4	1	4	2
Participante 10	5	1	5	1	3	1	4	2	4	1

A tabela 6.5 mostra a pontuação SUS de cada participante. Relembra-se que para se chegar a este valor é necessário passar pelos passos que foram detalhados na secção 6.2.2.

TABELA 6.5: Resultados do questionário.

	Pontuação SUS
Participante 1	100
Participante 2	82,5
Participante 3	82,5
Participante 4	90
Participante 5	82,5
Participante 6	100
Participante 7	75
Participante 8	75
Participante 9	87,5
Participante 10	87,5

Calculando a média de todas as pontuações SUS, obtém-se um valor final de **86,25**. Voltando à tabela 6.2, e como a pontuação SUS da solução desenvolvida é superior a 80,3, esta solução obtém assim a classificação máxima (classificação A) em termos de usabilidade.

6.4 Sumário

Este capítulo realizou uma avaliação à solução final desenvolvida. Para esta avaliação foram utilizadas métricas e metodologias objetivas, obtendo assim resultados fiáveis e significativos.

Para a métrica de funcionalidade usou-se os valores de cobertura de testes e uma validação da implementação dos requisitos funcionais. Para a métrica da usabilidade utilizou-se um questionário e o sistema SUS para obter uma pontuação objetiva para as respostas desse questionário.

A ferramenta desenvolvida obteve uma boa classificação em ambas as métricas, podendo dessa forma concluir-se que se trata de uma boa solução para o problema em questão, e podendo confirmar-se a validação da **hipótese de sucesso (H1)**.

Capítulo 7

Conclusão

Este é o capítulo final do documento. Neste capítulo é feita uma conclusão do tema principal desta dissertação.

O capítulo está divido em três secções. Na primeira secção é feito um resumo de todo o trabalho realizado durante este projeto. A segunda secção aborda possíveis trabalhos futuros e melhoramentos que podem ser implementados. Na terceira e última secção é realizado uma apreciação final deste projeto e desta dissertação.

7.1 Resultados Alcançados

O problema levantado no início deste projeto era o tempo perdido pelas equipas de desenvolvimento na fase inicial de um novo projeto, onde era necessário criar um novo produto ou serviço de raiz. A razão principal para esse tempo perdido era devido a esta fase ser maioritariamente manual. Dessa forma, o propósito deste projeto era a criação de uma solução que resolvesse este problema, tendo dois objetivos principais:

1. Primeiro, seria necessário uma ferramenta que automatizasse o maior número possível de tarefas, para que estas tarefas não tenham de ser feitas pelos membros das equipas de desenvolvimento.
2. Depois, esta ferramenta teria de gerar código útil para as equipas. Isto é, código que as equipas pudessem aproveitar como um modelo para os seus próprios desenvolvimentos, começando assim os projetos já com uma base desenvolvida.

Como resposta a esse problema, foi desenvolvida uma solução composta por um *script* principal que iria automatizar uma série de tarefas, sendo que cada uma dessas tarefas teria o seu próprio *script* de automação. Dessa forma, o utilizador apenas teria de invocar o *script* principal, sendo que depois este teria a responsabilidade de invocar todos os restantes *scripts*. Para além disso, este método também oferece a vantagem de oferecer alguma flexibilidade ao utilizador, dando-lhe assim a liberdade de selecionar que componentes precisa para o seu projeto. Ou seja, caso uma equipa tenha já um *back-end* implementado e necessite apenas de um *front-end* e de infraestrutura para esse *front-end*, a ferramenta permite essa flexibilidade.

A solução final foi desenvolvida completamente de raiz, oferecendo assim controlo total às equipas de desenvolvimento da Critical TechWorks de alterarem a solução como entenderem. Se uma equipa precisar que o código gerado seja ligeiramente diferente, então essa equipa pode facilmente ir ao repositório do respetivo *generator* e fazer as alterações que necessitar. Depois, quando executar o *script*, a solução vai gerar o código de acordo com as suas especificidades.

Em termos de funcionalidades, a solução desenvolvida tem as seguintes funcionalidades principais:

- Gerar um repositório *back-end*, composto por uma API REST desenvolvida em Node.js. Esta API vai ter um *endpoint* `/cars`, que é simplesmente um *endpoint* para as equipas de desenvolvimento utilizarem como exemplo. Este *endpoint* vai suportar todas as operações CRUD, permitindo operações `GET`, `POST`, `PUT` e `DELETE`.
- Gerar um repositório *front-end*, composto por uma aplicação *web* desenvolvida em React. Esta aplicação é uma SPA que vai comunicar com a API do *back-end*. A aplicação vai ter funcionalidades de obter, criar, editar e apagar dados, permitindo desta forma utilizar todos os *endpoints* da API.
- Os repositórios *back-end* e *front-end* vão ser gerados com testes. O *back-end* vai ter testes unitários desenvolvidos utilizando Mocha e Chai. O *front-end* vai ter testes unitários desenvolvidos utilizando Jest e testes E2E desenvolvidos utilizando Puppeteer.
- Os repositórios *back-end* e *front-end* vão ser gerados com ficheiros `Dockerfile`, permitindo assim que ambas estas componentes possam ser executadas em *containers* Docker.
- Ambos os repositórios vão ser gerados com ficheiros Kubernetes, permitindo dessa forma que ambas as componentes possam ser lançadas num *cluster* de Kubernetes.
- Ambos os repositórios vão ser gerados com *pipelines* do Jenkins. Essas *pipelines* vão automatizar o processo de CI/CD, fazendo com que o *software* seja passado por todas as fases necessárias antes de ser feito o *deploy*.
- Finalmente, a solução vai também gerar um repositório de infraestrutura. Este repositório vai conter ficheiros Terraform para gerar uma infraestrutura na AWS que permita alojar todas as componentes geradas pela solução.

7.2 Trabalhos Futuros

A solução desenvolvida cumpre com todos os requisitos funcionais e não-funcionais que foram levantados no início do projeto. Todavia, existe sempre espaço para melhorar. Dessa forma, foram identificados alguns pontos que poderiam ser melhorados em trabalhos futuros:

- **Ter uma *Graphical User Interface* (GUI)**

  A utilização da solução desenvolvida é feita inteiramente em linha de comandos. Como esta é uma solução feita para programadores, o facto de a ferramenta ser usada via linha de comandos não é necessariamente um problema, uma vez que esta é uma forma habitual para os programadores interagirem com *scripts*. No entanto, se a ferramenta tivesse uma GUI bem desenhada isso iria melhorar a experiência de utilização, tornando-a visualmente mais apelativa.

- **Permitir mais escolha de tecnologias**

  A solução desenvolvida vai gerar repositórios com tecnologias já pré-definidas, neste caso Node.js para o *back-end*, React para o *front-end* e AWS como a *cloud* utilizada. No entanto, poderia-se melhorar substancialmente a solução se esta permitisse escolher a tecnologia do repositório gerado. Por exemplo, poder escolher entre React, Vue ou Angular para o *front-end*, e Node.js, Java ou Python para o *back-end*.

Este ponto foi algo que foi pensado no início do desenvolvimento como um bónus que seria feito caso houvesse tempo para tal. No entanto, o tempo de desenvolvimento acabou por demorar um pouco mais do que antecipado, pelo que não foi possível implementar esta funcionalidade dentro do prazo estipulado.

7.3 Apreciação Final

A realização de todo este projeto, incluindo o desenvolvimento da ferramenta em si, como a elaboração da presente dissertação, foi sem dúvida uma experiência desafiante e enriquecedora. Primeiro, foi necessário melhorar alguns conhecimentos de escrita científica para a elaboração desta dissertação, e aprender métodos inteiramente novos, tais como os métodos utilizados na secção de Análise de Valor. Depois, para o desenvolvimento da solução final, foi necessário pôr em prática muitos dos conhecimentos aprendidos durante o currículo letivo do Mestrado em Engenharia Informática, assim como os conhecimentos adquiridos na Critical TechWorks.

Como resultado deste projeto, foi desenvolvida uma ferramenta que será útil não só para a empresa, como também será útil para acelerar o desenvolvimento de futuros projetos pessoais. Como tal, foi um projeto bastante positivo, tanto em termos de melhoramento de capacidades de engenharia de *software*, como também em termos de rigor de escrita e redação de documentos científicos.

Bibliografia

Arachchi, SAIBS e Indika Perera (2018). «Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management». Em: *2018 Moratuwa Engineering Research Conference (MERCon)*. IEEE, pp. 156–161.

Artac, Matej et al. (2017). «Devops: introducing infrastructure-as-code». Em: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, pp. 497–498.

BMW (2020). *Critical TechWorks*. `https://www.bmw.pt/pt/topics/fascination-bmw/critical-techworks.html`. Acedido em: 2020-12-02.

Bojinov, Valentin (2015). *RESTful Web API Design with Node.js*. Packt Publishing Ltd.

Borges, Luiz Eduardo (2014). *Python para desenvolvedores: aborda Python 3.3*. Novatec Editora.

Brihadiswaran, Gunavaran (2020). *A Performance Comparison Between C, Java, and Python*. `https://medium.com/swlh/a-performance-comparison-between-c-java-and-python-df3890545f6d`. Acedido em: 2021-04-16.

Brikman, Yevgeniy (2016). *Why we use Terraform and not Chef, Puppet, Ansible, Saltstack, or cloudformation*.

Brooke, John et al. (1996). «SUS - A Quick and Dirty Usability Scale». Em: *Usability evaluation in industry* 189.194, pp. 4–7.

Chanana, Banjot (2017). *Extending Docker Enterprise Edition to Support Kubernetes*. `https://www.docker.com/blog/docker-enterprise-edition-kubernetes/`. Acedido em: 2021-01-30.

Chung, Lawrence e Julio Cesar Sampaio do Prado Leite (2009). «On non-functional requirements in software engineering». Em: *Conceptual modeling: Foundations and applications*. Springer, pp. 363–379.

CircleCI (2020). *CI/CD Overview*. `https://circleci.com/docs/2.0/about-circleci/#section=welcome`. Acedido em: 2021-01-30.

Collier, Michael e Robin Shahan (2015). *Microsoft Azure Essentials-Fundamentals of Azure*. Microsoft Press.

Cortiñas, Alejandro et al. (2017). «Scaffolding and in-browser generation of web-based GIS applications in a SPL tool». Em: *Proceedings of the 21st International Systems and Software Product Line Conference-Volume B*, pp. 46–49.

Daityari, Shaumik (2020). *Angular vs React vs Vue: Which Framework to Choose in 2021*. `https://www.codeinwp.com/blog/angular-vs-vue-vs-react/`. Acedido em: 2021-02-02.

Dauzon, Samuel, Aidas Bendoraitis e Arun Ravindran (2016). *Django: Web Development with Python*. Packt Publishing Ltd.

Díaz Alonso, Rubén Cayetano (2017). «Software containerization with Docker». Tese de doutoramento. Turku University of Applied Sciences.

Doerrfeld, Bill (2019). *5 Container Alternatives to Docker*. `https://containerjournal.com/topics/container-ecosystems/5-container-alternatives-to-docker/`. Acedido em: 2021-01-30.

Dutta, Pranay e Prashant Dutta (2019). «Comparative Study of Cloud Services Offered by Amazon, Microsoft & Google». Em: *International Journal of Trend in Scientific Research and Development* 3.3, pp. 981–985.

Ebert, Christof et al. (2016). «DevOps». Em: *Ieee Software* 33.3, pp. 94–100.

Eldridge, Isaac (2018). *What Is Container Orchestration?* `https://blog.newrelic.com/engineering/container-orchestration-explained/`. Acedido em: 2021-01-30.

Flanagan, David e Pasi Matilainen (2007). *JavaScript: O guia definitivo*. Anaya Multimedia.

Forman, Ernest H e Saul I Gass (2001). «The Analytic Hierarchy Process — An Exposition». Em: *Operations research* 49.4, pp. 469–486.

Foundation, OpenJS (2020). *About Node.js*. `https://nodejs.org/en/about/`. Acedido em: 2020-12-27.

Gleason, Ryan (2020). *Node.js vs. Spring Boot — Which Should You Choose?* `https://medium.com/better-programming/node-js-vs-spring-boot-which-should-you-choose-2366c2f76587`. Acedido em: 2020-12-27.

Gosling, James, David Colin Holmes e Ken Arnold (2005). *The Java programming language*.

Grinberg, Miguel (2018). *Flask web development: developing web applications with python*. "O'Reilly Media, Inc."

Hahn, Evan (2016). *Express in Action: Writing, building, and testing Node. js applications*. Manning Publications,

Hat, Red (2020). *What is container orchestration?* `https://https://www.redhat.com/en/topics/containers/what-is-container-orchestration`. Acedido em: 2020-12-23.

Hochstein, Lorin e Rene Moser (2017). *Ansible: Up and Running: Automating configuration management and deployment the easy way*. "O'Reilly Media, Inc."

Hoda, Rashina, Norsaremah Salleh e John Grundy (2018). «The rise and evolution of agile software development». Em: *IEEE software* 35.5, pp. 58–63.

Inayatullah, Mohammad, Farooque Azam e Muhammad Waseem Anwar (2019). «Model-Based Scaffolding Code Generation for Cross-Platform Applications». Em: *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. IEEE, pp. 1006–1012.

Jun, Sunghae e Sangsung Park (2016). «Examining technological competition between BMW and Hyundai in the Korean car market». Em: *Technology Analysis & Strategic Management* 28.2, pp. 156–175.

Kakadia, Dharmesh (2015). *Apache Mesos Essentials*. Packt Publishing Ltd.

Kisller, Edward (2020). *7 Alternatives to Docker: All-in-One Solutions and Standalone Container Tools*. `https://jfrog.com/knowledge-base/7-alternatives-to-docker-all-in-one-solutions-and-standalone-container-tools/`. Acedido em: 2021-01-30.

Koen, Peter et al. (2001). «Providing Clarity and a Common Language to the "Fuzzy Front End"». Em: *Research-Technology Management* 44.2, pp. 46–55.

Lei, Kai, Yining Ma e Zhi Tan (2014). «Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js». Em: *2014 IEEE 17th international conference on computational science and engineering*. IEEE, pp. 661–668.

Lightsey, Bob (2001). *Systems engineering fundamentals*. Rel. téc. DEFENSE ACQUISITION UNIV FT BELVOIR VA.

Luo, Lu (2001). «Software testing techniques». Em: *Institute for software research international Carnegie mellon university Pittsburgh, PA* 15232.1-19, p. 19.

Mahalingam, Manoj (2014). *Learning Continuous Integration with TeamCity*. Packt Publishing Ltd.

Malaiya, Yashwant K et al. (1994). «The Relationship Between Test Coverage and Reliability». Em: *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*. IEEE, pp. 186–195.

Malhotra, Lakshay, Devyani Agarwal, Arunima Jaiswal et al. (2014). «Virtualization in cloud computing». Em: *J. Inform. Tech. Softw. Eng* 4.2, p. 136.

Mardan, Azat (2014). *Express. js Guide: The Comprehensive Book on Express. js*. Azat Mardan.

Marick, Brian et al. (1999). «How to Misuse Code Coverage». Em: *Proceedings of the 16th Interational Conference on Testing Computer Software*, pp. 16–18.

Marschall, Matthias (2015). *Chef infrastructure automation cookbook*. Packt Publishing Ltd.

Masse, Mark (2011). *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. "O'Reilly Media, Inc."

Miles, Lawrence D (1989). *Techniques of Value Analysis and Engineering*. Miles Value Foundation.

Mouat, Adrian (2015). *Using Docker: Developing and Deploying Software with Containers*. "O'Reilly Media, Inc."

Muhtaroğlu, F Canari Pembe et al. (2013). «Business Model Canvas Perspective on Big Data Applications». Em: *2013 IEEE International Conference on Big Data*. IEEE, pp. 32–37.

Musciano, Chuck e Bill Kennedy (2002). *HTML & XHTML: The Definitive Guide: The Definitive Guide*. "O'Reilly Media, Inc."

Nayak, Ramnath (2019). *When to use which Infrastructure-as-code tool*. `https://medium.com/cloudnativeinfra/when-to-use-which-infrastructure-as-code-tool-665af289fbde`. Acedido em: 2021-01-30.

Okuneu, Barys (2020). *Java vs Node.js*. `https://belitsoft.com/java-development-services/java-vs-nodejs`. Acedido em: 2021-04-16.

Oracle (2020). *Helidon's FAQ*. `https://github.com/oracle/helidon/wiki/FAQ`. Acedido em: 2020-12-27.

Pedregosa, Fabian et al. (2011). «Scikit-learn: Machine learning in Python». Em: *the Journal of machine Learning research* 12, pp. 2825–2830.

Pereira, Caio Ribeiro (2014). *Aplicações web real-time com Node.js*. Editora Casa do Código.

Pokorná, Jitka et al. (2015). «Value Proposition Canvas: Identification of Pains, Gains and Customer Jobs at Farmers' Markets». Em: *AGRIS on-line Papers in Economics and Informatics* 7.665-2016-45080, pp. 123–130.

Projects, The Pallets (2020). *Flask Quickstart*. `https://flask.palletsprojects.com/en/1.1.x/quickstart/#quickstart`. Acedido em: 2020-12-30.

Python, Full Stack (2020). *Flask*. `https://www.fullstackpython.com/flask.html`. Acedido em: 2020-12-30.

Rad, Babak Bashari, Harrison John Bhatti e Mohammad Ahmadi (2017). «An introduction to docker and analysis of its performance». Em: *International Journal of Computer Science and Network Security (IJCSNS)* 17.3, p. 228.

Saaty, Thomas L (1994). «How to Make a Decision: The Analytic Hierarchy Process». Em: *Interfaces* 24.6, pp. 19–43.

Saks, Elar (2019). «JavaScript Frameworks: Angular vs React vs Vue.» Em: *University of Applied Sciences*.

Sánchez-Fernández, Raquel e M Ángeles Iniesta-Bonillo (2007). «The concept of perceived value: a systematic review of the research». Em: *Marketing theory* 7.4, pp. 427–451.

Sauro, Jeff (2011). *Measuring Usability with the System Usability Scale (SUS)*. `https://measuringu.com/sus/`. Acedido em: 2021-09-20.

Sayfan, Gigi (2017). *Mastering Kubernetes*. Packt Publishing Ltd.

Scheepers, Mathijs Jeroen (2014). «Virtualization and containerization of application infrastructure: A comparison». Em: *21st twente student conference on IT*. Vol. 21.

Schwaber, Ken e Jeff Sutherland (2011). «The scrum guide». Em: *Scrum Alliance* 21, p. 19.

Schwarzmüller, Maximilian (2020). *Angular vs React vs Vue*. `https://academind.com/tutorials/angular-vs-react-vs-vue-my-thoughts/`. Acedido em: 2021-02-02.

Smart, John Ferguson (2011). *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. "O'Reilly Media, Inc."

Soppelsa, Fabrizio e Chanwit Kaewkasi (2016). *Native Docker Clustering with Swarm*. Packt Publishing Ltd.

Stackshare (2020). *Who uses Terraform?* `https://stackshare.io/terraform`. Acedido em: 2021-01-30.

Varia, Jinesh, Sajee Mathew et al. (2014). «Overview of Amazon Web Services». Em: *Amazon Web Services* 105.

Walls, Craig (2016). *Spring Boot in action*. Manning Publications.

Wohlgethan, Eric (2018). «Supporting Web Development Decisions by Comparing Three Major JavaScript Frameworks: Angular, React and Vue. js». Tese de doutoramento. Hochschule für Angewandte Wissenschaften Hamburg.

Xu, Jian e Xiaoming Liu (2018). «Technology is changing what a premium automotive brand looks like». Em: *Harvard Business Review*.

Zairi, Mohamed e Mohamed A Youssef (1995). «Quality function deployment: A main pillar for successful total quality management and product development». Em: *International Journal of Quality & Reliability Management*.