INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Análise e otimização de uma aplicação web

JOÃO FILIPE DA SILVA RIBEIRO
Outubro de 2019

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Análise e otimização de uma aplicação *web*

João Filipe da Silva Ribeiro

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

Orientador: Paulo Baltarejo Sousa

Porto, Outubro de 2019

Resumo

Na atualidade, a sociedade quando procura uma informação ou produto pretende encontrá-los no menor espaço de tempo possível. Com esta premissa, uma aplicação *web* precisa de ter o seu desempenho otimizado para satisfazer todos os seus clientes, mesmo os mais impacientes. Caso não o faça, corre o risco de os perder, diminuindo a sua quota de mercado. Assim, todos os segundos contam na indústria do comércio *online*.

No contexto do problema, as aplicações desenvolvidas atualmente tendem a acrescentar valor o mais rápido possível, descurando inicialmente a maneira como é realizado o código, ou como este deve ser pensado para que esteja preparado para receber novas funcionalidades sem grandes preocupações. Além disto, mesmo quando pensam em desempenho de uma página *web* tendem a preocupar-se apenas com a primeira impressão, não dando ênfase ao desempenho dos componentes criados com os quais o utilizador irá interagir nos momentos seguintes.

De forma a que seja possível ter uma aplicação *web* com um desempenho otimizado, foram desenvolvidos dois protótipos com o objetivo de se aplicarem metodologias e práticas diferentes aos seus componentes. Estas práticas foram obtidas a partir de um estudo aprofundado sobre o React, bem como da linguagem JavaScript na sua generalidade.

De forma a que se possa concluir quais as melhores práticas a aplicar, foram efetuadas medições em termos de tempo e de memória gastas, por forma a realizar comparações entre os vários casos. Assim, percebeu-se a importância do uso da função shouldComponentUpdate, bem como da virtualização de listas ou da memorização em cache.

Palavras-chave: E-commerce, React, Virtualização, Cache, Ciclo de vida

Abstract

Nowadays, when looking for a product or information society wants to find it in the shortest possible time. With this premise, a web application needs to have its performance optimized to satisfy all of its customers, even the most impatient ones. Failure to do so risks losing them, reducing their market share. Thus, every second counts in the online commerce industry.

In the context of the problem, currently developed applications tend to add value as quickly as possible, initially neglecting the way code is done, or how it should be thought so that it is prepared to receive new functionality without major concerns. Moreover, even when thinking about web page performance, they tend to worry only about the first impression, not emphasizing the performance of the created components that the user will interact within the following moments.

In order to be able to have a web application with an optimized performance, two prototypes were developed in order to apply different methodologies and practices to its components. These practices were derived from an in-depth study of React as well as the JavaScript language in general.

In order to conclude which best practices to apply, time and memory measurements were taken, in order to make comparisons between the various cases. With this, it was realized the importance of using the shouldComponentUpdate function, as well as list virtualization and cache memoization.

Keywords: E-commerce, React, Virtualization, Cache, Lifecycle

Agradecimentos

Primeiramente, agradeço ao Instituto Superior de Engenharia do Porto e ao Departamento de Engenharia Informática e seus docentes por todo o conhecimento que me foram transmitindo ao longo do meu percurso académico e que irei utilizar no meu futuro, tanto pessoal como profissional.

Ao Engenheiro Paulo Baltarejo de Sousa agradeço toda a disponibilidade para se reunir e discutir sobre o melhor caminho a tomar quanto à redação desta dissertação. Além disto, os seus conselhos foram e serão sempre úteis para o meu desenvolvimento pessoal.

A todos os que estão presentes, mas também a todos os que já partiram, agradeço o apoio incansável em todos os momentos e fases da minha vida. Certamente felizes e orgulhosos por verem o culminar de mais uma etapa, igualmente ansiosos por verem o que o futuro reserva.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Código

Acrónimos e Símbolos

Lista de Acrónimos

AHP	*Analytic Hierarchy Process*
API	*Application Programming Interface*
BDD	*Behavior Driven Development*
CD	*Continuous Delivery*
CI	*Continuous Integration*
CPU	*Central Processing Unit*
CSS	*Cascading Style Sheets*
DOM	*Document Object Model*
DSR	*Design Science Research*
E2E	*End to End*
FFE	*Fuzzy Front End*
HTML	*HyperText Markup Language*
NCD	*New Concept Development*
NPD	*New Product Development*
SAST	*Static Application Security Tool*
SPA	*Single Page Application*
SQL	*Structured Query Language*
TDD	*Test Driven Development*
XML	*Extensible Markup Language*
XSS	*Cross Site Scripting*

1 Introdução

Neste capítulo são apresentados alguns conteúdos de modo a contextualizar o leitor e a facilitar a leitura sobre o tópico a ser abordado ao longo deste documento. Posteriormente, é mencionado o problema que foi encontrado e os objetivos propostos a alcançar no fim desta dissertação. Para que uma solução seja implementada deve-se primeiro realizar uma análise do valor que esta trará, pelo que neste capítulo faz-se um resumo da análise de valor. Por fim, é mostrado um resumo da abordagem tomada e a estrutura que o documento possui.

1.1 Contexto

Nos dias de hoje ninguém gosta de um *website* lento. Com as velocidades de Internet que atualmente existem e com dispositivos eletrónicos que possuem processadores com múltiplos núcleos e com imensa memória, os utilizadores tornam-se impacientes quanto ao lento carregamento de uma página *web*. Se um *website* demora demasiado tempo a carregar, o utilizador irá fazer uma de duas coisas: clicar no botão de atualização do navegador várias vezes para ver se alguma coisa acontece ou então simplesmente procura uma outra página alternativa, de forma a encontrar o que pretende.

O desempenho de uma página é um assunto que já levou algumas das maiores empresas tecnológicas mundiais a investigarem o impacto que este tem nas vendas e na experiência do utilizador. Por exemplo, a Google (Farber, 2006) e a Amazon (Einav and Shalom, 2012) conseguiram concluir que um ligeiro atraso de cem milissegundos faz com que menos utilizadores frequentem as suas páginas e que haja uma menor conversão das visitas em vendas.

Por detrás de uma página existe uma aplicação concebida e pensada tanto por desenvolvedores de *software* como por gestores de produto que pretendem corresponder às necessidades do negócio e dos utilizadores. Ou seja, uma aplicação deve ser uma alternativa fiável às demais existentes e não possuir os mesmos problemas que as restantes.

O React é uma biblioteca JavaScript declarativa, eficiente e flexível para a criação de interfaces para o utilizador (Facebook, 2016). É sabido que existem outras *frameworks* que possuem as mesmas funcionalidades que o React, no entanto o estudo será focado nesta. Existe um pequeno estudo e comparação entre as três *frameworks* mais utilizadas no mercado no Anexo A.

1.2 Problema

Se perder visitantes não é motivo suficiente para convencer que o desempenho de uma página deve estar entre as principais prioridades no momento do desenvolvimento de *software*, então também existe o fator do custo de manter páginas que possuem um mau desempenho. Estas páginas terão de ser melhoradas, necessitarão de melhores servidores para ajudar a resolver alguma da lentidão, o que implica maiores custos em infraestrutura. Nos dias de hoje também se refere muito que uma aplicação deve ser escalável, pelo que se esta for lenta pode não conseguir fazer face a uma quantidade de pedidos maior. Tudo isto pode levar a que haja maior tempo de espera para uma resposta da página ou que a página fique indisponível por tempo indeterminado.

Além deste problema deve-se também pensar para lá da primeira sensação que a página fornece ao utilizador. Não adianta ter uma página inicial rápida se os componentes existentes nesta página, aquando da sua utilização, forem lentos, causando uma má experiência ao utilizador. Quando um utilizador clica num botão, espera que a página lhe mostre algum tipo de *feedback* de volta ao fim de dois segundos (Shneiderman, 1984). Se este tempo não for alcançado, então o utilizador irá começar a debruçar-se sobre outros pensamentos pendentes nos quais deve atuar, deixando para segundo plano a página ou até fechando a mesma.

1.3 Objetivos

O principal objetivo desta dissertação é avaliar e otimizar uma aplicação desenvolvida em React, no entanto dentro deste existem pequenos sub-objetivos que devem ser alcançados por etapas. Ao dividir em pequenos objetivos é possível ir percebendo gradualmente quais os passos a tomar e por que sequência devem ser feitos, dando uma sensação de continuidade ao leitor (Marchese, 2016). Assim, os sub-objetivos são:

- Estudar os padrões e anti padrões existentes em React. Um padrão é uma abordagem comum para resolver um tipo de problema comum, enquanto que um anti padrão é um comportamento geralmente observado que tende a reduzir a qualidade do código (Salman, 2018). Não basta fazer com que o código funcione, deve-se tentar sempre perceber o problema e aplicar uma solução reutilizável para problemas que ocorram recorrentemente (Osmani, 2012).
- Estudar as ferramentas utilizadas para auditoria de uma aplicação *web*. Para que se possa tirar maior partido da informação dada por estas ferramentas deve-se realizar um estudo prévio sobre o que elas analisam e quais as informações pertinentes que fornecem para que solucionem um determinado problema.
- Estudar os sintomas e as causas de quebra de desempenho de uma aplicação *web* e como podem ser contornados. Por vezes pensa-se que mudar uma pequena linha de código não possui grande impacto no resultado final, no entanto, a aplicação acaba por ficar mais lenta. Aliado ao estudo das ferramentas, deve-se utilizar o conhecimento obtido nesse momento e verificar quais as incongruências que a aplicação possui de forma a resolvê-las.
- Demonstrar, através de exemplos, como se podem tornar alguns componentes mais lentos em componentes mais rápidos, utilizando o conhecimento adquirido do estudo previamente efetuado.

1.4 Análise de valor

A criação de uma proposta de valor tem de começar sempre pela definição de uma ideia. Nesta dissertação pretende-se realizar uma aplicação *web* que consiga lidar com um elevado número de dados relacionados com uma loja virtual. Hoje em dia as lojas pretendem ter cada vez maior

oferta para os seus clientes, de modo a que cubram todos os seus gostos, convertendo-se em mais vendas. Aliados a isto, precisam que a sua aplicação responda de uma forma rápida às interações com os clientes.

Desta forma, tem de se perceber a perspetiva do consumidor e o valor que esta aplicação irá trazer para ele, tendo sempre em conta os custos associados, por exemplo a nível de tempo, mas também tendo em conta a experiência que deve ser proporcionada ao utilizador.

Para tudo isto, a análise de valor é realizada dentro dos modelos NCD (*New Concept Development*) e AHP (*Analytic Hierarchy Process*). Além destes, define-se o valor, o valor percecionado, o valor para o cliente e a proposta de valor. Por fim, o modelo de negócio Canvas completa o modelo de negócio desta aplicação.

1.5 Abordagem

Esta dissertação foca-se na procura de uma solução para um problema que pode ocorrer em qualquer organização e que tem impactos nefastos nas mesmas, tal como mencionado na Secção 1.1 e que será abordado em maior detalhe nos Capítulos 2 e 3. Desta forma, a dissertação seguirá a metodologia DSR (*Design Science Research*). Esta metodologia é amplamente aceite pela área de investigação de sistemas de informação e é composta por seis passos (Peffers *et al.*, 2007):

1. Identificação do problema e motivação: a definição do problema e a justificação do valor da solução. Esta justificação permite que o leitor aceite os resultados demonstrados e que perceba qual o problema que foi identificado. O problema identificado encontra-se na Secção 1.2 e a sua análise de valor encontra-se, de forma resumida, igualmente neste capítulo, mas também de forma detalhada no Capítulo 3.
2. Definição de objetivos para uma solução: a definição de objetivos para a solução do problema identificado. Estes objetivos encontram-se especificados na Secção 1.3.
3. *Design* e implementação: a criação do artefacto. Este passo inclui a arquitetura pensada e a identificação da funcionalidade pretendida. Posto isto, é desenvolvido o artefacto pensado. O *design* proposto encontra-se explicado de forma minuciosa no Capítulo 4. A implementação da solução preconizada é constatada no Capítulo 5.

4. Demonstração: é necessário provar que o artefacto desenvolvido consegue solucionar totalmente ou parcialmente o problema encontrado. Esta demonstração é alcançada através de experimentações e simulações na aplicação *web* desenvolvida. Este passo encontra-se explicitado no Capítulo 6.
5. Avaliação: pretende analisar a eficiência da solução proposta para o problema identificado. Nesta fase os objetivos são comparados com os resultados obtidos na demonstração efetuada. Caso a avaliação não alcance o resultado pretendido, o investigador pode voltar a um dos passos anteriores e melhorá-lo de forma a alcançar a avaliação pretendida. Este passo encontra-se detalhado no Capítulo 6.
6. Comunicação: Todo o trabalho efetuado deve ser comunicado para que este seja analisado e melhorado em futuras iterações. Esta fase é composta pela apresentação desta dissertação junto de um comité de avaliação.

1.6 Estrutura do documento

No primeiro capítulo é descrito o contexto da dissertação em causa, explicando igualmente o problema levantado, os objetivos a alcançar, bem como um breve resumo da análise de valor e da abordagem tomada.

No segundo capítulo, com o título de Estado da Arte, é feito um estudo do estado de arte, nomeadamente um enquadramento teórico e tecnológico relativamente ao assunto da dissertação.

No terceiro capítulo, denominado Análise de Valor, é feita a análise de valor que a nova aplicação proporciona. Para que se perceba o porquê da aplicação efetuada tem se por base modelos de negócio como o Canvas, o NCD e o AHP.

No quarto capítulo, enunciado como Análise e *Design* da Solução, especifica-se o *design* da solução para o problema, referindo os padrões e regras utilizados. Existe também uma secção onde se demonstram alternativas para que se entenda qual a solução final apropriada.

No quinto capítulo, com o título de Boas Práticas para Implementação, descreve-se a construção da solução em si e todas as peculiaridades que devem ser tidas em conta no momento.

No sexto capítulo, denominado Avaliação, existe uma descrição das experiências efetuadas e uma avaliação à solução criada. Explica-se como se pretende testar e avaliar o trabalho realizado de modo a perceber se alguns dos objetivos foram alcançados.

Por fim, no sétimo capítulo, designado Conclusões, existe uma série de conclusões que são retiradas a partir do problema encontrado, dos objetivos definidos e alcançados, bem como da avaliação da solução encontrada.

2 Estado da arte

Primeiramente, este capítulo pretende dar um enquadramento teórico ao leitor, dar-lhe a entender a importância da experiência do utilizador, não só em termos de *design* atrativo, mas também quanto ao desempenho da aplicação *web*. Posteriormente, realiza-se um enquadramento tecnológico onde se definem conceitos gerais e boas práticas a utilizar para obter um bom desempenho. Por fim, existe uma análise de soluções existentes no mercado.

2.1 Enquadramento teórico

O ser humano está cada vez mais ligado à tecnologia, pelo que este deve de estar ciente que interagir com a tecnologia tem impactos em termos emocionais, intelectuais e sensoriais. Por esta razão, quem cria o *design* gráfico tem de perceber e analisar o que um utilizador sente ao interagir com tecnologia (McCarthy and Wright, 2004). Assim, existem vários fatores que devem ser tidos em conta como a usabilidade, a acessibilidade, a estética, a ergonomia, o desempenho, entre outros (Cousins, 2017). Nesta dissertação existe um foco no desempenho, no entanto, é possível perceber que existem várias áreas a ter em conta antes de se pensar numa solução final para um projeto.

O desempenho é uma quantidade de trabalho útil realizada por um sistema de computador em comparação com o tempo e os recursos usados, ou seja, um melhor desempenho significa mais trabalho realizado em menos tempo, usando ou não, menos recursos. Assim, o desempenho de uma aplicação é cada vez mais um tópico que merece a atenção das organizações. Todos os utilizadores já desistiram de procurar algum assunto na Internet porque a página estava a

demorar muito a aparecer, porque esta era muito lenta após a página principal estar carregada ou porque as suas funcionalidades não correspondiam por inteiro ao que se pretendia. Ao se saber que esta página tem problemas, esta irá deixar de ser considerada no futuro como uma opção viável para obter uma informação pretendida, fazendo com que a página perca mais um possível cliente.

O desempenho de uma página *web* deve ter como preocupações manter os clientes e melhorar a conversão de mero visitante para cliente que atinge a finalidade pretendida, por exemplo, visitar pela primeira vez um site de *e-commerce* e realizar nesse momento a compra de um artigo. Para isto, as suas páginas devem estar preparadas para receber visitas de pessoas tanto com boas conexões à Internet como com piores conexões (Iriondo, 2018). Desta forma, estarão a tornar o seu produto apto a ser visitado por um maior número de visitantes.

Além destes fatores, o tempo que uma página demora a carregar é tido em conta nos resultados de um motor de busca, ou seja, as páginas *web* que forem mais rápidas serão apresentadas em primeiro lugar em detrimento das restantes (Hogan, 2014). Estudos revelam que até trinta e cinco por cento do tempo de espera para a renderização de uma página são causados por computações ligadas a linguagens como HTML (*HyperText Markup Language*) ou JavaScript (Wang *et al.*, 2013). Estas linguagens são as mais comumente utilizadas para páginas *web* e nem sempre são utilizadas seguindo as melhores práticas. Para que se sigam boas práticas devem ser estudadas a fundo as linguagens e ao serem utilizadas num projeto devem seguir metodologias como TDD (*Test Driven Development*). Esta técnica é utilizada para desenvolver *software* guiado pela escrita de testes, ou seja, num primeiro momento deve-se escrever um teste para uma funcionalidade que se pretende adicionar. Num segundo momento deve-se realizar o código funcional até este passar nos testes criados. Por fim, é aconselhado rever e melhorar o código tanto novo como antigo de forma a este estar bem estruturado (Fowler, 2005).

2.2 Enquadramento tecnológico

Nesta secção inicialmente são descritos conceitos gerais que são úteis para a compreensão da restante dissertação. De seguida, são mencionadas boas práticas utilizadas para que a página *web* tenha um bom desempenho inicial. Além disto, descrevem-se algumas das ferramentas de auditoria existentes no mercado, finalizando esse estudo com uma comparação entre elas. Por

último, descrevem-se igualmente algumas das *frameworks* para desenvolvimento *web* mais conhecidas no mercado, comparando no final os seus pontos fortes e fracos.

2.2.1 Conceitos gerais

Para que um leitor perceba inteiramente o que é mencionado neste documento é necessário detalhar alguns conceitos. Assim, nesta secção explicita-se conceitos como a estrutura de uma página no navegador, cache, DOM (*Document Object Model*), *Render Time*, *Time to First Meaningful Paint*, entre outros.

2.2.1.1 Estrutura de uma página no navegador

As páginas *web* têm tendência a serem todas diferentes umas das outras, no entanto todas elas tendem a partilhar componentes padrão. Uma página *web* pode ser considerada uma árvore de documentos, geralmente denominada *document tree*, que contém um qualquer número de ramificações (Lechat, 2014). Existem regras que especificam o que cada ramo pode conter:

- *Doctype*

O primeiro item a aparecer no código fonte é a declaração do *doctype*. Isto providencia ao navegador informação sobre qual o tipo de linguagem em que a página está escrita, o que pode afetar a maneira como o navegador renderiza o conteúdo. Exemplo disto é o HTML 4.01 Strict, este valida o conteúdo contra a especificação do HTML 4.01, no entanto não permite, por exemplo, utilizar elementos que já tenham deixado de ser suportados pela *web*.

- Elemento *html*

O elemento *html* aparece imediatamente a seguir ao *doctype*, sendo a raiz de toda a árvore de documentos. Este elemento divide-se em duas secções principais, *head* e *body*.

- Elemento *head*

O elemento *head* contém informação que descreve o documento em si, ou associa-o a outros recursos como *scripts* ou folhas de estilização. Os *scripts*, com o elemento *script*, tendem a ser construídos em JavaScript e é onde parte da lógica da página se encontra. As folhas de estilização, com o elemento *style*, permitem, por exemplo, alinhar ou colorir um determinado componente visual da página para que este seja mais apelativo ao utilizador.

O elemento *head* pode conter elementos como *title*, onde se identifica o nome do documento. Este elemento, quando utilizado, aparece na aba do navegador e também como opção para título de marcador nos favoritos do navegador.

Outros elementos contidos no *head* são por exemplo o *base*, o *link* ou o *object*. Este último é um elemento genérico que pode ter múltiplos propósitos para media.

- Elemento *body*

O elemento *body* tem dentro todo o conteúdo que é mostrado para o utilizador, incluindo parágrafos, listas, imagens, tabelas, entre outros. A maneira como a página *web* é mostrada dependerá sempre do conteúdo que é colocado neste elemento. Pode contar dentro dele elementos como o *header*, o *footer* e a *div*.

- Elemento *header*

O elemento *header* representa o conteúdo introdutório. Tendo como raiz o elemento *body*, este define o cabeçalho global da página. No entanto, caso esteja dentro de um elemento *article* ou *section*, já será um cabeçalho específico dessa secção.

- Elemento *footer*

Similar ao elemento *header*, o elemento *footer* representa o grupo de conteúdos do final da página. Contudo, também pode ser utilizado dentro dos elementos *article* ou *section* se se pretender restringir a apenas uma secção da árvore de documentos.

- Elemento *div*

O elemento *div* é usado principalmente para estabelecer uma hierarquia de conteúdo. Esta *div* define uma secção ou divisão de uma árvore de documentos, sendo o elemento mais utilizado hoje em dia nos navegadores devido à sua simplicidade e extensibilidade.

2.2.1.2 DOM

A DOM é uma interface que representa a maneira como os documentos de HTML e XML (*Extensible Markup Language*) são lidos pelo navegador. Permite que a linguagem JavaScript manipule, estruture e estilize o *website*. Após o navegador ler o documento HTML, este cria uma árvore representacional, denominada DOM e define como essa árvore pode ser acedida (Maldonado, 2018).

Com a possibilidade de manipular a DOM permite-se que sejam criadas aplicações que atualizam dados da página sem ser necessária a atualização da página como um todo. Também

se torna possível o utilizador criar as suas aplicações de forma customizável, com o conceito de *drag & drop*, ou seja, de se utilizar o rato do computador para se mover elementos dentro da página.

Assim e em complemento ao que foi dito na Subsecção 2.2.1.1, existe uma DOM, constituída por vários elementos e atributos, como sugere a Figura 1.

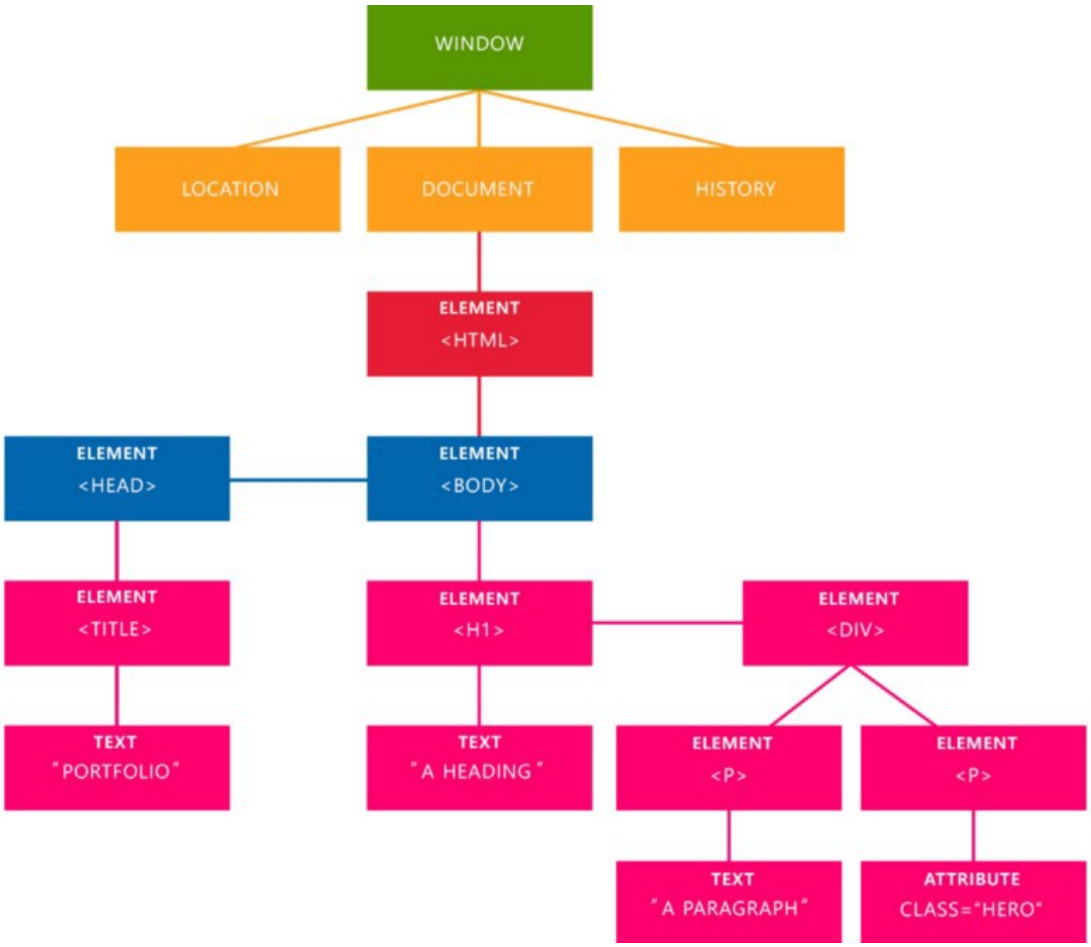

Figura 1 - Árvore representacional que um navegador cria após ler um documento (Maldonado, 2018).

Todos estes elementos devem ser vistos como nós dentro de uma árvore, sendo possível acedê-lo através de métodos específicos, como também se lhes pode atribuir eventos que são despoletados se uma determinada ação ocorrer.

- Métodos

Existem métodos como o getElementById() em que o objetivo é ir buscar um elemento através do identificador que este possui. Cada identificador é único e por isso não existe preocupação de estar a ir buscar um elemento que não o pretendido.

Outro método existente é o querySelector(), que tem como objetivo ir buscar o primeiro elemento que tiver um determinado seletor de CSS (*Cascading Style Sheets*) que foi passado como parâmetro na função.

- Eventos

Os eventos são também métodos, no entanto são os responsáveis pela interatividade do utilizador com a página *web*.

O evento de *click* é o mais utilizado em toda a *web*, permitindo com isto realizar uma determinada funcionalidade no momento em que o utilizador clica num específico elemento. Por exemplo, caso se queira escrever no interior de uma caixa de texto, o utilizador pode clicar com o rato dentro dessa mesma caixa, disparando o evento de *click* que irá fazer o cursor aparecer dentro da caixa e permitir a introdução de informação por parte do utilizador.

Caso o utilizador realize essa introdução de informação, poderá ser despoletado um outro evento, denominado *change*, onde este modificada o valor que o elemento possuía até então para o que o utilizador pretende. Este evento pode ser despoletado ou não consoante as pretensões do desenvolvedor da página, no entanto o evento *input* tem a mesma funcionalidade só que é sempre despoletado sincronamente.

2.2.1.3 Compactação de dados

A compactação de dados refere-se ao processo de conversão de um fluxo de entrada, contendo dados em tamanho original, num fluxo de saída, onde os dados possuem um tamanho menor. Este fluxo pode ser um arquivo ou um *buffer* de memória. A compactação de dados é cada vez mais necessária porque as informações que geradas e usadas estão em formato digital, ou seja, na forma de números representados por *bytes* de dados, podendo este número ser enorme (Gupta, Kumar and Rohit, 2016).

Existem dois tipos de compactação de dados: *lossless* e *lossy*. A compactação *lossless* é uma técnica onde nenhum dado é perdido. Uma réplica do ficheiro original é passível de se obter caso se descompacte o ficheiro compactado. Este tipo de compactação é geralmente utilizado

para guardar ou transmitir dados, sendo o mais comumente utilizado para textos (Gupta, Kumar and Rohit, 2016).

A compactação *lossy* é geralmente utilizada para imagens, áudio e vídeo. Nesta técnica de compactação é ignorada informação menos importante, sendo então impossível de obter o ficheiro original caso se faça a desencriptação do ficheiro encriptado (Gupta, Kumar and Rohit, 2016).

2.2.1.4 Cache

A cache é *hardware* ou *software* que permite guardar dados temporariamente num ambiente computacional. Trata-se de uma memória pequena, mas mais rápida que permite melhorar o desempenho de dados que são visualizados frequentemente, ou que tenham sido necessários num passado recente. A cache é utilizada, pois o local principal onde os dados são guardados nem sempre consegue aguentar as necessidades constantes de passagem de dados. A cache permite, assim, encurtar o tempo que se demora a obter os dados, reduzindo também a latência. No geral, utilizando a cache existe uma melhoria de desempenho das aplicações (Rouse, 2018).

Quando uma aplicação cliente que possui cache necessita de aceder a dados, este vai pesquisar primeiro se os dados pretendidos se encontram armazenados na cache. Caso não se encontrem os dados pretendidos na cache, será efetuado um pedido ao local onde estão os dados todos armazenados e esses novos dados requisitados irão ser guardados em cache para que num futuro próximo sejam mais facilmente acedidos (Rouse, 2018).

2.2.1.5 Page Load

Page Load é geralmente definido como o tempo que leva o navegador a despoletar o manipulador de eventos *window.onload* da linguagem JavaScript. Idealmente, o tempo de carregamento de uma página deve de ser menor que dois segundos. Consequentemente, tornou-se prática comum atrasar o carregamento de certos elementos até que o evento *onload* tenha sido executado. Assim, permite-se que os ativos importantes sejam carregados primeiro, sendo de seguida carregados os *scripts* de terceiros (Arsenault, 2017).

2.2.1.6 Time to First Meaningful Paint

Existe uma métrica denominada *Time to First Paint*, que mostra o tempo que os utilizadores têm de esperar até verem algo mais que uma imagem branca no navegador. Apesar de ser uma

métrica útil, esta pode esconder alguns problemas de desempenho visto o utilizador ainda não ter a página totalmente funcional. Assim, existe uma outra métrica chamada *Time to First Meaningful Paint*. Esta métrica permite perceber quanto tempo se demora a que haja conteúdo realmente importante e útil para o utilizador poder ver e navegar no navegador (Arsenault, 2017).

2.2.1.7 Speed Index

Por último, existe uma métrica com o nome de *Speed Index*. Esta representa o tempo médio em que as partes visíveis da página são exibidas, sendo expressa em milissegundos e o seu resultado depende do tamanho da janela de exibição do navegador. Para obter este índice, precisa-se de calcular o quão completa a página *web* está em vários momentos ao longo do carregamento da página. Existem dois métodos de calcular o quão completa uma página *web* se encontra. Um dos métodos baseia-se na captura de vídeo, onde é feita uma gravação do carregamento da página no navegador, sendo de seguida realizada uma inspeção a cada *frame* do vídeo. O outro baseia-se em eventos de pintura realizados pelo navegador aquando do carregamento da página *web*.

2.2.2 Boas práticas para obter um bom desempenho

Nos próximos tópicos são abordados alguns exemplos de boas práticas de forma a que uma aplicação *web* possua um bom desempenho. Outras boas práticas serão mostradas e aplicadas nos Capítulos 5 e 6.

2.2.2.1 Compactar recursos

A aplicação *web* desenvolvida pode possuir alguns megabytes de tamanho, pelo que obrigar o utilizador a descarregar uma enormidade de recursos irá tanto aumentar o tempo de espera do utilizador para poder mexer na página, como irá gastar maior quantidade do pacote de dados do utilizador. Assim, deve-se recorrer a compressores de dados como o gzip e o brotli. Neste momento o gzip é o mais utilizado pois é o que todos os navegadores suportam, contudo, o brotli, apoiado pela Google, tem vindo a aumentar de popularidade e possui uma maior compressão. No entanto, para navegadores mais antigos não existe de momento suporte (Google, 2018b).

2.2.2.2 Reduzir recursos

Por vezes não basta compactar os recursos como anteriormente explicado, também é necessário reduzi-los, removendo dados desnecessários ou redundantes sem afetar os recursos a ser processados pelo navegador. Para isto pode-se, por exemplo, utilizar bibliotecas como o UglifyJs, de forma a reduzir a quantidade de JavaScript que o utilizador necessita de descarregar para que a página funcione (Google, 2018c).

2.2.2.3 Cache no navegador

Procurar recursos na rede pode ser uma tarefa lenta e dispendiosa. O utilizador pode estar numa determinada página, mudar para uma segunda página, mas no momento seguinte querer voltar à página inicial. Para evitar que em todas estas três atividades haja pedidos ao servidor pode-se ativar a opção de guardar dados em cache. Assim, o utilizador terá rapidamente acesso aos dados anteriormente mostrados, tendo um tempo de espera menor e não sobrecarregando o servidor (Google, 2018a).

2.2.2.4 Reduzir o tamanho da DOM

Ter uma DOM demasiado grande pode afetar o desempenho de várias formas:

- Eficiência da rede e desempenho de carga. Se o servidor enviar uma árvore DOM demasiado grande, o cliente irá receber conteúdos desnecessários. Isso irá diminuir a rapidez da página pois o navegador terá de estruturar todos os nós, mesmo que estes não sejam visíveis.
- Desempenho em tempo de execução. Como os utilizadores e outros programas interagem com a página, o navegador terá de estar constantemente a recalcular posições e estilos dos nós. Combinar uma grande árvore DOM com regras de estilo complexas podem provocar uma lentidão severa.
- Desempenho da memória. Se se estiver a utilizar seletores ou outro tipo de funções que guardem muitas referências em muitos nós de forma desnecessária, pode-se estar a requerer demasiado das capacidades dos dispositivos dos utilizadores.

Como recomendação deve-se possuir sempre menos de mil e quinhentos nós no total, tendo estes no máximo uma profundidade de trinta e dois níveis. Nenhum nó deve ser pai de mais de sessenta nós (Google, 2019c).

2.2.3 Ferramentas de auditoria/avaliação

Para que a aplicação possa ter um melhor desempenho tem-se sempre em mente alguns passos que devem ser executados. Devem ser feitas medições da aplicação, analisar o resultado dessas mesmas medições, traçar objetivos de desempenho, trabalhar para alcançar esses objetivos e por fim repetir todo este processo. Esta secção foca-se nos dois primeiros passos.

No mercado existem várias ferramentas que fazem a auditoria de aplicações e que permitem perceber quais as partes que mais necessitam de atenção.

2.2.3.1 WebPagetest

A ferramenta WebPagetest (WebPageTest, 2019) é uma das mais conhecidas ferramentas a nível mundial para a análise do desempenho de uma aplicação *web*. Quando se entra no seu *website* este pede para introduzir qual o *website* que se pretende analisar e assim num só clique efetuar um teste básico. No entanto, esta ferramenta tem muitas variáveis que podem ser configuradas para que se possa tirar um maior número de informação pertinente. Existe capacidade de testar com diferentes navegadores, com uma conexão mais limitada que a que se possui, pedir para que sejam feitos testes como se estivesse a ser utilizado um dispositivo móvel, mesmo que neste momento se esteja a aceder a esta ferramenta através um computador, entre outros. É uma ferramenta que dá grande liberdade, no entanto nem sempre fornece dados fidedignos. Caso esta ferramenta esteja a ser acedida por muita gente os seus servidores não parecem aguentar bem a carga e acabam eles mesmos por ser lentos a fazer pedidos ao *website* fornecido. Além disto, há a possibilidade de existir uma fila de espera para que o teste seja efetuado. Uma das soluções é implementar um servidor privado e correr uma instância desta ferramenta localmente, no entanto acaba por ser mais custoso em termos de tempo de configuração e consome mais recursos que as ferramentas seguintes.

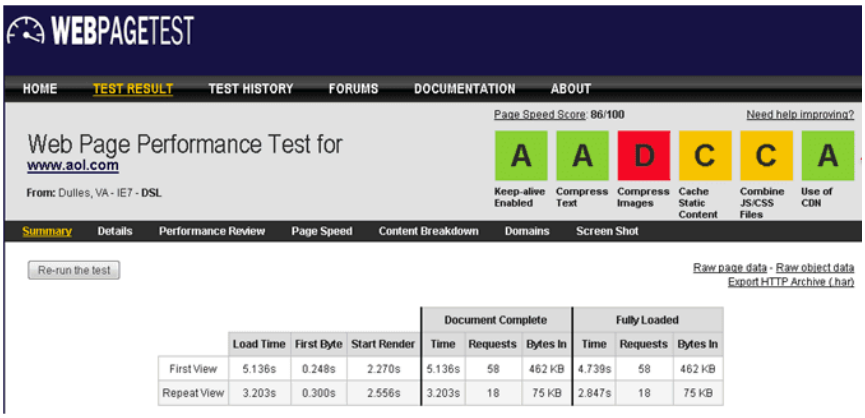

Figura 2 - WebPagetest: Exemplo de relatório obtido com a ferramenta WebPagetest.

Através da Figura 2 pode-se ver algumas métricas que são importantes de conhecer no momento de avaliar o desempenho de uma página *web*. Entre elas estão:

- *Load Time* – tem o mesmo significado que o *Page Load* descrito na Subsecção 2.2.1.5;
- *Fully Loaded* – pretende medir o tempo que demorou a que uma página *web* estivesse totalmente carregada e deixasse de existir qualquer tipo de atividade na rede durante dois segundos;
- *First Byte* – é o tempo que demora desde o início da navegação até ao primeiro *byte* da página ser recebido pelo navegador;
- *Start Render* – tem um significado igual ao *Time to First Paint* já mencionado na Subsecção 2.2.1.6.

2.2.3.2 Lighthouse

Ao contrário da ferramenta anteriormente descrita, o Lighthouse (Google, 2019a) não possui fila de espera, pode ser utilizado ininterruptamente através da linha de comandos ou estando integrado no Google Chrome. Para isso, tem-se de instalar uma extensão que habilita a funcionalidade. Assim, pode-se auditar as aplicações sempre que pretendido, recebendo no final uma pontuação de cada parte avaliada e sugestões de melhorias que devem ser realizadas. Pode-se avaliar boas práticas de engenharia, acessibilidade, o tempo que demora a renderizar informação pela primeira vez no navegador do utilizador, entre outros. Aliado a isto existe bastante documentação explicativa do problema em si e de como se pode solucioná-lo. Uma das desvantagens que tanto esta ferramenta como a anterior possuem é a de não testarem o desempenho dos componentes após a renderização da página, ou seja, uma página pode possuir um componente *button* ou *dropdown* que tenha um fraco desempenho que este não será detetado neste momento.

Na Figura 3 conseguem-se visualizar seis métricas que a ferramenta considera importantes, que são:

- *First Contentful Paint* – o tempo que a primeira parte do conteúdo do DOM é renderizado. Enquanto que o *First Paint* apenas se refere ao primeiro *pixel* renderizado, esta métrica avalia um estado posterior da página *web* no navegador, mas antes do *First Meaningful Paint*, onde o navegador renderiza conteúdo que o utilizador esteja interessado;

- *First Meaningful Paint* – esta métrica encontra-se referida no tópico anterior, mas também na Subsecção 2.2.1.6;
- *Speed Index* – esta métrica encontra-se referida igualmente na Subsecção 2.2.1.7;
- *First CPU* (*Central Processing Unit*) *Idle* – trata-se de uma métrica que mede quando a página *web* se encontra minimamente interativa, ou seja, quando a grande maioria dos elementos visuais já se encontra renderizados e a página corresponde também à maioria das ações despoletadas pelo utilizador em tempo razoável. Anteriormente denominada de *First CPU Idle*;
- *Time to Interactive* – este momento acontece sempre posteriormente aos restantes anteriormente mencionados, pois é o momento no qual a página *web* possui já todo o seu conteúdo renderizado de forma estável e funcional, permitindo assim ao utilizador poder navegar com a melhor experiência de utilização possível da respetiva página;
- *Estimated Input Latency* – esta métrica pretende estimar quão rápido a página *web* responde a uma interação que tenha com o utilizador, ou seja, quando clicamos num determinado botão ou numa caixa de texto, quanto tempo a página *web* demora a receber e processar essa informação.

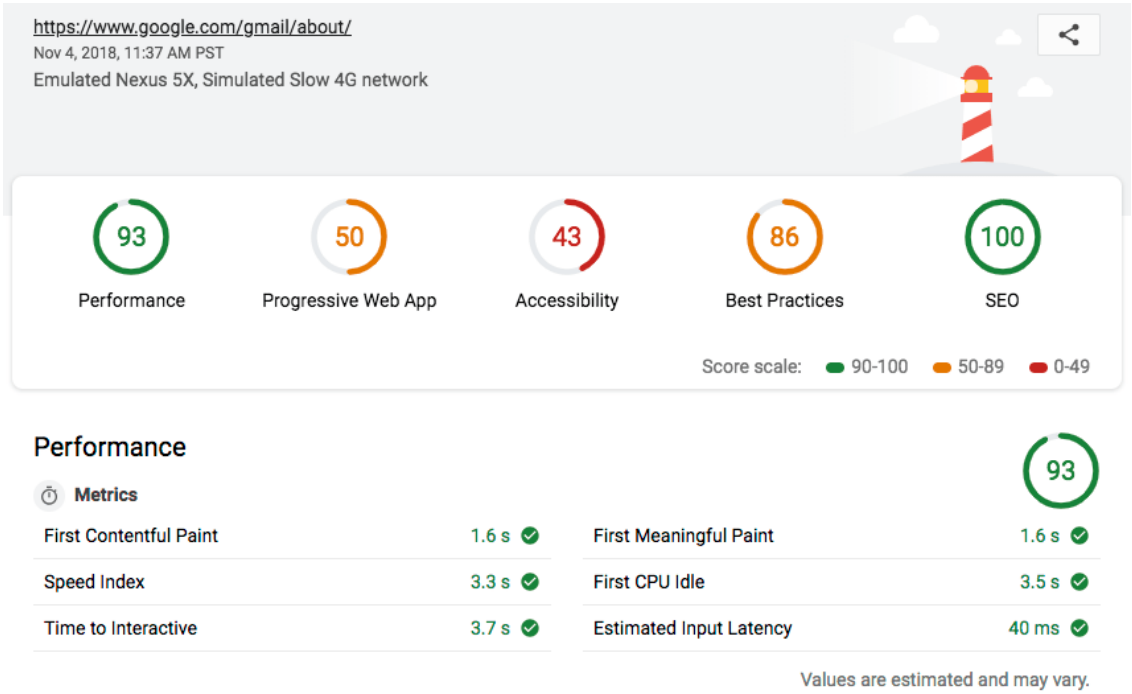

Figura 3 - Lighthouse: Exemplo de relatório obtido com a ferramenta Lighthouse.

2.2.3.3 Chrome Dev Tools – Performance Tab

O Chrome Dev Tools – Performance Tab (Basques, 2019), ao contrário das outras, consegue avaliar o desempenho de uma aplicação *web* ao longo de uma utilização exemplificativa do que um utilizador faria. Assim, pode-se depreender onde se localiza o problema da aplicação e em parte da retenção de utilizadores. Por exemplo, é possível simular um processador com um menor número de núcleos lógicos de forma a perceber o que acontece aos utilizadores que possuem dispositivos móveis ou mesmo fixos, mas com menor potência que os dos desenvolvedores. Com isto, pode-se ficar a perceber o que um utilizador visualiza e como a página reage aos seus movimentos.

Outro problema comum e que pode ser detetado com esta ferramenta são os chamados *memory leaks*. Um programa que usa explicitamente alocação e desalocação possui um *memory leak* quando não consegue libertar os objetos que não irá conseguir aceder no futuro. Um programa que usa *garbage collection* tem um *memory leak* quando retém a referência para objetos que não irá conseguir aceder no futuro (Nguyen and Rinard, 2007).

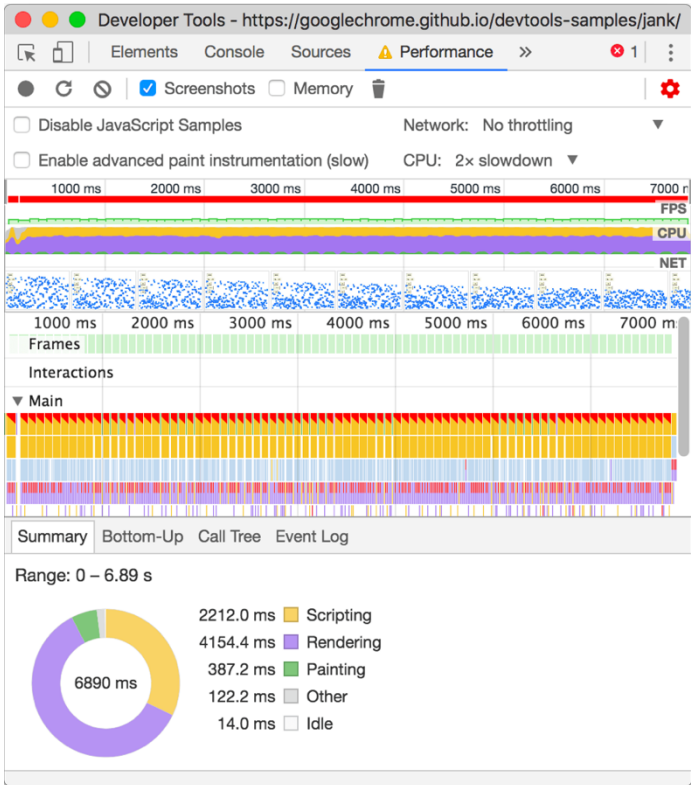

Figura 4 - *Google Chrome Performance Tab*: Exemplo de relatório obtido com esta ferramenta.

Com a Figura 4 pode-se perceber que existem detalhes como o tempo que o navegador passa a renderizar componentes gráficos ou a executar *scripts* necessários ao funcionamento da

página. Estes tempos são uma das principais ajudas para se perceber onde a aplicação está a quebrar. Além disto, como é mostrado na Figura 5, também se pode perceber o momento em que ocorrem algumas das métricas anteriormente mencionadas como o *First Meaningful Paint*, entre outras.

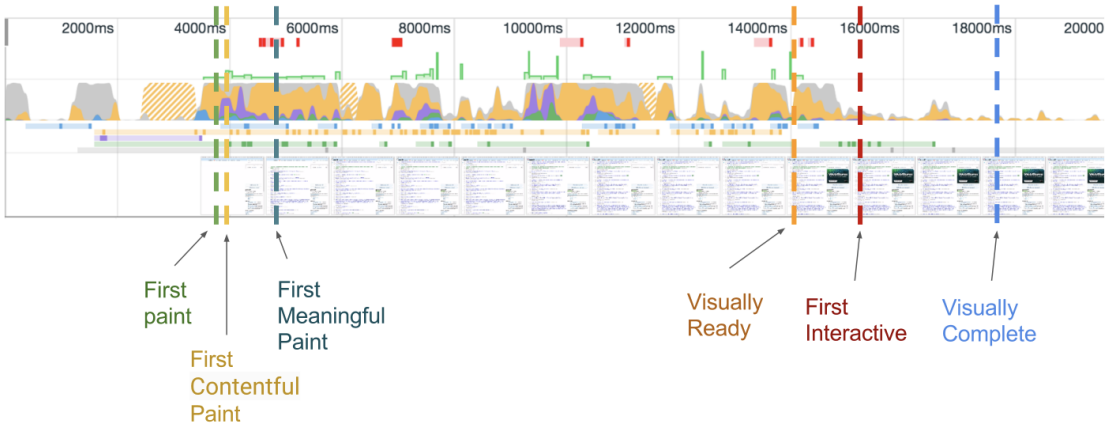

Figura 5 - Espaço temporal onde se pode visualizar os momentos em que são despoletas várias métricas (Denysov, 2017).

2.2.3.4 React Profiler

Com a chegada do React versão 16.5 foi disponibilizada uma API (*Application Programming Interface*) denominada Profiler (Vaughn, 2018) que tem como objetivo agregar informações relativas ao tempo que cada componente é renderizado, ajudando assim a perceber quais os componentes que possuem um mau desempenho e que precisam de ser revistos. De forma a que este processo seja passível de ser visualizado, deve ser instalada a extensão React Developer Tools no navegador em que pretendemos testar a nossa aplicação, sendo neste caso o Google Chrome.

Para se obter as informações relativas ao tempo que cada componente demora a executar deve-se ao painel Profiler e clicar no botão existente para que a partir desse momento, qualquer interação que seja efetuada com a aplicação *web*, seja armazenada. Por fim, é possível parar o registo de atividade da aplicação, recebendo de seguida um relatório com toda a informação angariada. Entre esta informação encontra-se o tempo e o número de vezes que cada componente esteve a ser renderizado. A Figura 6 mostra um exemplo de informação angariada de uma aplicação *web*.

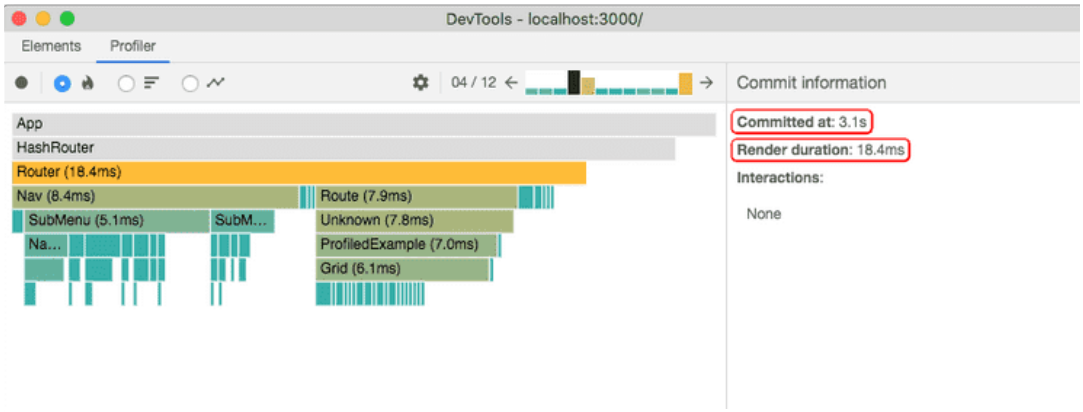

Figura 6 - Exemplo React Profiler (Vaughn, 2018).

2.2.3.5 Comparação entre ferramentas

Com o término do estudo de algumas das ferramentas existentes no mercado deve-se realizar uma comparação de forma a que se possa perceber qual ou quais as que servem para o propósito desta dissertação. Assim, a Tabela 1 apresenta os pontos a favor e contra de cada ferramenta.

Tabela 1 - Comparação entre ferramentas.

Ferramentas	**Pontos fortes**	**Pontos fracos**
WebPagetest	Possui uma grande variedade de configurações possíveis (escolher o dispositivo de testes, bloquear determinados pedidos no momento de arranque da página); Permite que se crie um servidor próprio para consumo interno.	Podem existir implicações nos resultados dos testes devido à localização onde se efetua o teste; Apenas possível testar o momento em que a página *web* carrega; Fila de espera.
React Profiler	Permite visualizar o tempo que cada componente consome e o número de vezes que é renderizado. Tem o suporte do Facebook.	Não permite medir outras grandezas sem ser o tempo que demora um componente a ser renderizado, como por exemplo, a memória.

Ferramentas	Pontos fortes	Pontos fracos
Lighthouse	É uma ferramenta *open source* que permite diversas configurações tal como o Webpagetest; Permite inclusão numa *pipeline* para que caso o desempenho da página *web* não esteja de acordo com os mínimos estipulados não seja feito *deploy* da mesma; Permite inclusão nos protótipos para que sejam efetuados testes localmente; Tem o suporte da Google.	Apenas é possível testar o momento em que a página *web* carrega e não posteriores ações que o utilizador realiza.
Chrome Dev Tools – Performance Tab	É uma ferramenta que permite diversas configurações; É a única que permite perceber o que acontece à página *web* para lá do momento em que se encontra totalmente pronta e estável para o utilizador interagir; Tem o suporte da Google.	Apenas existe no Chrome, podendo as páginas *web* não ficarem otimizadas para os restantes navegadores.

Com esta análise realizada percebe-se que não existe uma solução ideal, pelo que a junção da ferramenta React Profiler e Chrome Dev Tools – Performance Tab, permitem alcançar o objetivo principal desta dissertação, o de analisar e otimizar uma aplicação *web*. A ferramenta React Profiler colmata um ponto fraco existente na Performance Tab ao se conseguir perceber exatamente qual o componente que precisa de ser revisto e não a aplicação como um todo.

2.3 Sumário

Neste capítulo é possível ficar a perceber o enquadramento teórico em que esta dissertação se encontra, fazendo passar a informação de que os utilizadores cada vez mais são exigentes com o tempo de resposta de uma aplicação *web* e que esta é uma era tecnológica em que se pretende ter tudo no momento, sem qualquer tipo de esperas.

Também se pode perceber as tecnologias que existem para o desenvolvimento *web* e enumerar algumas boas práticas que levam a que haja bom desempenho das aplicações e que façam os clientes satisfeitos com a experiência que lhes é providenciada. Por último, também se faz uma comparação entre ferramentas de auditoria e avaliação, percebendo assim quais as que se adequam mais ao que se pretende avaliar neste documento.

3 Análise de valor

Neste capítulo faz-se uma análise de valor ao que esta dissertação trará, utilizando modelos já comprovados globalmente. Inicialmente, descreve-se o modelo NCD, abordando e analisando a oportunidade identificada, existe também uma geração e seleção de ideias e por último o desenvolvimento dos conceitos. Outro modelo utilizado é o Canvas, este permite facilitar a visão geral do negócio. Por último, o método AHP torna possível o uso de critérios qualitativos e quantitativos no processo da avaliação das ideias e desenvolvimento.

3.1 Modelo New Concept Development

O modelo NCD tem como intenção ajudar a gerir melhor os estados iniciais do processo de inovação e fornecer uma linguagem comum nas atividades. O processo de inovação divide-se em três partes:

1. *Fuzzy Front End* (FFE) – Trata-se de um ambiente de incerteza e imprevisibilidade no que toca a espaços temporais, a análises de ideias e na identificação das oportunidades.

2. *New Product Development* (NPD) – É um ambiente disciplinado e planeado com foco nos objetivos a alcançar. É também nesta parte que é desenvolvido o produto ou serviço.

3. Comercialização – É o último passo, onde se pretende divulgar e vender o produto ou serviço que foi desenvolvido e planeado na parte anterior.

Posto isto, o modelo NCD permite que sejam definidos os padrões chave do FFE. Uma das suas características é ser um modelo circular composto por cinco elementos. Todos estes estão interligados a um motor que dá ignição a cada um dos elementos. Não são sujeitos a qualquer ordem em particular e podem ser realizados as vezes que forem necessárias (Gassmann and Schweitzer, 2014). Os cinco elementos são:

- Identificação da oportunidade;
- Análise da oportunidade;
- Geração de ideias;
- Seleção de ideias;
- Desenvolvimento do conceito.

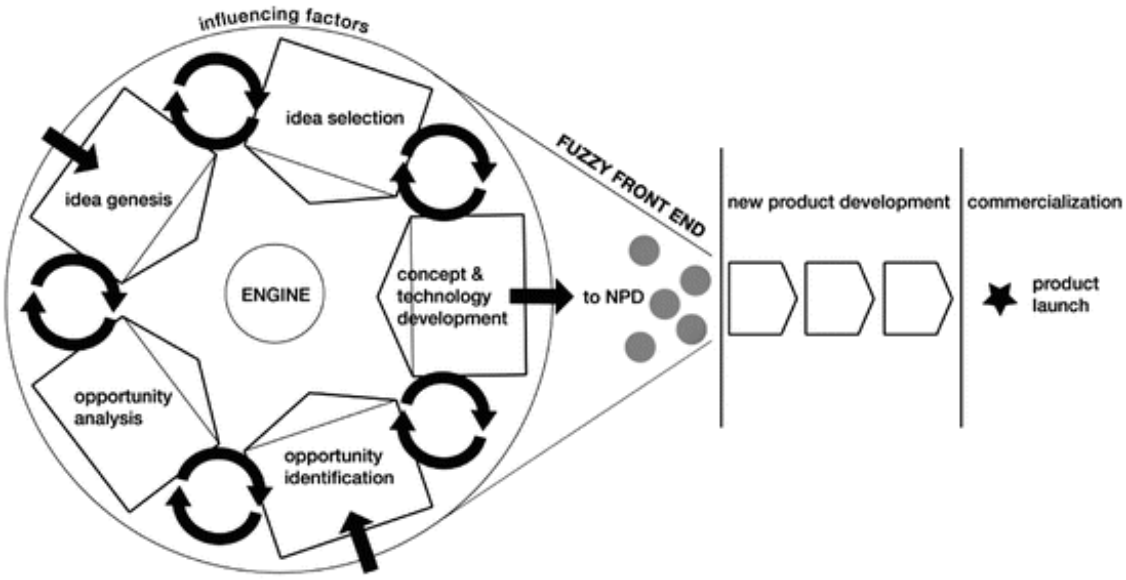

Figura 7 - Modelo NCD e restantes partes (Gassmann and Schweitzer, 2014).

3.1.1 Identificação da oportunidade

Neste elemento a pessoa ou empresa identifica a oportunidade que pode vir a prosseguir. Esta oportunidade pode levar a que haja uma mudança na direção do negócio ou que haja uma melhoria em produtos existentes. Para esta identificação é possível utilizar técnicas como o *brainstorming*, o *mind mapping*, entre outros (Koen *et al.*, 2002).

Neste caso, a identificação da oportunidade ocorreu quando se constatou que muitas organizações não se estavam a preocupar com a qualidade das suas aplicações *web*, mas apenas em entregá-las, assumindo que isso por si só traria o máximo de valor. Assim, existe uma oportunidade de perceber o impacto que esta despreocupação pode causar numa solução final em termos tecnológicos.

3.1.2 Análise da oportunidade

Neste elemento, uma oportunidade é avaliada para confirmar que vale a pena ser prosseguida. Informações adicionais são necessárias para traduzir a identificação de oportunidades em oportunidades específicas de negócio e tecnologia.

Neste caso, a oportunidade foi analisada quando se investigou quanto tempo os utilizadores estão dispostos a esperar para que uma página *web* esteja funcional para eles. Também se investigou o impacto que a tomada de decisões erradas quanto à arquitetura pode ter na solução final.

3.1.3 Geração de ideias

Este elemento pretende transformar a ideia estudada numa proposta concreta. Não é expectável que este pensamento seja perfeito no momento da sua criação, passando por várias iterações onde será melhorado através de discussões e análises.

Desta forma, surgem algumas ideias:

1. Criar uma ferramenta para analisar o desempenho de uma aplicação *web* como um todo;
2. Criar uma ferramenta de compressão de dados como o anteriormente mencionado gzip.
3. Criar dois protótipos onde:
    - O primeiro protótipo não siga as boas práticas de engenharia e que o desempenho da solução não seja um fator preponderante, pretendo apenas que corresponda ao que um possível cliente pretenda;
    - O segundo protótipo siga as boas práticas de engenharia e que o desempenho da solução seja a principal preocupação, mantendo exatamente as mesmas funcionalidades que o primeiro protótipo.

3.1.4 Seleção de ideias

Maior parte das vezes num negócio não existe dificuldade em se encontrarem novas ideias. O problema com que as organizações se deparam é na seleção de quais as ideias que lhes são mais vantajosas no momento em que se encontram e dentro dessas com quais devem prosseguir o seu investimento. Para as ajudar, existem processos iterativos de seleção de ideias, no entanto nenhum é infalível. Um dos métodos de decisão passíveis de ser utilizado e no qual se foca esta dissertação é a análise hierárquica.

3.1.4.1 Análise hierárquica

Análise hierárquica ou AHP é um método de decisão multicritério. Os métodos de apoio à decisão multicritério permitem a priorização de alternativas numa situação de critérios conflituosos, procurando satisfazer as restrições com uma solução de compromisso. Assim, estes métodos fornecem apoio não só à negociação, mas também à decisão em grupo (Buchanan and Gardiner, 2003).

O método AHP permite o uso de critérios qualitativos e quantitativos no processo de avaliação de ideias. Este tem como base dividir o problema de decisão em vários níveis hierárquicos para que haja uma compreensão fácil e posterior avaliação com o menor número de falhas possível.

Desta forma, o objetivo principal desta dissertação é analisar e otimizar o desempenho de uma aplicação *web*, tendo como critérios a avaliar os seguintes:

1. Complexidade do algoritmo: Avaliar se a solução é de fácil resolução ou com maior grau de complexidade para se alcançar o objetivo pretendido;
2. Eficácia: Avaliar se a solução implementada cumpre os princípios do desempenho de uma aplicação e as boas práticas gerais da engenharia;
3. Eficiência: Avaliar de que forma se alcançou uma solução que cumpre os princípios do desempenho de uma aplicação e as boas práticas gerais da engenharia;
4. Tempo: Avaliar o tempo que é despendido a alcançar a solução, pois uma solução pode demorar menos tempo, mas não cumprir qualquer critério de qualidade.

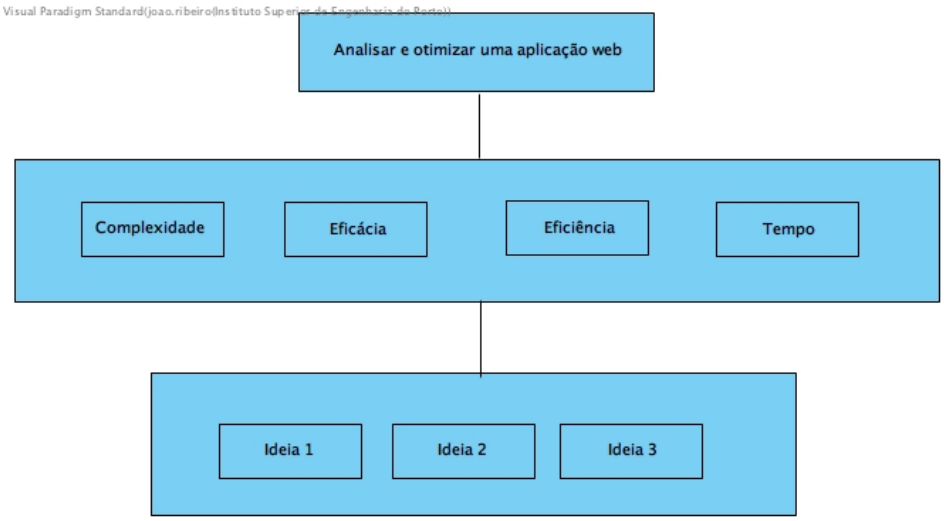

Figura 8 - Árvore hierárquica de decisão.

Com a árvore hierárquica formada é preciso passar a etapa seguinte, em que se avalia os critérios com recurso a uma escala com um conjunto de níveis de importância para comparar os diversos critérios, ou seja, com estes valores percebe-se a importância de um critério em relação a outro. A Tabela 2 mostra a escala de valores.

Tabela 2 - Escala fundamental - Níveis de importância de comparações (Saaty, 1990).

Nível de importância	**Definição**	**Explicação**
1	Igual importância.	As duas atividades contribuem igualmente para o objetivo.
3	Fraca importância.	A experiência e o julgamento favorecem levemente uma atividade em relação à outra.
5	Forte importância.	A experiência e o julgamento favorecem fortemente uma atividade em relação à outra.
7	Muito forte importância.	Uma atividade é muito fortemente favorecida em relação à outra.
9	Importância absoluta.	A evidência favorece uma atividade em relação a outra com o mais alto grau de certeza.
2, 4, 6, 8	Valores intermediários.	Quando se procura uma condição de compromisso entre duas definições.

Com base nesta escala e após definidos os critérios é possível então formar uma tabela de avaliação AHP para se perceber qual a melhor solução para atingir o objetivo principal. Na Tabela 3 são incluídos os pesos dados a cada critério segundo a escala mencionada.

Tabela 3 - Tabela de avaliação AHP.

Critérios de avaliação	**Complexidade**	**Eficácia**	**Eficiência**	**Tempo**
Complexidade	1	1/2	3	7
Eficácia	2	1	7	5
Eficiência	1/3	1/7	1	2
Tempo	1/7	1/5	1/2	1
Total	3 10/21	1 59/70	11 1/2	15

Com a Tabela 3 fica-se a perceber que a eficácia e a complexidade são os critérios que têm maior importância na tomada de decisão sobre qual das três ideias deve ser implementada. Posteriormente, a eficiência e o tempo apesar de ser importantes, não tem tanto impacto. O passo seguinte neste método é a normalização desta tabela, onde se deve utilizar cada valor introduzido e dividi-lo pelo total da coluna respetiva.

Tabela 4 - Tabela normalizada.

Critérios de avaliação	**Complexidade**	**Eficácia**	**Eficiência**	**Tempo**
Complexidade	0,2877	0,2713	0,2609	0,4667
Eficácia	0,5753	0,5426	0,6087	0,3333

Critérios de avaliação	Complexidade	Eficácia	Eficiência	Tempo
Eficiência	0,0959	0,0775	0,0870	0,1333
Tempo	0,0411	0,1085	0,0435	0,0667
Total (aproximado)	1,0000	1,0000	1,0000	1,0000

Posto isto, deve-se obter o vetor de prioridade, que tem por objetivo identificar a ordem de importância de cada critério. Alcança-se esse valor calculando a média aritmética dos valores de cada linha da matriz.

Tabela 5 - Critérios com prioridade relativa.

Critérios de avaliação	Complexidade	Eficácia	Eficiência	Tempo	Prioridade Relativa
Complexidade	0,2877	0,2713	0,2609	0,4667	0,3217
Eficácia	0,5753	0,5426	0,6087	0,3333	0,5150
Eficiência	0,0959	0,0775	0,0870	0,1333	0,0984
Tempo	0,0411	0,1085	0,0435	0,0667	0,0650

Analisando a Tabela 5 e olhando para as ideias geradas percebe-se que a eficácia, critério mais importante, estará mais presente na terceira ideia, indo de acordo com o objetivo principal desta dissertação. A complexidade será maior do ponto de vista da realização das restantes ideias, no entanto, estas não trazem tanto valor visto existirem alternativas de grande qualidade no mercado.

3.1.5 Desenvolvimento do conceito

Desenvolver dois protótipos em que um deles siga boas práticas de engenharia, como reutilização de componente, padrões de arquitetura, entre outros. Tem também de estar de acordo com as diretrizes de desempenho de uma aplicação. O outro protótipo não se deve

reger por estes princípios, não havendo tanto cuidado com a maneira como se alcança a solução, importando apenas que esta seja alcançada.

Se estes dois protótipos forem realizados com estas regras em mente será possível compreender melhor quais os cuidados que devem ser tidos em conta logo no momento em que se inicia um projeto para que este não venha a ter problemas de desempenho numa fase posterior, em que a complexidade tanto do código como do projeto em si já é maior.

3.2 Valor para o Cliente e Valor Percecionado

Qualquer atividade comercial é intrinsecamente sobre troca de valor. Trata-se de entregar algum bem ou serviço tangível e intangível, tendo sido aceite o seu valor. O valor tem sido definido em diferentes contextos teóricos como necessidade, desejo, interesse, critérios, crenças, atitudes e preferências (Nicola, Ferreira and Ferreira, 2012).

No contexto desta dissertação, o principal valor é a obtenção de conhecimento de como uma aplicação deve ser desenvolvida para que não tenha problemas de desempenho ao longo do seu ciclo de vida. Desta forma, previne-se que a cada incremento de funcionalidades não haja uma detioração da qualidade do *software* e que os clientes nunca tenham de procurar novas alternativas para o que pretendem alcançar.

3.2.1 Valor para o Cliente

Apesar de já mencionado na Secção 1.1, nunca é demais realçar que qualquer utilizador não gosta de esperar muito tempo para que uma página *web* esteja em condições para ser navegada e para lhe mostrar a informação que pretende.

Um estudo efetuado pela Google descobriu que cinquenta e três por cento dos utilizadores que acedem a uma página *web* através dos seus dispositivos móveis tendem a sair da página caso esta não esteja funcional em três segundos. Além disto, a média de carregamento de uma página numa conexão 3G demora em média dezanove segundos, enquanto que numa conexão 4G, esta desce para os catorze segundos (Kirkpatrick, 2016).

No entanto, esta empresa não foi a única a investigar este assunto. Um estudo realizado pelo grupo *Aberdeen* mostra o impacto que cada segundo tem no abandono que existe por parte dos utilizadores.

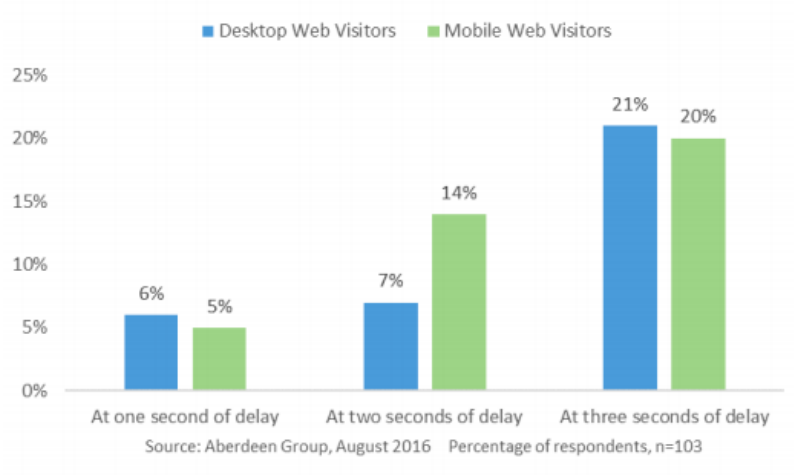

Figura 9 - Momento em que os utilizadores abandonam um *website* (Arsenault, 2016).

Com isto, pode-se efetivamente perceber que apesar de três segundos não parecerem nada no mundo real, no mundo virtual trata-se de um tempo considerável de espera (Arsenault, 2016).

Através desta dissertação torna-se possível dar aos utilizadores e à comunidade *open source* ferramentas úteis para que as suas aplicações *web* consigam lidar com uma complexidade algorítmica elevada e com uma quantidade de dados substancial de forma fácil e rápida.

3.2.2 Valor Percecionado

O valor percecionado do consumidor tem sido apontado como o mais importante indicador para que um cliente pretenda voltar a comprar algum bem material ou não material, junto da empresa. Dito isto, com o reconhecimento da importância deste valor pelo consumidor, vem o reconhecimento de que os retalhistas devem entregar o valor que aumentará a intenção de compra dos consumidores, criando e entregando boas experiências de compra (Morar, 2013).

De forma a que seja mais percetível este valor, existe uma análise de benefícios e sacrifícios que deve ser feita. Esta análise tem por base uma perspetiva longitudinal dividida em quatro elementos.

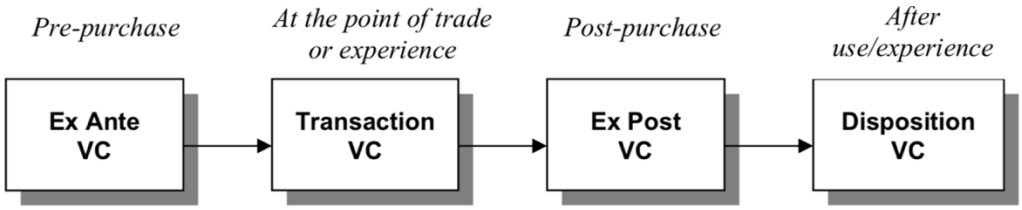

Figura 10 - Perspetiva longitudinal (Woodall, 2003).

Com isto em mente pode-se definir quais os benefícios e sacrifícios inerentes a cada elemento desta perspetiva.

Tabela 6 - Benefícios e sacrifícios.

	Benefícios	**Sacrifícios**
Antes da compra		Custo de investigação.
Transação		Custo de desenvolvimento.
Após a compra	Melhoria do conhecimento para projetos futuros.	Custo de infraestrutura; Custo de manutenção; Custos de divulgação.
Após uso	Melhoria no desempenho da aplicação *web*; Reter utilizadores; Poupar custos de infraestrutura desnecessária.	Custo de infraestrutura; Custo de manutenção.

Analisando a Tabela 6 pode-se perceber que existem diversos tipos de custos, de investigação para que no momento de se implementar uma solução esta seja o menos custosa possível, tendo por fim também impacto nos custos de manutenção e da infraestrutura que suporta. Por outro lado, existem os benefícios de reter maior número de utilizadores e principalmente melhorar o desempenho da aplicação *web*, dando uma melhor experiência de utilização ao cliente.

3.3 Proposta de Valor

A proposta de valor desta dissertação é a de criar dois protótipos que sejam capazes de mostrar as diferenças de desempenho entre um protótipo que tenha como objetivo estar funcional para que o cliente possa utilizar e outro protótipo que siga as diretrizes de desempenho de uma aplicação *web* e as boas práticas de engenharia, sendo algumas destas mencionadas na Secção 2.2.

Com este contraste de protótipos e avaliação dos mesmos é possível ficar a perceber o impacto que existe na escolha da forma como desenvolver os protótipos. Este impacto tanto pode ser no consumo de recurso como de memória ou processador, como também pode ser no tempo que demora a que um utilizador consiga realizar as tarefas que pretende.

3.4 Modelo de Negócio Canvas

Um modelo de negócios descreve a lógica de como uma organização cria, entrega e captura valor. Acredita-se que um modelo de negócios pode ser melhor descrito através de nove blocos básicos que mostram a lógica de como uma empresa pretende ganhar dinheiro. Os nove blocos cobrem as quatro principais áreas de um negócio: clientes, oferta, infraestrutura e viabilidade financeira. O modelo de negócios é como uma base para que uma estratégia seja implementada por meio de estruturas, processos e sistemas organizacionais (Osterwalder, Pigneur and Smith, 2010).

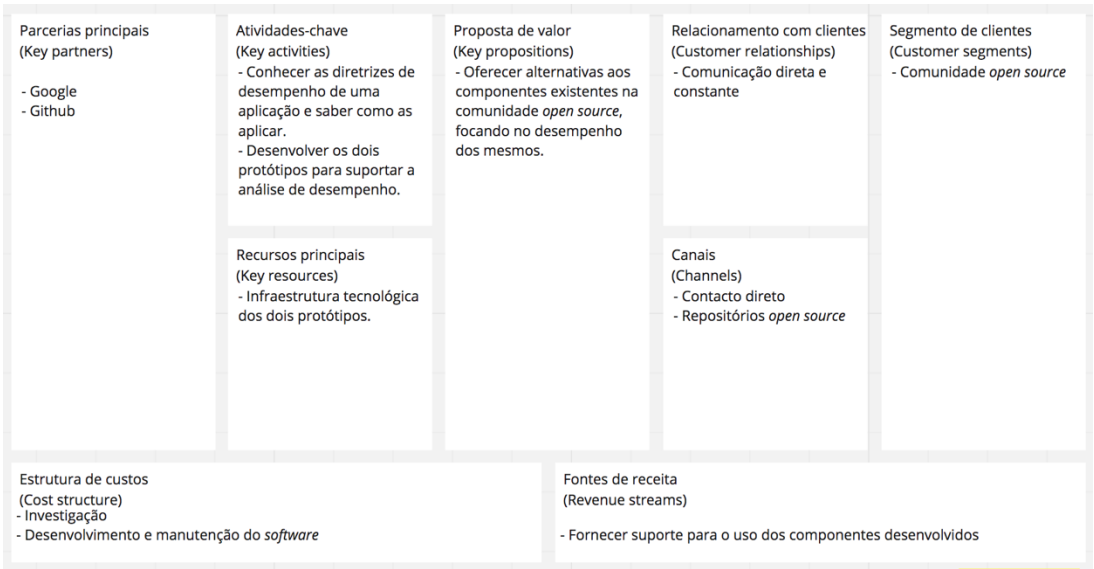

Figura 11 - Modelo de negócio Canvas.

Através da Figura 11 é possível perceber que existe uma proposta de valor bem definida, a qual é a de oferecer alternativas aos componentes existentes na comunidade *open source*, tendo em maior consideração o desempenho dos mesmos. Com isto, pode ser obtida receita dando suporte às empresas que pretenderem ajuda na compreensão e utilização dos componentes desenvolvidos.

A nível de custos, existe o desenvolvimento e manutenção dos componentes, pois é necessário eliminar o maior número de dependências externas possíveis em termos de *software*. Em termos de parcerias, a possibilidade de publicar os componentes no Github permite dar maior visibilidade ao trabalho realizado e angariar um maior número de aderentes à utilização dos componentes desenvolvidos.

3.5 Rede de Valor

A análise de rede de valor é uma metodologia de modelagem de negócios que visualiza atividades de negócios e conjuntos de relações a partir de uma perspetiva dinâmica de todo o sistema. Este inclui várias abordagens de análise exclusivas e também se integra a outras ferramentas de modelagem, como ferramentas de processo, ferramentas de análise de redes sociais e dinâmicas de sistemas (Allee, 2006).

Quanto a esta dissertação, existe a pretensão de disponibilizar para a comunidade *open source* os dois protótipos realizados, permitindo especialmente que o protótipo otimizado seja reutilizado por toda a comunidade, bem como possam ser sugeridas eventuais melhorias ou acrescento de funcionalidades que sejam vistos como úteis. Assim, deve existir uma relação de confiança e entre ajuda no que toca aos membros desta comunidade.

3.6 Sumário

Com este capítulo é possível compreender o propósito da realização desta dissertação e o que ela acrescenta ao estado atual do mercado tecnológico. Todo o processo de geração e seleção de ideias permite que se alcance uma ideia base bastante sólida e com espaço para crescer não só de complexidade, mas também de valor. Criar dois protótipos diferentes permite que se tire conclusões sobre os cuidados a ter no momento de criar componentes novos.

Com o estudo sobre o valor para o cliente percebe-se a importância que se deve dar aos detalhes no visionamento e na implementação da solução, abrindo portas a que se retenham estes clientes e que possam ser chamados novos clientes, melhorando as perspetivas de um negócio.

Delineando um modelo de negócio Canvas consegue-se perceber onde está a receita que se pretende realizar, mas também identificar logo os pontos onde se tem de investir mais, ou seja, com maiores despesas. Desta forma, tem-se uma perspetiva não só emocional, mas também racional do que o negócio pode ser e de como pode fruir.

4 Análise e *Design* da Solução

Neste capítulo pretende-se, primeiramente, realizar uma análise dos componentes necessários para a solução e os casos de uso que permitem colmatar. De seguida, pretende-se apresentar a arquitetura do protótipo que se quer conceber de forma otimizada, mostrando igualmente alternativas.

4.1 Análise

Esta secção é composta por uma divisão entre uma análise de componentes a implementar e uma enumeração dos casos de uso que estes componentes vêm ajudar a resolver. existe um diagrama de casos de uso que ajuda o leitor a compreender as funcionalidades da aplicação.

4.1.1 Análise de componentes

Nesta subsecção faz-se um estudo sobre cada componente e o seu efeito e propósito numa página *web*.

4.1.1.1 Componente *Icon*

Um ícone pode ser um dos principais elementos que torna uma marca reconhecível por todo o globo. Estes podem surtir efeito quando usados para melhorar o interesse visual e captar a atenção de um utilizador. Além disto, podem também dar a entender ao utilizador que possuem uma funcionalidade, dando de volta *feedback* importante (Williams, 2019). Com isto, um ícone,

por exemplo, do carrinho apresenta-se como importante para dar informação fácil e rápida ao utilizador.

4.1.1.2 Componente *Button*

Em qualquer página *web* existente deve-se apelar a que o utilizador faça a ação de *click*, de forma a conhecer novas partes da página e ir de encontro ao que pretende. Exemplos muitos comuns destes componentes são os que apelam a que se faça um registo gratuito na página *web*, ou que se entre num determinado artigo com promoção. Uma página pode conter vários botões e cada um desenhado ao seu estilo, com ações consideradas de maior relevo, igualmente denominadas primárias. Existem também os *buttons* de menor relevo, denominados secundários, com estilização diferente e não tão chamativa (Sailing, 2012).

4.1.1.3 Componente *Dropdown*

Escolher entre usar ou não o componente *dropdown* pode ser difícil. Este componente permite agregar listas com bastante informação que não necessita de estar a ser mostrada ao utilizador a todo o momento. Exemplo deste componente é a seleção do país numa página *web*. Como o utilizador não necessita de estar a ver a informação dos países a todo o momento, é oportuno esconder estas opções dentro de um componente que o permita (Holst, 2018).

4.1.1.4 Componente *Input*

O componente *Input* permite introduzir texto que o utilizador ache oportuno. Quer seja texto para que a página *web* encontre resultados mais específicos, quer seja para introduzir informação que seja pertinente guardar na base de dados. É este o componente que lida com a grande maioria da interação entre o utilizador e o teclado (Appleseed, 2014).

4.1.1.5 Componente *Product Card*

O componente *Product Card* tem como objetivo ser visualmente apelativo e dar ao utilizador o máximo de informação possível num curto espaço da página *web*. Para que o produto se torne apelativo e que cative o utilizador a pesquisar mais sobre ele, deve possuir imagens em tamanho reduzido (Bohachenko, 2018).

4.1.1.6 Componente *Filter Card*

Sem as ferramentas ideais, encontrar o product que se pretende pode ser considerada uma tarefa impossível para o utilizador. Possuir um componente *Filter Card* com opções de filtragem comuns permite ao utilizador alcançar mais rápido os produtos que tem em mente, sendo mais rápido avançar para o pagamento dos mesmos. Assim, este componente com campos de

filtragem relacionados com os detalhes do produto tornam a experiência do utilizador mais apelativa (Baymard, 2019).

4.1.1.7 Componente *Header*

O componente *Header* é um dos mais importantes pela simples razão de que geralmente é o primeiro que o utilizador visualiza. Assim, é o que causa o primeiro impacto junto do utilizador e o que pode ditar a sua atração pela página *web* ou o repúdio, indo em busca de um concorrente mais apelativo. Além disso, estes devem conter as informações mais pertinentes como o local para onde se vai para a listagem de produtos ou para o carrinho de compras (English, 2014).

4.1.1.8 Componente *Pagination*

Desde do início das páginas *web* que existem duas opções para listar todos os produtos. A primeira trata-se da paginação, aliada ao componente *Pagination*, que permite ter a listagem segmentada, podendo uma aplicação *web* colocar os produtos que lhe interessam vender mais na primeira página e esconder os restantes. Outra opção, trata-se do carregamento infinito de produtos. Esta última permite o mesmo que a primeira, no entanto é mais fácil para o utilizador alcançar produtos que sejam possivelmente menos vantajosos de vender (Bustos, 2016).

4.1.2 Casos de uso

Os casos de uso resumem-se às ações que um utilizador pretende realizar na aplicação *web*. Desta forma e com base na Figura 12 pode-se delinear os seguintes casos de uso:

- Gerir as suas informações pessoais, tais como nome, morada, entre outros;
- Ver a página de listagem de produtos, por forma a ter todo o catálogo de produtos existente na aplicação *web*;
- Poder escolher um produto onde pretende focar a sua atenção;
- Poder filtrar o catálogo de produtos, por exemplo, por marca de produto;
- Exportar informação dos produtos de forma a ter consigo em qualquer momento a cor do produto ou outras peculiaridades que ache pertinentes;
- Ver a página de detalhe de um produto pretendido, com toda a informação relativa ao mesmo;
- Escolher o tamanho do produto na página de detalhe;

- Ver o historial de preço do produto, de forma a perceber se já esteve alguma vez mais barato ou não.

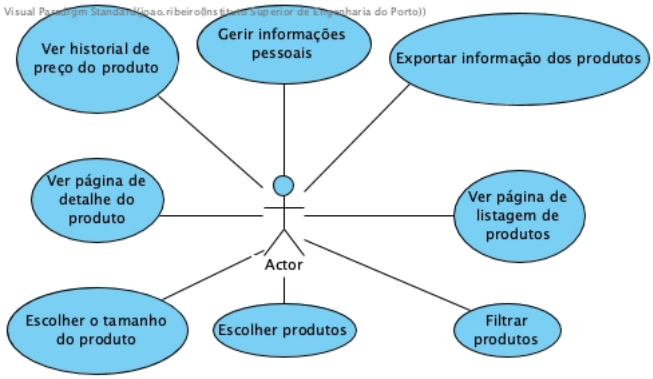

Figura 12 - Diagrama de casos de uso da aplicação.

4.2 Arquitetura de *software*

Nesta secção são enumerados os princípios de arquitetura a implementar na solução final e respetivas alternativas. Além disto, evoca-se o modelo arquitetural 4+1 para mostrar as várias vistas existentes dos protótipos que são implementados no Capítulo 5, sendo que a vista lógica e a vista de implementação se encontram nesta secção. As vistas de implantação e de processos estão localizadas nos Anexos B e C. Contudo, dá-se especial relevo ao protótipo que pretende seguir as boas práticas de engenharia.

4.2.1 Princípios de arquitetura

Nesta secção são apresentados dois conceitos que fazem parte das boas práticas de soluções arquiteturais, mas também na gestão de projetos de *software* como um todo. Assim, tem-se uma visão mais generalizada de todos os pontos que se devem ter em conta aquando do pensamento da arquitetura de uma aplicação.

4.2.1.1 *Design* atómico

Esta subsecção encontra-se separada em duas, sendo que a primeira demonstra a versão que foi efetivamente aplicada em relação ao *design* atómico, estando de seguida mencionada uma versão alternativa que não alcança todos os requisitos pretendidos.

4.2.1.1.1 Versão aplicada

Antigamente apenas existiam os computadores de secretária ou portáteis para navegar na Internet, o que tornava mais fácil adaptar uma aplicação *web* apenas a estes dois componentes.

Hoje em dia existem *tablets*, *smartphones*, televisões inteligentes, entre tantos outros dispositivos com diferentes tamanhos de ecrã e que proporcionam experiências de utilização diferentes. Com isto, cria-se uma tarefa extra de pensar numa aplicação *web* que seja compatível e que proporcione a mesma experiência de utilização em todos estes dispositivos. Assim, deve-se pensar não só numa perspetiva de programação, mas também em termos de visualização gráfica, partindo componentes com maiores responsabilidades em pedaços mais pequenos e mais fáceis de gerir. Deste pensamento surgiu o *design* atómico.

Muitas vezes relacionado com a biologia para uma fácil compreensão, o *design* atómico pretende que haja pequenos elementos, denominados átomos, que quando aliados a outros formem uma molécula. No momento em que uma molécula se agregar a outras, então irá dar origem a um organismo e assim sucessivamente como mostra a Figura 13.

Figura 13 - Componentes do *design* atómico (Costa, 2017).

Um átomo é a unidade básica na construção de um componente, sendo sempre o mais simples e com menos lógica dentro dele. De pouca utilidade servem estando isolados, pelo que são agregados de forma a criarem uma molécula. Exemplos de átomos são:

- Palete de cores;
- Tipografia;
- Ícones;
- Botões simples como uma caixa de seleção.

Uma molécula é um grupo de pelo menos dois átomos agregado que muitas vezes servem de base para sistemas de *design*. Estas moléculas devem juntar estritamente o número de átomos que necessitam para a sua funcionalidade estar completa, de forma a que haja uma otimização das mesmas e uma reutilização sempre que possível. Exemplos de moléculas são:

- Botões mais complexos;
- Componentes presentes num formulário;

- Ilustrações.

Com a agregação de pelo menos duas moléculas dá-se origem a um organismo. Estes organismos podem ter várias moléculas repetidas, alcançado já uma interface visual mais complexa. Exemplos de organismos são:

- Cabeçalho de uma página;
- Gráficos;
- Abas de uma página;

Com a agregação de dois ou mais organismos dá-se a origem de um *template*. Um *template* é um objeto no nível da página que coloca componentes num *layout* e articula a estrutura de conteúdo subjacente do *design* (Frost, 2016). Um exemplo é uma página *web* inicial. Por último, existem as páginas que são instância dos *templates* anteriormente definidos.

4.2.1.1.2 Versão alternativa

Nem sempre um princípio é aceite por todos na comunidade, pelo que acabam por surgir alternativas. Apesar do *design* atómico trazer uma separação de responsabilidades e de num primeiro momento parecer intuitivo, pode existir sempre um debate sobre se certos componentes são moléculas ou átomos. Um átomo pode ser um elemento simples de HTML, no entanto átomos como a cor ou tipografia podem não ser bem átomos na visão de certas pessoas pois estes incorporam sempre um elemento de HTML (Reimann, 2018). Todo este debate pode ser discutido e uma solução geral pode ser definida dentro de uma equipa. Contudo, uma alternativa pode ser a seguinte:

- Fundação – elementos como as cores, tipografia, espaçamentos ou ícones;
- Elementos – com uma definição similar aos átomos, onde são exemplos os botões de caixa de seleção ou outros elementos simples de HTML;
- Módulos – junção entre a definição de organismos e moléculas, onde não existe uma categorização restrita;
- Protótipo – este termo adequa-se aos momentos em que *Templates* definidos se conjugam com dados de forma a que se torne numa página que todos os utilizadores entendam.

4.2.1.2 Repositório com múltiplos componentes

Esta subsecção encontra-se separada em duas, sendo que a primeira demonstra a versão que foi efetivamente aplicada em relação à estrutura de repositórios e seus componentes, estando de seguida mencionada uma versão alternativa que não alcança todos os requisitos pretendidos.

4.2.1.2.1 Versão aplicada

Um repositório com múltiplos componentes (Scott, 2017) é um conceito nos dias de hoje cada vez mais comum, estando ligado à simplificação da organização de componentes, sendo mais fácil de coordenar alterações a todos os componentes de uma vez só. As dependências externas tornam-se mais fáceis de lidar e todos os processos ligado a um componente, como o de testes ou o de *deploy*, tornam-se mais fáceis de gerir. No entanto, a principal desvantagem é permitir o acesso por completo ao repositório a novos membros. Torna-se impossível de caso se pretenda que um novo utilizador, menos experiente por exemplo, tenha apenas acesso a um determinado projeto, ele poder ir modificar os restantes projetos sem que haja um controlo do mesmo.

4.2.1.2.2 Versão alternativa

A alternativa existente neste caso é a de criar múltiplos repositórios de forma a que haja uma separação de responsabilidades sobre quem pode modificar ou não um certo projeto. No entanto, existirá sempre uma maior quantidade de trabalho na manutenção dos mais diversos repositórios e que podem provocar a repetição de várias tarefas desnecessariamente.

4.2.2 Vista lógica

A vista lógica pode ser demonstrada através de um diagrama de componentes. O propósito destes diagramas é o de mostrar a relação que existe entre os vários componentes num sistema. Ao serem desenhados estes diagramas permite-se uma visão da estrutura física do sistema, tendo sempre em atenção os componentes do sistema e como eles se relacionam. Nesta subsecção demonstram-se duas possíveis versões da vista lógica do protótipo a implementar.

4.2.2.1 Versão aplicada

Na Figura 14 é possível compreender os componentes existentes na solução pensada. Inicialmente, existem dois componentes principais:

- Navegador, onde a interface gráfica da aplicação *web* se encontra e com a qual o utilizador irá interagir;
- Web App, o local onde se encontram três componentes, o motor de renderização da parte gráfica, a lógica de negócio e a base de dados;
    - Motor de renderização - trata de modificar a interface gráfica que irá ser mostrada no navegador, passando sempre a informação que vem dos restantes componentes;
    - Lógica de negócio - é um componente onde se encontram as regras pelas quais a aplicação *web* se pretende reger, ou seja, é neste componente que se decide que informação é passada tanto no sentido da base de dados como no sentido oposto;
    - Base de dados - é o componente onde os dados se encontram alojados e que serve de fonte de verdade para consultas de informações pertinentes.

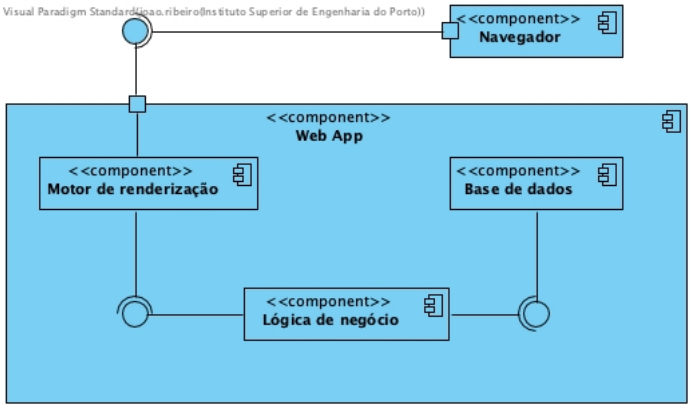

Figura 14 - Diagrama representativo da vista lógica.

4.2.2.2 Versão alternativa

Uma possível alternativa, no caso de se pretender ter em uma aplicação de baixo acoplamento e que permita ser escalável, é a da Figura 15. Esta alternativa permite que, no caso da aplicação *web* estar a receber muitos pedidos, dar resposta com a replicação da estrutura de *back-end* onde a lógica de negócio se encontra. Esta é uma visão comum em arquiteturas ligadas a micro serviços (Jackson, 2019), utilizando contentores e balanceadores de carga para minimizar o efeito de lentidão sentido pelo utilizador.

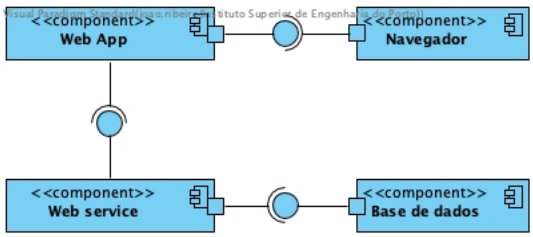

Figura 15 - Diagrama alternativo da vista lógica.

4.2.3 Vista de implementação

A vista de implementação descreve a organização estática do *software* num ambiente de desenvolvimento. Com isto em mente, esta vista deve ter em conta o princípio arquitetural do *design* atómico, detalhado na Subsecção 4.2.1.1. Assim, existem duas versões descritas nesta subsecção, a aplicada e a alternativa.

4.2.3.1 Versão aplicada

Com a Figura 16 percebe-se que existe um repositório que pode conter várias aplicações *web* funcionais. Uma aplicação *web* não é mais do que uma ou mais páginas reunidas. No caso de ser apenas uma página, é geralmente denominada de SPA (*Single Page Application*). Posto isto, uma página é composta por um ou mais Templates e assim sucessivamente até à unidade básica de construção de componentes que é o átomo.

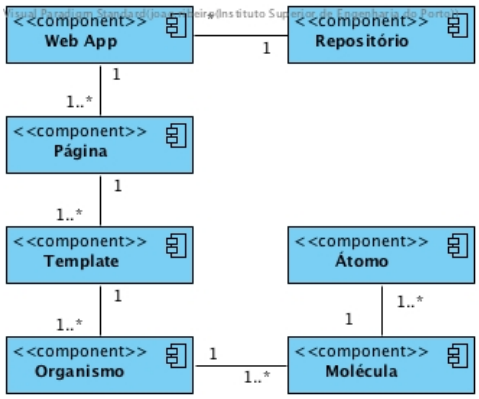

Figura 16 - Diagrama representativo da vista de implementação.

4.2.3.2 Versão alternativa

Como existem visões não concordantes com o *design* atómico, deve existir também uma alternativa à vista de implementação que tem por base esse conceito. Assim e aproveitando a alternativa mencionada na Subsecção 4.2.1.1. alcança-se a Figura 17. Um repositório continua

a ter várias aplicações *web*, no entanto uma aplicação *web* possui um ou mais protótipos. Um protótipo tem um ou mais módulos, pelo que o mesmo acontece daí em diante até à unidade básica denominada fundação.

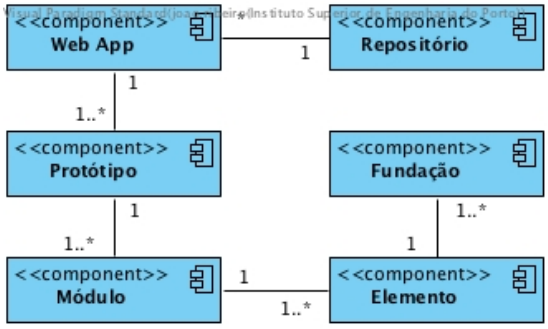

Figura 17 - Diagrama alternativo da vista de implementação.

4.3 Sumário

Com este capítulo existe uma explicação das razões de certos componentes serem escolhidos para alcançar a solução pensada. Por exemplo, para que se introduza dados escritos, o melhor componente é o *input*, ou para se agregar listas que não necessitem de estar sempre visíveis, o melhor é o componente *dropdown*.

Quanto aos princípios de arquitetura, optar por um repositório só faz com que o código esteja todo localizado num só ponto, tornando-se mais fácil de dar manutenção. Quanto ao *design* atómico, optar-se por uma versão com mais categorias permite distinguir melhor em que local cada componente deve ser integrado. Apesar de existir muitas vezes o debate sobre se um componente se trata de um átomo ou molécula, este deve ser tido e classificado. Assim, não se foge ao debate, como na versão alternativa.

Na vista lógica a versão aplicada faz sentido para aplicações pequenas que não necessitem de ser escaladas por possuírem bastante tráfego. Na vista de implementação, aplica-se o *design* atómico escolhido, de modo a que haja uma convergência de ideias.

5 Boas Práticas para Implementação

Neste capítulo pretende-se primeiramente demonstrar as boas práticas de programação dos componentes definidos no Capítulo 4. Além disto, referem-se os componentes React que foram criados para serem introduzidos nas páginas, igualmente aqui descritas.

5.1 Conceitos e cuidados

Nesta secção pretende-se mostrar os impactos que certas pequenas decisões, no momento de implementar código JavaScript, podem ter aquando de uma utilização mais pesada ou quando são postas à prova com testes de carga.

5.1.1 Ciclo de vida de um componente React

Um componente de React tem um ciclo de vida composto por algumas funções padrão que permite ao desenvolvedor aproveitá-las para tirar o máximo partido do componente em si. Um ciclo de vida é composto por vários momentos, sendo um deles em que o componente está a

ser montado na DOM, outro em que pode ou não ser atualizado e por fim, outro em que é removido da DOM. A Figura 18 mostra quais as funções que ocorrem em cada momento.

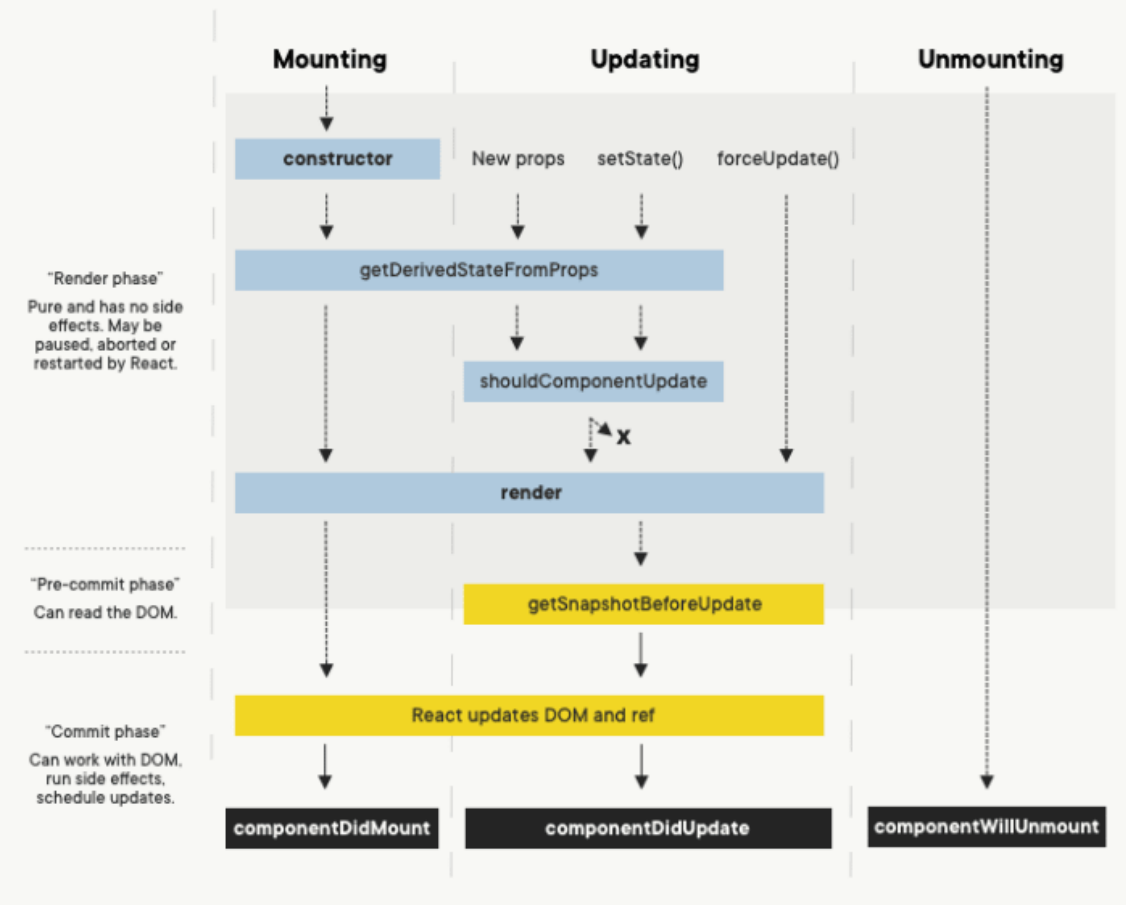

Figura 18 - Ciclo de vida de um componente (Boldare, 2019).

5.1.1.1 Função constructor

O método constructor é um método especial para criar e inicializar um objeto criado com uma classe. Só pode existir um método destes por cada classe. Pode-se introduzir a palavra chave super para chamar a função construtor da superclasse. Além disto, se se pretende que este componente possua estado, então é nesta função que este deve ser inicializado.

5.1.1.2 Função getDerivedStateFromProps

A função getDerivedStateFromProps é invocada após o construtor e antes da função *render*, neste caso tanto no momento em que se injeta o componente na DOM, como no momento em que se o atualiza. Deve retornar um objeto que atualize o estado do componente ou nulo, de forma a não existir qualquer atualização. Esta função existe para casos muito específicos, nenhum deles relacionado com otimização de uma aplicação, pelo que não existe necessidade de o estudar aprofundadamente.

5.1.1.3 Função render

A função render é a única que tem de ser obrigatoriamente declarada num componente React. Esta função examina as propriedades e o estado do componente, tendo como retorno outros elementos React, valores primitivos como booleanos ou números inteiros, outros valores não primitivos mais complexos, etc. Esta função não deve mudar o estado do componente nem interagir diretamente com o navegador. A função render não será invocada caso no momento de atualização do componente, a função shouldComponentUpdate retornar um booleano falso.

5.1.1.4 Função componentDidMount

A função componentDidMount é invocada imediatamente após o componente ser injetado na DOM, sendo o local propício a fazer pedidos aos servidores que possuem os dados de que o nosso componente necessita. É possível atualizar o estado do componente nesta função, pelo que o utilizador não irá reparar que houve uma atualização da aplicação. No entanto, isto pode trazer problemas de desempenho à aplicação, pelo que o recomendado é iniciar o estado sempre na função construtor, tal como referido e exemplificado na Subsecção 5.1.1.1.

5.1.1.5 Função shouldComponentUpdate

A função shouldComponentUpdate é a mais importante no que toca à otimização de uma aplicação, no entanto, é sempre preferível utilizar um componente puro, especificado na Subsecção 5.1.3, em vez desta função. Nesta função são comparadas as propriedades e estado atuais do componente antes da sua atualização com as propriedades e estado com que irá ficar aquando da sua atualização. Caso não haja diferenças entre todas estas, a função retorna um booleano falso e o componente não irá ser renderizado, poupando recursos ao computador do utilizador. A Figura 19 corrobora o que foi anteriormente dito. O componente C1 tem a sua função shouldComponenteUpdate a retornar um booleano verdadeiro, bem como a virtual DOM não é equivalente, pelo que tem de passar pelo processo de atualização completo. Já o componente C2 retorna um booleano falso na mesma função, pelo que tanto ele como os seus filhos não precisam de atualização, sendo nestes casos onde se melhora mais o desempenho da aplicação. Tendo os componentes C3 e C6 os mesmos resultados de C1, estes têm de sofrer atualização. O componente C7 encontra-se na mesma situação que C2. Por último, o componente C8 precisa de ser atualizado por causa da função de atualização ter retornado um booleano verdadeiro, no entanto esta atualização não é completa pois as DOM's virtuais são equivalentes.

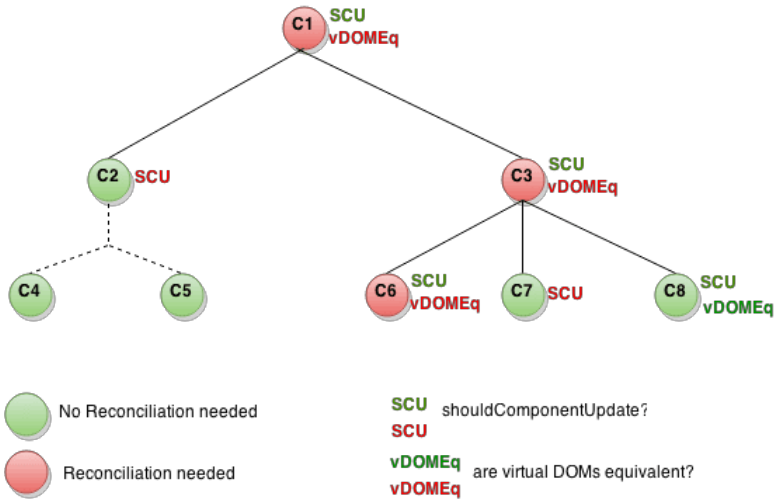

Figura 19 - Exemplo de árvore de componentes (Facebook, 2019).

5.1.1.6 Função getSnapshotBeforeUpdate

A função getSnapshotBeforeUpdate é invocada logo após a informação renderizada ser injetada na DOM. Permite capturar informação na DOM, como por exemplo a posição na página onde o utilizador se encontra. Isto é útil para aplicações que possuam um canal de comunicação, no entanto, nas aplicações realizadas isso não acontece, pelo que não se trata de uma função a ter em conta para este estudo.

5.1.1.7 Função componentDidUpdate

A função componentDidUpdate apenas é chamada após existir uma atualização do componente, ou seja, no arranque inicial do componente esta função nunca é chamada. Esta função também é útil para se fazer pedidos a serviços externos, comparando previamente se as propriedades antigas do componente são iguais às propriedades atuais dele. Um erro muito comum nesta função é o de tentar comparar propriedades objeto como um todo, fazendo com o que a linguagem JavaScript apenas compare as referências dos objetos e não o seu conteúdo. Isto pode levar a que o componente, e por consequência a aplicação, entre num ciclo infinito de atualizações. Além disto, a renderização extra do componente pode degradar o desempenho do componente.

5.1.1.8 Função componentWillUnmount

A função componentWillUnmount é invocada no momento antes do componente ser retirado da DOM. Este é o sítio ideal, por exemplo, para cancelar pedidos à rede ou para limpar dados que deixem de ser necessários no futuro. Quando não se faz a limpeza de forma correta, podem

ficar alguns objetos assignados de forma incorreta à aplicação, fazendo com que o *garbage collector* não atue corretamente. O *garbage collector* tem como objetivo limpar partes da memória que já não estejam a ser utilizadas, ou seja, se existir um ou mais objetos que estejam a ser erradamente referenciados, pode-se ter em mãos um problema comumente denominado de *memory leak*, tal como descrito na Subsecção 2.2.3.3.

5.1.2 Estado e propriedades de um componente

Um componente React pode ter propriedades e um estado, sendo ambos considerados objetos pela linguagem JavaScript. A diferença principal entre estes dois conceitos é de que as propriedades são passadas para dentro do componente, no entanto o estado é gerido dentro do próprio componente. São comparáveis, respetivamente, à passagem de parâmetros numa função e à declaração de variáveis numa função.

No caso de se querer modificar uma propriedade, esta pode receber o seu valor inicial através do componente pai, podendo este último modificá-lo. Uma propriedade pode ser passada para componentes filhos e ser igualmente modificada. A única situação que não é possível ocorrer a uma propriedade de um componente é esta ser modificada dentro do próprio componente, algo que no estado do componente já é possível. Por outro lado, o estado de um componente não pode ser modificado pelo componente pai nem pelo componente filho.

5.1.3 Componentes puros e imutabilidade

Um componente puro de React é bastante similar a um componente normal, sendo que a sua única diferença é a não existência da função shouldComponentUpdate descrita na Subsecção 5.1.1.5. Este tipo de componente deve ser utilizado quando é expectável que este possua propriedades e estado de estrutura simples, ou seja, não é recomendável que este tenha componentes filhos complexos, visto não existir atualização de propriedades nos ramos existentes a partir deste nó da DOM. Assim, todos os componentes filho devem ser igualmente puros.

Dados imutáveis são dados que não podem ser modificados após serem criados. Isto permite que o desenvolvimento de uma aplicação seja mais simples, tornando possível a memorização dos dados por parte do navegador, bem como facilitando as técnicas de deteção de mudanças.

5.1.4 Fragmentos

Fragmentos é um padrão em React que permite um componente retornar múltiplos elementos sem ter que os agrupar, habitualmente, num elemento HTML *div*. Com isto, retira-se um nó da árvore DOM, tendo os seus impactos sido descritos na Subsecção 2.2.2.4.

```
1   <React.Fragment>
2     <Filters
3       p={p}
4       query={query}
5       list={list}
6       filters={filters}
7       dispatch={this.props.dispatch} />
8       <List p={p} list={list} dispatch={this.props.dispatch} />
9   </React.Fragment>
```

Código 1 – Exemplo de código com fragmentos.

O excerto de Código 1 mostra um componente *List* e um componente *Filters*, ambos com várias propriedades a serem passadas. No React 15 estes componentes teriam de ser agregados por um elemento HTML *div* sem qualquer estilização, no entanto, com o React 16 este processo foi substituído pelo React.Fragment.

5.1.5 Memorização em cache

A memorização em cache é uma técnica que armazena o resultado de uma função que seja considerada muito pesada computacionalmente e que seja denominada de pura, onde não existem alterações à aplicação dentro dessa mesma função. Sempre que os parâmetros de entrada na função forem iguais aos da função memorizada em cache isso significa que o resultado será o mesmo, pelo que se dispensa a execução dessa mesma função e apenas se retorna o resultado em cache. Desta forma, poupa-se recursos de processamento e a aplicação torna-se mais rápida. Caso os parâmetros de entrada sejam diferentes, então o processo de memorização em cache irá decorrer de novo, removendo o anterior e assim sucessivamente. Existe a memorização em cache de funções pura que sejam, por exemplo, seletores para popular um componente como a *dropdown*, mas também existe, desde a versão 16.6 do React, a função memo, que pode ser utilizada quando são componentes funcionais e não baseados em classes. A função memo mostra-se então como uma equivalência à função shouldComponentUpdate no caso dos componentes funcionais.

```
1   const countries = [
2     {
3       text: 'pais1',
4       value: 'pais1',
5     },
6     {
7       text: 'pais2',
8       value: 'pais2',
9     },
10  ];
11  const selector = (arg) => arg.map((entry) => ({
12    text: entry.text,
13    value: entry.value,
14  }));
15  let val = _.memoize(selector);
```

Código 2 - Exemplo de memorização em cache.

No excerto de Código 2 recorre-se à biblioteca lodash, pois esta já possui a função memoize que se pretende utilizar para a memorização em cache. Neste caso, existe um seletor que percorre todos os elementos de um vetor, passando algumas das suas propriedades para o resultado final. Assim, no caso de usarmos a função val(countries), ou seja, passando o vetor countries como parâmetro, iremos fazer com que toda a função seja memorizada em cache num primeiro momento. Caso usemos novamente a função val exatamente com os mesmos parâmetros de entrada, o resultado final será o mesmo que da primeira vez, no entanto, não existe o processamento efetuado pela função map.

5.1.6 Virtualização de listas

Por vezes uma aplicação tem uma lista infindável de elementos, pelo que se se for renderizar todos estes elementos, mesmo que escondidos da visão do utilizador, isto irá provocar um aumento enorme de nós na DOM e com isso piorar o desempenho da aplicação. Assim, nasceu o conceito de virtualização de listas, que pretende renderizar apenas os elementos de uma lista que o utilizador consegue visualizar. À medida que o utilizador navega pela lista, os nós na DOM que deixam de ser visíveis para o utilizador são removidos, sendo de imediato colocados novos nós na DOM que representem os novos valores visíveis. Existem bibliotecas que permitem a utilização deste conceito de uma forma fácil, como por exemplo a react-window (Vaughn, 2019).

```
1    <List
2      style={{ outline: 'none' }}
3      tabIndex={-1}
4      height={this.getCalculatedOptionsHeight()}
5      width={width}
6      rowHeight={optionHeight}
7      rowCount={this.state.visibleOptions.length}
8      scrollToIndex={
9        this.state.highlighted === -1 ? 0 : this.state.highlighted
10     }
11     rowRenderer={this._renderOption} />
```

Código 3 - Exemplo de lista virtualizada.

No excerto de Código 3 é possível visualizar um componente *List* que possui várias propriedades, algumas delas responsáveis pelo tamanho visual do componente. Este componente é importado diretamente da biblioteca react-window, sendo que deixa ao critério do desenvolvedor escolher certos parâmetros. Por exemplo, a propriedade rowRenderer passa uma função renderOption que é responsável por renderizar todas as opções da lista. Neste caso, renderiza um componente *Button* por cada opção existente. Posteriormente, esta biblioteca faz a gestão de quais as opções a serem visualizadas e quais as que não devem estar presentes na DOM de forma a melhorar o desempenho da aplicação.

5.1.7 Desempenho de funções iterativas

Quando possuímos bastante informação agregada num vetor, muitas vezes precisa-se de iterar sobre ele por diversos motivos, filtrar alguns dos seus dados, modificar a sua estrutura, ou outras operações. Com isto, a linguagem JavaScript expandiu e criou várias formas de o fazer, tanto com a sua linguagem nativa como com pequenas bibliotecas utilitárias. Tendo todas estas opções, deve-se procurar a que é mais rápida para satisfazer as nossas pretensões, no entanto, também é de relevar que se deve escolher uma opção que permita manter o código legível e de fácil manutenção e adaptação. Nesta subsecção referem-se as várias possibilidades de iteração, sendo a sua comparação em termos de desempenho no Capítulo 6. Primeiramente, são testadas as funções em termos de tempo, mais propriamente o número de operações que conseguem realizar por segundo. Finalmente, são as testadas as funções em termos de ocupação de memória. As funções a ser testadas são as seguintes:

- For loop - executa um conjunto de ações em cada elemento do vetor, até uma certa condição não ser alcançada, sendo esta geralmente a comparação entre o índice iterador com o tamanho do vetor;
- For each - executa um conjunto de ações por cada elemento do vetor;
- Map - cria um novo vetor com os resultados das instruções implementadas para cada elemento do anterior vetor;
- Filter - cria um novo vetor com todos os elementos que passaram as validações implementas na função;
- Find - retorna o valor do primeiro elemento do vetor que satisfaz as validações implementadas na função;
- Reduce - realiza no vetor as instruções delineadas pelo desenvolvedor, retornando apenas um valor resultado. Esta função é alimentada por quatro parâmetros, o acumulador, o valor atual, o índice atual e o vetor original. O valor que irá ser retornado no final é o resultado da junção de cada iteração feita sob o vetor.

5.1.7.1 Biblioteca lodash

A biblioteca lodash (Sirois and Hall, 2019) é um conjunto de funções utilitárias que pretende facilitar a escrita de funções recorrentes numa aplicação. Para além das funções equivalentes às anteriormente descritas nesta subsecção, existem também funções, como por exemplo, a de comparação de valores de objetos extensos e com muitos níveis de uma forma simples, reduzindo o número de linhas de código de várias para apenas uma. Além disto, caso o objeto ganhe maiores proporções no futuro, não é necessário introduzir mais linhas de código, visto esta função lidar com os objetos na sua totalidade. Esta funcionalidade é deveras útil para a função mencionada na Subsecção 5.1.1.5.

5.1.8 Passar uma função como propriedade

Quando se possuí um componente com um objeto que tem, por exemplo, cinco propriedades e pretendemos passar para o seu componente filho alguma dessa informação, mas não toda, devemos de especificar quais as propriedades a passar, em vez de simplesmente enviar todo o objeto em si, por exemplo, através do operador *spread*. Pode-se criar um novo objeto contendo as propriedades que se pretendem passar ou então criar propriedades singulares no componente filho que recebam cada uma das propriedades do objeto do componente pai. Assim mantem-se os componentes o mais nivelados possíveis, retirando muita da

complexidade que iria aparecer no futuro em termos de manutenção e extensão dos componentes.

No entanto, nem sempre as propriedades a serem passadas são objetos simples, pelo que é possível passar uma função como propriedade do componente pai para o componente filho. De forma a otimizar os métodos criados num componente com estado, deve-se ligar os métodos através da função *bind*, ou construí-los recorrendo à forma de funções *arrow*, tendo estas últimas a particularidade de não possuírem o seu próprio contexto *this*, entre outras particularidades. Realizar estas operações no método *render*, explicado na Subsecção 5.1.1.3., não é boa prática, pois a cada renderização do componente uma nova instância da função irá ser criada. Para que este problema seja ultrapassado existem duas alternativas. Uma delas é realizar o *bind* do método no construtor, enunciado na Subsecção 5.1.1.1., do componente. Outra alternativa prende-se com a criação de uma função *arrow* à parte do método *render*.

```
1   class ManageUser extends Component {
2     constructor(props) {
3       super(props);
4
5       this.state = {
6         inputFirstNameText: '',
7       };
8      this._handleFirstNameChange = event => this.handleChange('inputFirstNameText', event.target.value);
9     }
10
11     render() {
12       return (
13         <div className={classNames(Styles.container)}>
14           <div className={Styles.label}>
15             <label htmlFor={"1"}>{'First Name'}</label>
16           </div>
17           <Input type="text" value={this.state.inputFirstNameText}
18             onChange={this._handleFirstNameChange} />
19         </div>
20       );
21     }
22
23     handleChange(key, value) {
24       this.setState({ [key]: value });
25     }
26   }
```

Código 4 – Exemplo de passagem de uma função como propriedade.

No excerto de Código 4 existe um componente denominado ManageUser, que serve como componente pai para toda a gestão das informações pessoais do utilizador. Dentro desse componente existe a função constructor sendo esta responsável por herdar possíveis propriedades de componentes hierarquicamente acima, mas também de inicializar o estado do

seu componente, que possui uma variável de tipo *string* vazia. Por último possui a já referida função que é executada apenas quando é despoletada uma atualização no componente *Input*.

De seguida, encontra-se a função render para renderizar o componente *Input*, mais alguma estilização da página. Neste componente são passadas propriedades como o tipo de *Input* que se pretende, se texto ou numérico, o valor que deve apresentar, neste caso passando o valor do estado do componente ManageUser e por último a função que gere a atualização dos componentes.

A função handleChange é despoletada no momento em que existe atividade no componente *Input*, despoletando então uma alteração do estado da variável inputFirstNameText no momento em que o utilizador se encontra a digitar caracteres.

5.1.9 Normalização do estado

Certamente todo o desenvolvedor alguma vez se deparou com estruturas de dados com várias camadas numa aplicação, criando ainda novas camadas sempre que se acrescenta uma nova funcionalidade à aplicação existente. O mais comum de acontecer nestes casos é o de criar uma estrutura com base num vetor, com múltiplos objetos dentro do mesmo, no entanto, também é possível criar um objeto que possua todos esses mesmos objetos no seu interior. Caso se opte por utilizar a estrutura com um vetor, para se atualizar um objeto dentro do mesmo é necessário percorrer todo o vetor, mesmo que o objeto que se pretenda atualizar esteja numa das primeiras posições, sendo esta operação custosa a partir do momento que o vetor possui milhares de posições.

```
1   const naoNormalizado = [
2       {
3         id: 'identificador1',
4         valor: 'valor1',
5         texto: 'texto1',
6       },
7       {
8         id: 'identificador2',
9         valor: 'valor2',
10          texto: 'texto2',
11        },
12    ];
```

Código 5 - Exemplo de estado não normalizado.

Caso se opte por seguir a estrutura com base num objeto, no momento de atualizar um objeto filho do mesmo, este irá ser referenciado diretamente pela sua chave, não necessitando de percorrer todo o objeto. A este último método chamamos-lhe normalização do estado.

```
1   const normalizado = {
2       'identificado1': {
3           id: 'identificador1',
4           valor: 'valor1',
5           texto: 'texto1',
6       },
7       'identificador2': {
8           id: 'identificador2',
9           valor: 'valor2',
10          texto: 'texto2',
11      },
12  };
```

Código 6 - Exemplo de estado normalizado.

5.2 Componentes

Nesta secção são abordados todos os componentes existentes na aplicação *web*. Todos estes componentes foram alvo de isolamento, utilizando um repositório externo onde estes se encontram, passíveis de serem reutilizados noutra aplicação. Com isto, seguem os princípios arquiteturais definidos na Subsecção 4.2.1. onde sendo componentes diferentes, encontram-se todos dentro do mesmo repositório. Outro princípio seguido, o do *design* atómico, é efetivamente aplicado, por exemplo no componente *dropdown*, em que este possui o componente *button*, que por sua vez reutiliza também o componente *icon*. Todo este desenvolvimento é suportado numa aplicação denominada *Storybook*, sendo esta a agregadora de todos os componentes criados e que permite em tempo real verificar o resultado de possíveis alterações efetuadas a um componente. Isto permite ter um isolamento da aplicação *web* final, o que torna mais fácil detetar inconsistências nas alterações efetuadas.

5.2.1 Icon

O componente *icon* tem por base algumas propriedades como o *svg*, o local de onde o objeto é lido e executado, mas também possui uma propriedade *title*, onde se pode referir o título do ícone em si e por último um *uniqueId*, onde se pode dar uma identificação única referente ao

ícone *svg* que se está a lidar. Este componente é utilizado dentro de outros componentes de seguida descritos.

5.2.2 Button

Grande parte das ações que um utilizador efetua numa aplicação *web* está relacionada com um componente *button*, quer seja para entrar na página de detalhe de um produto, quer seja para guardar as informações pessoais de um utilizador. Para este componente reaproveita-se o elemento de HTML <button>, utilizando estilização CSS por cima do mesmo. Além disto, este componente *button* permite que haja ícones dentro dele, dando uma informação mais enriquecida ao utilizador sobre as ações que pode tomar ao clicar no determinado botão.

5.2.3 Dropdown

O componente *dropdown* é bastante útil para agregar bastante informação junta num local que não precisa de estar sempre visível para o utilizador. Por exemplo, no caso de se querer utilizar este componente para se filtrar a lista de produtos por uma determinada marca, todas as opções encontram-se agregadas apenas neste componente, não ocupando visualmente mais espaço na página do que deveriam no momento em que este se encontra no seu estado fechado. Assim, quando o utilizador pretende filtrar uma marca, efetua um clique no componente *dropdown*, que possui um componente botão por trás, para abrir a *dropdown* e aí então poder selecionar a marca pretendida, sendo que todas as opções se encontram igualmente com o componente *button* por detrás. Para que haja uma informação visual para o utilizador, o componente *button* possui dentro dele um ícone a informar que a opção foi selecionada, levando posteriormente essa informação até ao componente *dropdown*. Além destes componentes, existe a possibilidade de habilitar a pesquisa de forma a que seja mais fácil de encontrar a opção pretendida, em vez de obrigar o utilizador a navegar por entre todas as opções. Ao habilitar esta pesquisa, aparece antes das opções um componente *input*, onde permite o utilizador digitar o que pretende.

5.2.4 Input

O componente *input* tem por base o elemento de HTML <input>, possuindo por cima estilização CSS para que tenha um aspeto visual mais apelativo ao utilizador. Além disto, ao digitarmos alguns caracteres dentro do mesmo, aparece no lado direito um componente *button*, com um componente *icon* dentro de si, a permitir que o utilizador possa apagar num clique apenas todo o texto que digitou.

5.2.5 Checkbox

O componente *checkbox*, alavancado pelo elemento de HTML <input> com o tipo *checkbox* e com estilização CSS por cima, permite ao utilizador, no componente *Filter Card*, selecionar qual o intervalo de preços que está disposto a desembolsar numa possível compra de um produto. Junto a este deve existir uma informação descritiva adicional, para que o utilizador perceba qual o intervalo de preços que está a selecionar. Para atingir este efeito introduz-se o elemento de HTML <label>.

5.2.6 Card

O componente *Card* não é nada mais que um conjunto de elementos de HTML estilizadas com CSS de forma a que haja algum relevo em relação à cor de fundo da página. Assim, o utilizador percebe as delimitações visuais deste componente, permitindo que haja um foque apenas no que se encontra nele.

5.2.7 Product Card

O componente *Product Card* reutiliza o componente *Card* anteriormente mencionado para que haja uma delimitação visual por cada produto listado. Além disto, reutiliza o componente *button* no canto inferior esquerdo para que o utilizador possa ir para a página de detalhe do produto, mas também possui outro *button* no canto inferior direito caso o utilizador pretenda adicionar o produto ao seu carrinho de compras. Por cima destes dois *buttons*, existem dois elementos de HTML <label> com a menção ao nome da marca e ao modelo do produto. No topo do *Card* existe uma fotografia do produto em si para que haja uma breve visualização do produto por parte do utilizador.

5.2.8 Filter Card

O componente *Filter Card* reutiliza o componente *Card* para que o utilizador sinta uma separação entre a secção de filtragem de produtos e a secção onde estes são listados. Além deste, também reutiliza o componente *dropdown* para que seja possível filtrar por marcas, o componente *checkbox* para que filtre pelo intervalo de preços pretendido e pela cor do produto. Todas as ações de filtragem têm de seguida impacto na secção de listagem de produtos, retirando os produtos, representados através de *Product Cards*, que não contemplam os requisitos do utilizador.

5.2.9 Header

O componente *header* pretende delinear quais as páginas existentes que o utilizador pode visualizar. Este componente encontra-se sempre presente em qualquer página que o utilizador se encontre, de forma a que este possa navegar pela aplicação *web* de uma forma mais fácil e rápida. Este componente é constituído por elementos de HTML estilizadas com CSS, contendo no seu interior *buttons* que fazem o roteamento direto entre páginas. Possui *buttons* para ir até à página de listagem, até ao carrinho de compras, aos contactos e à página de gestão de informações pessoais.

5.2.10 Pagination

O componente *pagination* encontra-se sempre no final da página de listagem de produtos, para que o utilizador possa navegar entre páginas de produtos existentes, até encontrar um ou mais que sejam do seu agrado. Este componente é composto por elementos de HTML com estilização CSS, sendo cada página existente composta por um componente *button*. Além destes, nas suas laterais existem *buttons*, com elementos de HTML <label> a seu lado, que permitem navegar para as páginas anteriores ou seguintes.

5.3 Páginas

As páginas de uma aplicação *web* não são mais que um conjunto de componentes, tendo esse mesmo nome seguindo a lógica do *design* atómico mencionado na Subsecção 4.2.1. Para que haja uma aplicação *web* que permita a um utilizador visualizar e comprar produtos, existindo uma série de páginas fundamentais, estando elas descritas nesta secção.

5.3.1 Página de listagem

A página de listagem é o local onde se apresenta ao utilizador todos os produtos que temos em loja, ou seja, toda a mercadoria que é passível de ser vendida. Esta página deve ter uma estilização agradável, estando apoiada nos componentes criados para o efeito. Nesta página existe o componente *header* na parte superior, tendo no fim da página o componente *pagination* para que o utilizador possa continuar a ver mais produtos passíveis de serem vendidos. No lado esquerdo encontra-se o componente *Filter Card*, para que a pesquisa seja mais enriquecida para o utilizador, de forma a que encontre os produtos que pretende mais rapidamente e incentivando a compra impulsiva. Por último, do lado direito, encontram-se todos os produtos existentes, com ou sem filtragem já aplicada, para que possam ser analisados e que permitam entrar na página de detalhe de um produto caso este tenha despertado a curiosidade do cliente.

5.3.2 Página de detalhe

Na página de detalhe encontra-se igualmente o componente *header* que permite uma rápida mudança de página. Para além deste, existe toda a informação existente em relação ao produto, bem como uma foto em tamanho maior em relação à que se encontra na página de listagem. Por último, existe a possibilidade de escolher o tamanho pretendido através de um componente *dropdown* e um *button* que permite adicionar o produto ao carrinho de compras.

5.3.3 Página de compra

A página de compra é onde o utilizador última todos os detalhes da sua compra. Tem um resumo dos produtos selecionados, bem como a informação onde a encomenda será entregue e o método de pagamento. Toda esta página possui o componente *header* no topo e

componentes *Card* com *inputs* no seu interior para que o utilizador introduza as informações de entrega e pagamento.

5.3.4 Página de gestão de informação

A página de gestão de informação tem como objetivo ser o ponto central em que o utilizador atualiza as suas informações pessoais, podendo alterar, por exemplo, a morada caso se mude para um local diferente do anterior. Possui o componente *header* no topo tal como as restantes páginas. Além desta, possui o componente *Card* com vários *inputs* para introdução do nome, morada e outras informações pertinentes. Tem também uma *dropdown* onde se deve selecionar qual o país em que reside. No fundo da página, existe um *button* para salvar todas as informações que foram alteradas.

5.4 Sumário

Com este capítulo pode-se entender quais os cuidados que se devem ter durante o ciclo de vida do desenvolvimento de uma aplicação *web*. Primeiramente, o ciclo de vida dos componentes React deve ser bem assimilado para que se possa tirar o maior aproveitamento possível do mesmo. Posteriormente, utilizar memorização em cache ou virtualização de listas permite igualmente melhorar o desempenho da aplicação em si. De seguida, são descritos os componentes e páginas implementados para responder aos casos de uso criados. Todos estes componentes e páginas regem-se pelos princípios arquiteturais do mono repositório e do *design* atómico, seguindo assim todas as boas práticas enumeradas neste capítulo.

6 Avaliação

Qualquer momento que é gasto a desenvolver uma aplicação, deve sempre ter em conta a sua manutenção no futuro, estando protegido por os mais variados tipos de testes. Para que se perceba se existe discrepância entre os dois protótipos devem ser realizadas experiências que o demonstrem. Além disto, deve-se também averiguar se o protótipo que segue as boas práticas de engenharia e as diretrizes de desempenho está efetivamente otimizado.

6.1 Testes

No caso de se realizar uma nova funcionalidade ou de se modificar alguma parte da aplicação, deve sempre existir uma quantidade de testes automatizados que permitam averiguar se a totalidade da aplicação se encontra funcional antes de ser enviada para um ambiente de produção. Inicialmente pode parecer contraproducente gastar tanto tempo a implementar os mais variados tipos de teste, no entanto existe um retorno no futuro quanto à rapidez com que se detetam falhas ou simplesmente no lançamento automatizado da aplicação. Nesta secção, explicitam-se testes unitários, funcionais e de segurança. Todos estes testes correm numa *pipeline* feita em *Jenkins* de forma a que o conceito de CD (*Continuous Delivery)* e CI (*Continuous Integration*) seja efetivamente seguido.

6.1.1 Testes unitários

A base do conjunto de testes de uma aplicação é composta por testes unitários. Estes testes garantem que uma determinada unidade, a que está a ser alvo do teste, da base de código da aplicação esteja a funcionar conforme o planeado. Os testes unitários têm o âmbito mais restritivo de todos os testes de uma aplicação. O número de testes unitários ultrapassa em larga escala qualquer outro tipo de testes que existem na aplicação (Vocke, 2018).

De forma a serem realizados os testes unitários é necessário adicionar bibliotecas à aplicação como é o caso do *Jest* (Jest, 2019) e do *Enzyme* (Airbnb, 2019), tendo este último uma configuração extra para que se adapte consoante a versão do React utilizada. De forma a que haja sempre uma exigência alta para com o desenvolvedor e de forma a que se proteja o código criado, deve existir uma regra de percentagem mínima de linhas de código cobertas por testes unitários. Assim, fazem-se testes à grande maioria dos cenários existentes na aplicação e previne-se que no futuro haja erros na aplicação desnecessários.

Posto isto, deve-se definir a estrutura dos testes que se pretendem realizar, de forma a dar uma sensação de uniformidade entre eles. Existe um padrão que possui duas denominações, sendo estas "*Arrange, Act, Assert*", baseada em TDD e "*Given, When, Then*", muito utilizada em BDD (*Behavior Driven Development*) (Nair, 2018). No fundo, o que se pretende é, numa fase inicial, configurar os dados de teste, para de seguida se chamar o método a ser testado dentro do teste, confirmando no final que os resultados expectáveis estão a ser efetivamente retornados.

6.1.2 Testes funcionais

O testes funcionais, também por vezes denominados de E2E (*End to end),* servem para automatizar a interação do utilizador com a aplicação *web*, de forma a que sempre que se efetuar alguma modificação no código desta aplicação, todo o fluxo da mesma é revisto, de forma a despistar possíveis erros que tenham sido criados nesse momento. Assim, é mais uma barreira de forma a que não haja o perigo de lançar uma aplicação para um ambiente de produção com problemas.

Para estes testes existem imensas ferramentas tais como *TestCafe* (TestCafe, 2019), *Puppeteer* (Google, 2019b) e *WebdriverIO* (OpenJS, 2019), sendo esta última a utilizada neste projeto por já existir familiaridade com a mesma. O *WebdriverIO* é totalmente desenvolvido em JavaScript,

pelo que os desenvolvedores que se sentem confortáveis a programar nesta linguagem têm uma curva de aprendizagem menor, aliado à documentação da página oficial. Para além disto, tem uma sintaxe simples, pelo que facilmente se pode pegar num determinado elemento através de seletores e posteriormente chamar um método em cima desse mesmo elemento. Para cada projeto é necessário um ficheiro de configuração do *WebdriverIO*, no entanto este é facilmente configurável com o apoio da documentação. Por último, é compatível com bibliotecas como *mocha* (Mocha, 2019) ou *jasmine* (Jasmine, 2019), que tornam a criação de testes mais simples.

6.1.3 Testes de segurança

Os testes de segurança são um tipo de testes ao *software* que têm como propósito descobrir vulnerabilidades no sistema e determinar se os recursos e dados do mesmo estão protegidos contra intrusos. Para que se perceba se a aplicação se encontra protegida recorre-se a uma ferramenta denominada Checkmarx. Esta ferramenta é uma SAST (*Static Application Security Tool*) (Koussa, 2018) e tem como intenção rastrear o código fonte de forma a identificar vulnerabilidades de segurança tais como SQL (*Stuctured Query Language*) *Injection* (Portswigger, 2019), XSS (*Cross Site Scripting*) (OWASP, 2018), entre outros. Esta ferramenta pode ser integrada no processo de construção e implantação da aplicação num servidor de produção, fornecendo no final dos seus testes um documento que permite perceber se existem ou não falhas de segurança e em que parte do código elas se encontram, de forma a serem rapidamente corrigidas.

6.2 Perfil da aplicação

Nesta secção são referidas inicialmente as grandezas e metodologias de avaliação a ter em conta nos dois protótipos criados. Posteriormente, são mostrados os resultados dos testes efetuados quanto ao tempo e à memória que cada um dos protótipos consome, percebendo-se assim o impacto que as boas práticas de desenvolvimento têm num produto final.

6.2.1 Grandezas e metodologia de avaliação

As grandezas que são consideradas como mais relevantes a utilizar na avaliação dos protótipos são o tempo e a memória alocada a cada protótipo em determinados momentos de utilização. Ao ser utilizado o tempo como métrica de avaliação permite-se perceber o espaço temporal que os dois protótipos demoram a responder e a se tornarem funcionais, tanto no momento inicial, como após o despoletar de uma ação por parte do utilizador. Principalmente para os momentos em que houver lentidão por parte dos protótipos, avaliar a memória alocada permite perceber se existe um determinado *memory leak* num dos protótipos, sendo este um grande causador da lentidão das aplicações *web*.

A metodologia de avaliação a utilizar é com base no Google Chrome Performance Tab e no React Profiler, já descritos nas Subsecções 2.2.3.3. e 2.2.3.4. respetivamente. Ambos os protótipos são submetidos ao mesmo percurso de traçagem de perfil da aplicação, de forma a que haja uma comparação fidedigna e com base em várias experiências semelhantes. Como existem sempre momentos dispares na alocação de memória, entre outros recursos, do computador, estas comparações têm por base resultados obtidos ao final de várias tentativas, estando elas mencionadas em cada subsecção com mais detalhe.

6.2.2 Medição do tempo

Para medir o tempo que o impacto das boas ou más práticas de desenvolvimento têm numa aplicação escolhem-se seis casos para serem estudados, cobrindo eles desde pormenores existentes na *framework* React até a instruções genéricas da linguagem JavaScript, permitindo assim uma análise mais abrangente.

6.2.2.1 Função shouldComponentUpdate

A função shouldComponentUpdate, explicada na Subsecção 5.1.1.5., pretende poupar recursos computacionais. Para que haja um exemplo concreto a ser testado, escolhe-se o caso de uso referente à gestão de informações pessoais, descrito na Subsecção 4.1.2., com a sua página mencionada na Secção 5.3. Esta página possui vários componentes diferentes, igualmente enunciados na Secção 5.2. A página possui sete *inputs* onde se podem introduzir informações como o primeiro e último nome, morada, e-mail, entre outros. Além destes, possui uma *dropdown* onde é possível escolher o país do utilizador e um *button* onde pode clicar para salvar as informações.

O componente da página possui então vários componentes filho, pelo que, se se seguir as boas práticas de programação em React, no caso de um componente filho estar a ser atualizado, com por exemplo o primeiro nome do utilizador, os restantes componentes não precisam de sofrer atualização pois os seus valores mantêm-se, poupando assim recursos. Efetua-se esta atualização de um componente filho cerca de duzentas vezes com esta boa prática implementada e outras duzentas vezes sem estar implementada.

Tabela 7 - Média de tempo do componente com e sem função shouldComponentUpdate.

	Média em milissegundos
Boa Prática	$2.5 \pm 1.09\%$
Má Prática	$5.5 \pm 1.22\%$

Com os resultados da Tabela 7 percebe-se a diferença de desempenho entre as duas opções, sendo a redução de tempo mais de metade. Para que se perceba de uma forma mais visual os componentes que são atualizados e os que não sofrem qualquer tipo de atualizações, utiliza-se o React Profiler. Na Figura 20 pode-se perceber todos os componentes existentes na página de gestão de informação do utilizador, sendo o componente *ManageUser* o que agrega todos os componentes de *input*, *buttons* e *dropdown*.

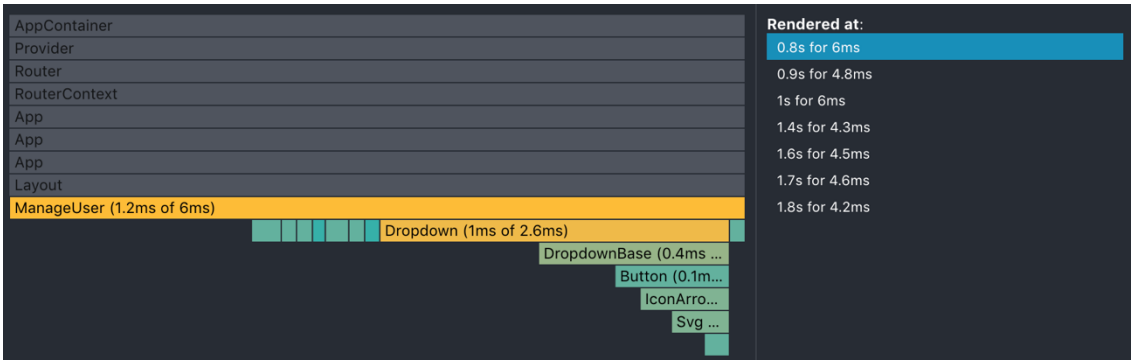

Figura 20 - Exemplo em que não se aplica a função componentShouldUpdate.

Com a Figura 20 confirma-se que foi utilizado o conceito de *design* atómico mencionado na Subsecção 4.2.1., pois dentro da *dropdown* existe o componente *button*. Este encontra-se igualmente representado por uma barra verde após a barra amarela representativa da *dropdown*, sendo este último botão o já mencionado botão de salvar a informação do utilizador.

O componente *ManageUser* e seus componentes filhos, na primeira renderização, demoraram um total de seis milissegundos, sendo que o componente pai apenas é responsável por cerca de um milissegundo desses seis. Na parte lateral direita da Figura 20 menciona-se o número de vezes que o componente pai foi renderizado, ou seja, o utilizador pressionou sete teclas, efetuando-se sete renderizações de todos os componentes. Tal como explicado anteriormente, na primeira renderização o componente *ManageUser* e seus filhos demoraram seis milissegundos a renderizar, aquando do Profiler estar a ser executado no primeiro segundo.

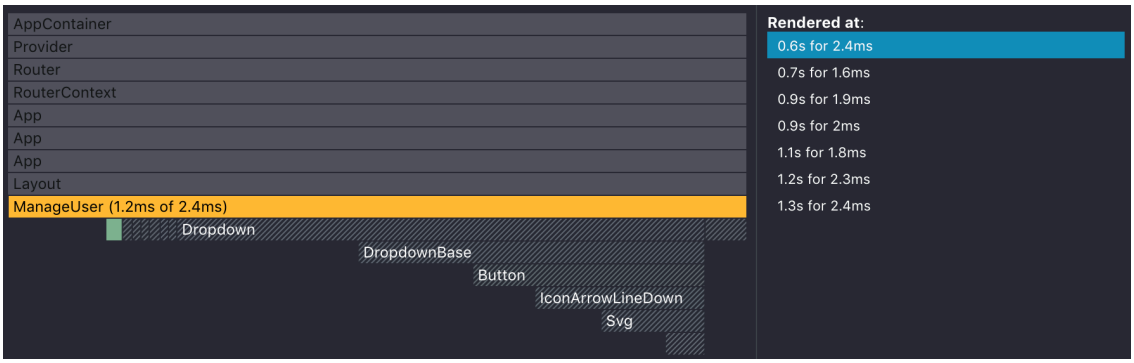

Figura 21 - Exemplo em que se aplica a função shouldComponentUpdate.

Na Figura 21 existem diferenças pelo facto de estar a ser aplicada a função shouldComponentUpdate junto dos componentes *button*, *input* e *dropdown*. Existe desde logo uma diminuição no valor de milissegundos em que o componente pai é executado, que é consequência de todos os seus componentes filho, à exceção daquele em que o utilizador se encontra a digitar igualmente sete valores, não sofrerem de atualização, não consumindo recursos computacionais.

Com toda esta informação conclui-se que utilizar a função shouldComponentUpdate é preponderante na otimização de uma aplicação *web* em termos de tempo, pois existe, em média, um decréscimo de três milissegundos, tal como mostra a Tabela 7.

6.2.2.2 Memorização em cache

A função memo do React é equivalente à função shouldComponentUpdate, no entanto a primeira serve para componentes funcionais, enquanto que a segunda serve para componentes de classe. Assim, a medição de tempo e conclusão descritas na Subsecção 6.2.2.1. aplicam-se igualmente à função memo. No entanto, além destas funções, existe a possibilidade de memorizar uma função pura que não contenha componentes React. Por exemplo, existindo uma *dropdown* com dez mil opções, que sejam sempre as mesmas dez mil opções, existe a

possibilidade de realizar uma memorização na função que mapeia as opções para o componente *dropdown*. Desta forma, testa-se essa função com e sem memorização, repetindo o processo mil vezes e culminando na Tabela 8.

Tabela 8 - Média de tempo da função com e sem memorização.

	Média em milissegundos
Com Memorização	$0.107 \pm 0.89\%$
Sem Memorização	$2.799 \pm 1.01\%$

Com os resultados da Tabela 8 e forçando sempre para que o *garbage collector* não seja executado a meio das operações, percebe-se que o ganho é bastante acentuado aquando do uso com memorização.

6.2.2.3 Virtualização de listas

Tendo a explicação por detrás do conceito de virtualização de listas sido descrito na Subsecção 5.1.6. resta medir em termos temporais se adotar esta solução é mais vantajosa ou não. De forma a que haja uma comparação fidedigna e aproveitando a página de listagem de produtos, desenvolvida para satisfazer os casos de uso referentes a ver uma listagem de produtos e a poder filtrá-los, foi desenvolvida uma secção de filtragem nessa página que detém um componente *dropdown*. Este componente pretende satisfazer o caso em que o utilizador quer filtrar os produtos por determinadas marcas. Como existem imensas marcas no mundo, acaba por ser passível de ser ter uma *dropdown* com, por exemplo, cinco mil marcas. Tendo este valor em conta, cria-se um processo de teste com quatrocentas repetições, sendo que em metade se aplica a virtualização de listas e na outra metade não.

Tabela 9 - Média de tempo do componente com e sem virtualização.

	Média em milissegundos
Com Virtualização	$30.9 \pm 1.43\%$
Sem Virtualização	$343.2 \pm 1.89\%$

Com os dados da Tabela 9 entende-se que a virtualização faz toda a diferença no momento de executar o componente *dropdown* com bastantes opções no seu interior. Com recurso ao React Profiler podemos perceber, através da Figura 22, que este componente, quando se encontra com listas virtualizadas, se torna efetivamente bem mais rápido.

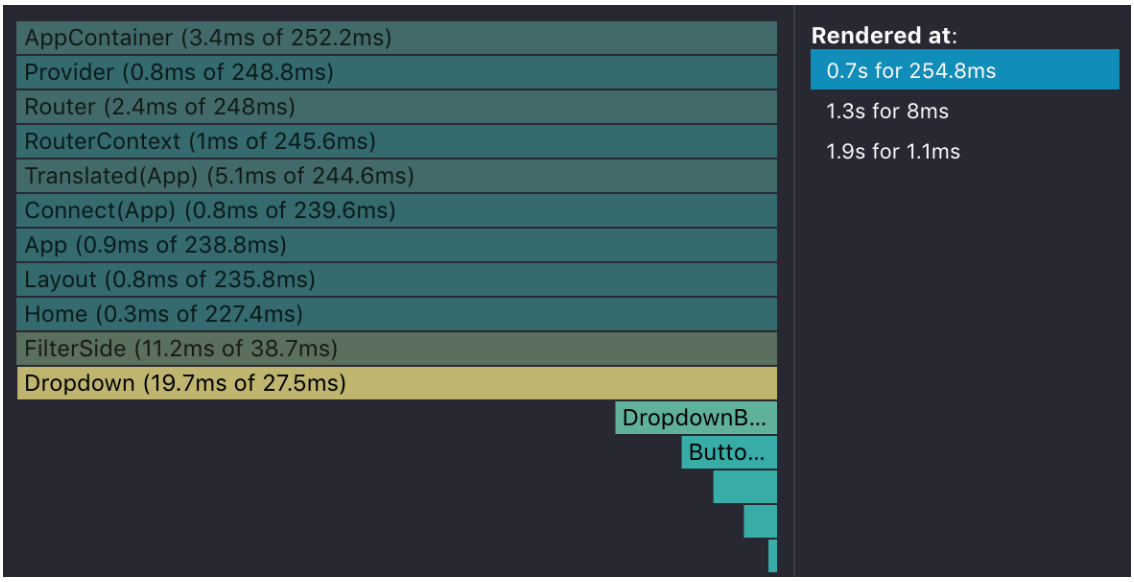

Figura 22 - Exemplo em que se aplica a virtualização de listas.

6.2.2.4 Desempenho de funções iterativas

Tendo por base um modelo de objeto semelhante para todos os casos, devem ser gerados dados em pequena, média e grande escala. Aliado a isto, encontram-se as várias funções a ser comparadas com instruções equivalentes, para que haja um máximo de semelhança possível e de forma a obter os resultados mais fidedignos possíveis. Como o desenvolvedor não se deve cingir apenas a executar a comparação uma vez para cada vetor diferente, executa-se um número considerável de vezes, neste caso definido como mil, o algoritmo de comparação de funções, recebendo por fim um valor relativo ao número de execuções por segundo.

Inicialmente, opta-se por comparar as funções estudadas com um vetor de cinco posições e assim obtém-se os resultados descritos na Tabela 10. Com os dados da Tabela 10 é possível depreender que o método mais rápido é a função *reduce*, seguida pela função *for each* e *filter*.

Tabela 10 - Resultados para um vetor com cinco posições.

Função	Operações por segundo
for each	708,427 ± 0.73%
reduce	795,244 ± 0.91%
map	653,102 ± 0.55%
Filter	680,770 ± 0.92%
for	632,653 ± 0.66%

Como nem sempre um vetor tem apenas estas posições ocupadas, deve-se sempre fazer o mesmo teste, mas com mais algumas posições, com vista a se perceber se os resultados se mantêm iguais ou se existe uma mudança no paradigma da função mais rápida a efetuar as instruções. Assim, utiliza-se um vetor com quinhentas posições e alcança-se os resultados apresentados na Tabela 11.

Tabela 11 - Resultados para um vetor com quinhentas posições.

Função	Operações por segundo
for each	7,580 ± 1.03%
reduce	7,639 ± 0.80%
map	7,564 ± 1.22%
filter	7,411 ± 0.91%
for	6,794 ± 0.98%

Com os resultados da Tabela 11 é possível depreender que a função *reduce* continua a ser a mais rápida de todas, no entanto as funções *for each, map e filter* encontram-se todas muito perto umas das outras, estando bem no final a função *for*.

Por último, de forma a se perceber se a tendência encontrada na Tabela 11 se mantém, gera-se um vetor com cinco mil posições, ou seja, uma estrutura já pesada e que não é muito comum encontrar em páginas *web*. Após o algoritmo ser executado, os resultados encontram-se enumerados na Tabela 12.

Tabela 12 - Resultados para um vetor com cinco mil posições.

Função	Operações por segundo
for each	$727 \pm 1.45\%$
reduce	$735 \pm 1.33\%$
map	$714 \pm 1.62\%$
filter	$702 \pm 1.80\%$
for	$663 \pm 1.55\%$

Analisando os resultados da Tabela 12, percebe-se que a função *reduce* continua a ser a mais rápida de todas, pelo que a função *for each* e *map* encontram-se nos lugares seguintes respetivamente.

Todos estes resultados devem ter sempre em conta a máquina onde estes testes são corridos, bem como o momento em que o *garbage collector* pode estar a atuar, libertando memória que numa outra opção estaria alocada e que fez com que a função demorasse mais tempo a ser concluída. Com isto pode-se entender que independentemente do tamanho do vetor, existem funções com mais capacidade de processamento que outras. Porém, os ganhos entre funções podem ser mínimos no desempenho da aplicação, pelo que se deve também ter sempre em conta a legibilidade do código para que a manutenção e acréscimo de funcionalidades no futuro seja menos penosa.

6.2.2.5 Assignar e propagar objetos

Sempre que é necessário copiar um objeto existente e adicionar-lhe novas propriedades existem duas formas mais comuns de o fazer. Pode-se assignar, com a ajuda da função *Object.assign* ou então propagar o objeto, com a ajuda do operador *spread*. De forma a que se

perceba qual a forma mais rápida de o sistema lidar com esta operação, formulam-se cinco casos, sendo todos executado no mínimo mil vezes, tal como na Subsecção 6.2.2.6.

O primeiro caso enumera que o objeto inicial e o objeto com as propriedades que se pretendem adicionar ao primeiro são inicializados fora da função de assignar objetos, sendo posteriormente copiados para um terceiro objeto.

```
1    const init = { a: 'a', b: 'b' };
2    const mid = { c: 'c' };
3    const end = Object.assign(init, mid);
```

Código 7 - Exemplo do caso um.

No segundo caso, o objeto com as propriedades extra é apenas declarado dentro da função *Object.assign*, não havendo assim a inicialização da variável fora da função, com o resto do processo igual ao primeiro caso.

```
1    const init = { a: 'a', b: 'b' };
2    const end = Object.assign(init, { c: 'c' });
```

Código 8 - Exemplo do caso dois.

No terceiro caso, declara-se as duas primeiras variáveis da mesma forma que no primeiro caso, no entanto, a terceira variável recorre ao operador *spread* para unificar uma cópia das outras duas dentro de si.

```
1    const init = { a: 'a', b: 'b' };
2    const mid = { c: 'c' };
3    const end = { …init, …mid};
```

Código 9 - Exemplo do caso três.

No quarto caso, a primeira variável é inicializada e uma cópia dela, através do operador *spread* é introduzida na variável final, sendo no final adicionadas as propriedades extra que se pretende.

```
1    const init = { a: 'a', b: 'b' };
2    const end = {…init, { c: 'c' }};
```

Código 10 - Exemplo do caso quatro.

Por último, no quinto caso, a primeira variável é inicializada e uma cópia dela, através do operador *spread* é introduzida na variável final somente após esta já possuir as propriedades extra pretendidas.

```
1    const init = { a: 'a', b: 'b' };
2    const end = {{ c: 'c' }, ...init };
```

Código 11 - Exemplo do caso cinco.

Tabela 13 - Assignar e propagar objetos.

	Operações por segundo
Caso 1	24,182,400 ± 1.43%
Caso 2	19,485,027 ± 1.12%
Caso 3	22,233,564 ± 1.63%
Caso 4	55,915,104 ± 0.93%
Caso 5	19,854,956 ± 0.88%

Com os dados da Tabela 13, pode-se concluir que o caso quatro é claramente o favorito para se ter uma aplicação mais otimizada. Os casos um e três são as alternativas seguintes com melhor desempenho, no entanto conseguem apenas cerca de metade das operações por segundo que o caso quatro mostra.

6.2.2.6 Desempenho de condições

Em todas as aplicações existe um momento em que se encontram condições, sendo o objetivo delas que num caso mostrem certos resultados e noutros casos mostrem resultados completamente diferentes ao utilizador. O que não é muitas vezes tido em conta é a forma em como estas clausulas são criadas, escolhendo geralmente o desenvolvedor a que se sente mais familiarizado. Contudo, para se perceber se existe impacto nessa decisão, realiza-se um estudo com os três casos mais comuns possíveis, executando esses casos igualmente mil vezes para se obter uma média de resultados fidedigna.

No primeiro caso realiza-se uma condição *if* com um retorno booleano no seu interior e caso não entre nessa condição irá ser retornado um outro valor booleano oposto.

```
1    const test = 'test';
2    if (test === 'test1') {
3        return true;
4    }
5    return false;
```

Código 12 - Exemplo do caso um.

No segundo caso, realiza-se exatamente a mesma condição, no entanto caso não entre nessa condição, irá entrar numa condição *else*, tendo o mesmo retorno que no primeiro caso.

```
1    const test = 'test';
2    if (test === 'test1') {
3        return true;
4    } else {
5        return false;
6    }
```

Código 13 - Exemplo do caso dois.

No terceiro caso, não existe condição *if* nem *else*, sendo feita imediatamente a comparação junto do retorno do valor booleano.

```
1    const test = 'test';
2    return test === 'test1';
```

Código 14 - Exemplo do caso três.

Tabela 14 - Desempenho de condições.

	Operações por segundo
Caso 1	15,829,983 ± 0.99%
Caso 2	15,782,926 ± 1.23%
Caso 3	15,798,855 ± 1.31%

Com os dados da Tabela 14, apesar de os três casos estarem bem próximos em termos de operações por segundo, o primeiro caso mostra-se como o mais rápido. Este estudo é bastante útil para os momentos em que se pretende perceber se se deve renderizar certos componentes

visuais de React. Caso não seja necessário renderizar esses componentes, torna-se o processamento da aplicação mais rápido e reduz-se o número de nós na DOM.

6.2.3 Medição da memória

Porque nem tudo se baseia apenas no tempo que se pode poupar, existe também a necessidade de medir a memória que é despendida em certas tarefas, de forma a que se perceba se esta está correlacionada com o tempo ou se existem disparidades, tendo por último o desenvolvedor que decidir qual o recurso mais importante para ele. Nesta subsecção faz-se a medição dos casos que anteriormente foram sujeitos a medição de tempo.

6.2.3.1 Função shouldComponentUpdate

A medição efetuada na Subsecção 6.2.2.1. conclui que se se introduzir a função shouldComponentUpdate esta faz com que os componentes não sejam tantas vezes atualizados e acaba por poupar tempo. Aproveitando exatamente o mesmo caso de estudo com o mesmo número de réplicas, obtém-se os dados da Tabela 15, que representa em média um intervalo de memória ocupada, em Megabytes, por cada uma das opções tomadas.

Tabela 15 - Média de intervalo de memória da aplicação com e sem função shouldComponentUpdate.

	Média de intervalo em megabytes
Boa Prática	$47.9 \pm 2.29\% - 51.2 \pm 3.11\%$
Má Prática	$59.7 \pm 4.22\% - 64.8 \pm 2.11\%$

Com os resultados da Tabela 15 percebe-se que existe uma discrepância de valores, em que por se efetivamente se seguir uma boa prática, que demora menos tempo e que renderiza menos vezes os componentes filho, o intervalo de memória alocada à aplicação será igualmente menor, existindo assim uma correlação nesta função em específico.

As medições de memória são efetuadas com recurso à Performance Tab do Google Chrome, tendo como resultado gráficos representativos do espaço em memória que ocupa a aplicação *web*. A Figura 23 mostra um exemplo de um caso onde a função shouldComponentUpdate não é aplicada.

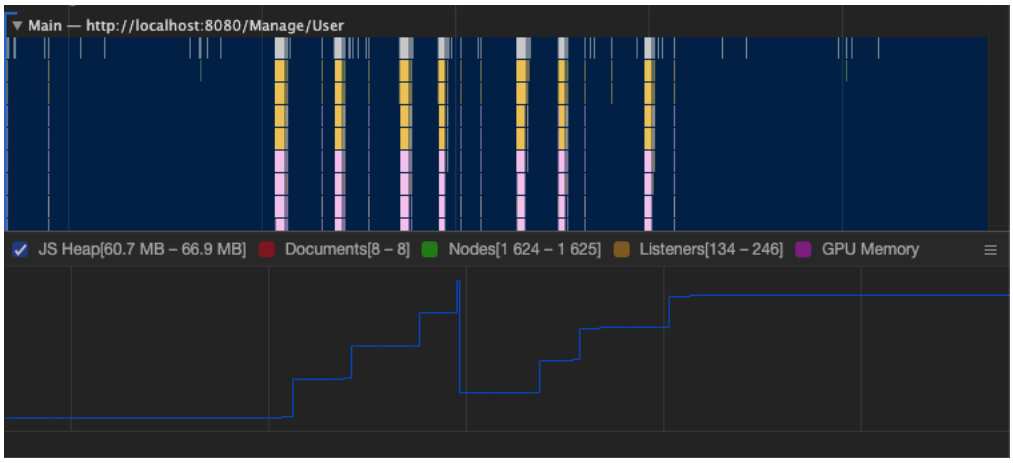

Figura 23 - Exemplo de memória alocada em aplicação sem função shouldComponentUpdate.

Na Figura 23 observa-se uma alocação de memória num intervalo entre os 60.7 e os 66.9 Megabytes, estando um gráfico onde mostra o seu crescimento gradual até que, num certo ponto, o *garbage collector* da linguagem JavaScript entra em ação, voltando a reduzir a memória alocada. Na Figura 24, já com a função a ser introduzida na aplicação *web* pode-se visualizar que existe um crescimento gradual, mas que nunca é necessário que o *garbage collector* entre em ação, estando sempre a memória alocada entre o intervalo de 47.3 a 50.4 Megabytes.

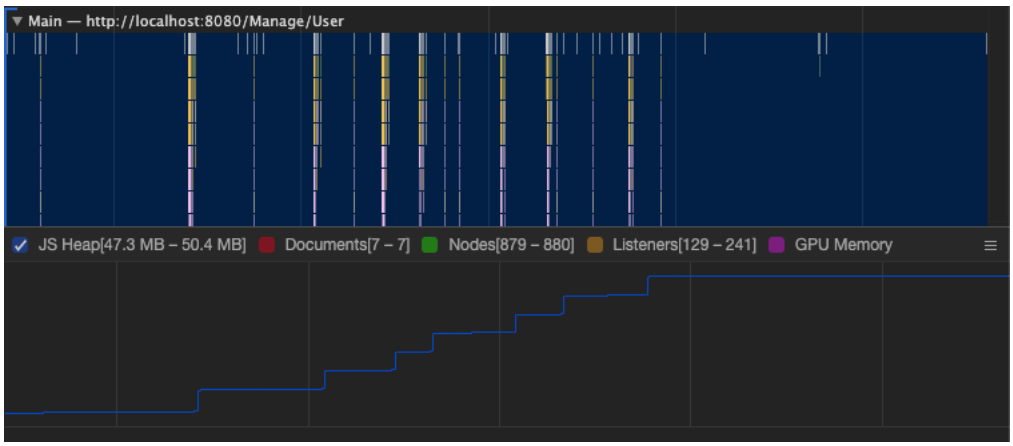

Figura 24 - Exemplo de memória alocada em aplicação com função shouldComponentUpdate.

6.2.3.2 Memorização em cache

Tal como descrito na Subsecção 6.2.2.2. existe uma equivalência entre a função memo e a função shouldComponentUpdate. Assim, em termos de medição de memória as duas são equivalentes, o que torna a função memo a melhor opção aquando da criação de componentes funcionais que possam ser memorizados. Além disto, também é estudado nessa mesma secção o impacto que tem uma função pura ser memorizada ou não em cache, neste caso um seletor de opções para uma *dropdown*. Assim, efetuam-se os testes nos mesmos moldes que na medição de tempo de modo a que haja concordância nos resultados.

Tabela 16 - Média de memória com e sem memorização.

	Média em kilobytes
Com Memorização	$10824.97 \pm 2.43\%$
Sem Memorização	$8719.23 \pm 3.21\%$

Com os resultados da Tabela 16 percebe-se que mesmo guardando a função em memória, o facto de esta não executar mais operações e simplesmente retornar o resultado, faz com que a média de memória gasta seja inferior.

6.2.3.3 Virtualização de listas

Para se perceber se a virtualização de listas é no seu todo uma aposta com ganhos, deve-se medir em termos de memória se esta ocupa menos espaço que a não inclusão da mesma. Dessa forma segue-se o processo de testes exatamente igual ao que foi seguido na Subsecção 6.2.2.3.

Tabela 17 - Média de intervalo de memória da aplicação com e sem virtualização.

	Média de intervalo em megabytes
Com Virtualização	$36.7 \pm 1.99\% – 54.3 \pm 2.33\%$
Sem Virtualização	$46.3 \pm 2.11\% – 79.5 \pm 2.41\%$

Com a Tabela 17 existe a confirmação de que não só é mais vantajoso, em termos de tempo, aplicar a virtualização em listas, como também em termos de memória existe uma poupança

acentuada. O intervalo existente mostra que sem virtualização, o valor encontra-se bem próximo do valor máximo do intervalo com virtualização.

Muita desta melhoria deve-se igualmente à quantidade de nós na DOM que não são renderizados, bem como ao número de componente à escuta de serem ativados, comumente denominados de *listeners*. A Figura 25 é um exemplo de uma situação em que a virtualização é aplicada e por isso o número de nós é mais reduzido.

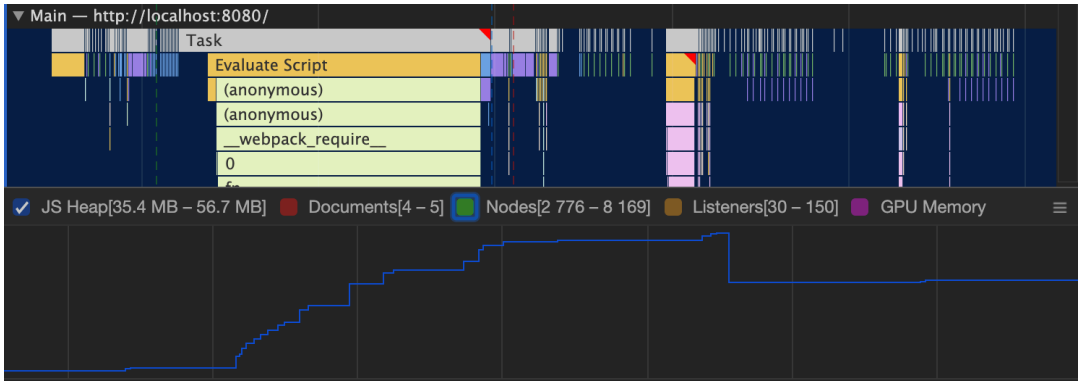

Figura 25 - Exemplo de memória alocada em aplicação com virtualização.

No caso em que não se aplica a virtualização, o número de nós tende a ser maior, bem como o número de *listeners*, pelo que o espaço em memória ocupado acaba por ser igualmente maior. Tal é verificado no exemplo da Figura 26.

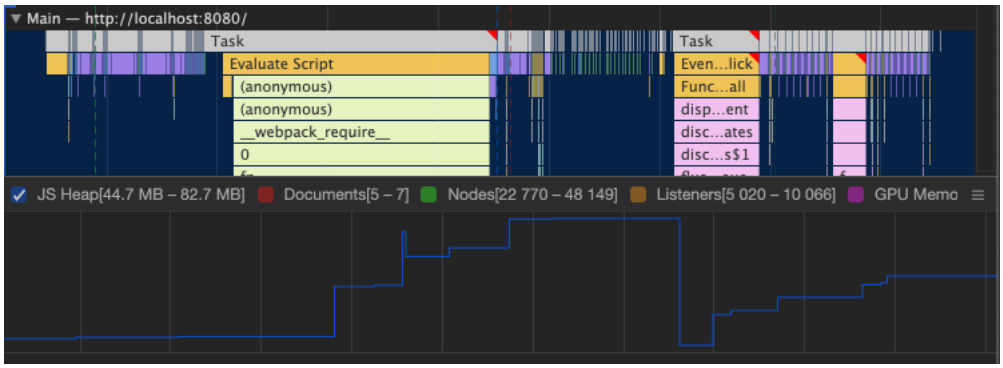

Figura 26 - Exemplo de memória alocada em aplicação sem virtualização.

6.2.3.4 Desempenho de funções iterativas

Seguindo exatamente o mesmo processo de teste enunciado na Subsecção 6.2.2.4., onde é medido o tempo das funções interativas, começa-se por realizar a medição de memória para um vetor com apenas cinco posições.

Tabela 18 - Resultados de memória ocupada para um vetor com cinco posições.

Função	Média em kilobytes
for each	4066.18 ± 2.44%
reduce	4315.60 ± 1.13%
map	4317.52 ± 1.22%
filter	4291.55 ± 0.79%
for	4291.73 ± 1.08%

Com os resultados da Tabela 18 percebe-se que a diferença não é acentuada entre as quatro das cinco funções, pelo que a diferença de tempos é mais notória. A função que se destaca pela positiva é a *for each*. De seguida, aplica-se o mesmo processo de testes, mas com um vetor maior, de quinhentas posições, a fim de se perceber se este padrão se mantém.

Tabela 19 - Resultados de memória ocupada para um vetor com quinhentas posições.

Função	Média em kilobytes
for each	4104.71 ± 2.59%
reduce	4106.33 ± 1.44%
map	4115.98 ± 1.65%
filter	4093.34 ± 1.14%
for	4043.92 ± 2.04%

Após análise dos resultados da Tabela 19 compreende-se que a função *for each* deixa de ser a que ocupa menos espaço, sendo ultrapassada principalmente pela função *for*. No entanto, percebe-se que a diferença continua a não ser tão significativa como a que existe na medição

de tempos. Para que se perceba se realmente esta tendência ocorre independentemente do tamanho do vetor, faz-se um último teste com um vetor de cinco mil posições.

Tabela 20 - Resultados de memória ocupada para um vetor com cinco mil posições.

Função	Média em kilobytes
for each	$4506.97 \pm 1.22\%$
reduce	$4504.24 \pm 1.59\%$
map	$4584.63 \pm 1.39\%$
filter	$4401.44 \pm 0.99\%$
for	$4092.96 \pm 1.03\%$

Na Tabela 20 existe uma confirmação evidente de que a função *for* é a que ocupa menos espaço em memória. De seguida, encontra-se a função *filter*, pelo que as restantes encontram-se mais próximas umas das outras. No entanto, existe alguma degradação na função *map.*

6.2.3.5 Assignar e propagar objetos

Para medir a memória utilizada no momento de assignar ou propagar objetos utiliza-se os mesmos cinco casos que são utilizados na Subsecção 6.2.2.5., sendo estes percorridos igualmente mil vezes até se obter uma média de memória ocupada.

Tabela 21 - Memória utilizada a assignar e propagar objetos.

	Média em kilobytes
Caso 1	$4062.06 \pm 0.43\%$
Caso 2	$4062.23 \pm 0.65\%$
Caso 3	$4063.28 \pm 0.44\%$

	Média em kilobytes
Caso 4	4063.13 ± 0.77%
Caso 5	4062.40 ± 0.85%

Com os resultados da Tabela 21 percebe-se que quando se aplica a função *Object.assign* existe uma ligeira melhoria em termos de ocupação de memória, no entanto, esta é quase insignificativa, por exemplo em relação ao melhor caso onde se aplica o operador *spread*, ou seja, o caso cinco. Desta forma, percebe-se que este estudo traz mais benefícios em termos de tempo do que necessariamente em memória.

6.2.3.6 Desempenho de condições

Para medir a memória utilizada nas condições segue-se o mesmo processo de teste que é efetuado na Subsecção 6.2.2.6., em que se executa o algoritmo mil vezes e se obtém uma média de memória ocupada por cada um dos três casos.

Tabela 22 - Memória utilizada na definição de condições.

	Média em kilobytes
Caso 1	4061.56 ± 0.10%
Caso 2	4061.55 ± 0.05%
Caso 3	4061.55 ± 0.05%

Com os resultados da Tabela 22 alcança-se pela primeira vez valores muito próximos de idênticos em todos os casos, mostrando assim que não existe qualquer tipo de impacto na escolha a tomar, o que não é o caso quando se tem em conta o tempo, como mostrado na Subsecção 6.2.2.6.

6.3 Sumário

Neste capítulo existe uma menção a boas práticas de desenvolvimento de *software* como a TDD, onde o desenvolvimento é orientado aos testes anteriormente definidos. Junto desta definição existe a enumeração dos tipos de testes aplicados a esta aplicação *web*, como unitários, funcionais ou de segurança. Posteriormente, para que se avalie as decisões a tomar no momento de se desenvolver uma aplicação *web* otimizada, são feitas medições em termos de tempo despendido e memória ocupada. Apesar de em alguns casos a diferença ser irrisória, noutros casos como na aplicação da função shouldComponentUpdate, memorização em cache e na virtualização de listas, percebem-se os enormes ganhos ao serem tidos em conta.

7 Conclusões

Neste capítulo são apresentadas as conclusões referentes ao projeto e à dissertação que foi criada através do mesmo, aliados aos objetivos que foram alcançados. Posteriormente, evoca-se qual o trabalho a realizar no futuro. Por último, realiza-se uma apreciação final acerca do projeto como um todo e a aprendizagem obtida a nível pessoal.

7.1 Resumo do relatório

Nesta secção é apresentado um resumo do relatório em si, tendo sempre em mente o contexto e problema em que se insere.

Hoje em dia todos os utilizadores são bombardeados com informação, bem como quando procuram algo, pretende que isso lhes seja fornecido o menor tempo possível. Além disto, se desejarem realizarem algum tipo de compra, querem que seja de uma forma rápida e fácil. Para uma empresa que o seu negócio seja o *e-commerce*, a sua aplicação *web* tem de ser rápida a responder às necessidades dos clientes, de forma a não os perderem. Com isto, existe uma necessidade de ter uma aplicação *web* otimizada do ponto de vista do desempenho da mesma.

Para alcançar uma aplicação *web* otimizada, deve-se primeiramente pesquisar sobre o que define uma página *web*, no que é que esta assenta em termos tecnológicos. Com o Capítulo 2 obtém-se conhecimento sobre elementos HTML importantes para todas as páginas *web*, bem

como métricas que ajudam a perceber se uma página se encontra lenta. Além disto, algumas boas práticas são enumeradas de modo a que haja um pensamento crítico desde o momento de inicialização de um projeto. Por último, um estudo das ferramentas de auditoria a páginas *web* permite obter mais informação, ficando-se a perceber quais os pontos que mais carecem de atenção por parte do desenvolvedor. Com o Capítulo 3 existe uma análise de valor que esta dissertação trará ao mundo tecnológico, passando por uma fase de geração e posteriormente seleção de ideias. Com o Capítulo 4 existe uma análise aos componentes que devem ser introduzidos numa página *web* relacionada com *e-commerce*, de forma a que se perceba a razão por serem estes alguns dos alvos de otimização de desempenho. Além disto, são enumerados princípios de arquitetura importantes para a realização da solução, tais como possuir um repositório central e utilizar o *design* atómico. De seguida, são estudadas as vistas lógica e de implementação, estando as de implantação e de processos nos anexos, pois não possuem tanta relevância. No Capítulo 5 são mencionadas boas práticas de implementação para uma solução com um desempenho otimizado, havendo um estudo aprofundado sobre o JavaScript em geral, mas também com parte relevante em relação ao React e suas peculiaridades. Existe uma efetivação de implementação dos componentes analisados no Capítulo 4, com base no *design* enunciado nesse mesmo capítulo. No Capítulo 6 existe uma avaliação da solução implementada, mostrando medições efetuadas relativas ao tempo e memória. Com estas medições percebem-se quais os cuidados que devem ser tidos em conta e que realmente têm impacto severo no desempenho de uma aplicação.

7.2 Objetivos alcançados

Após o estudo efetuado ao longo dos Capítulos 2 e 3, com o desenho de uma solução no Capítulo 4 e respetiva implementação no Capítulo 5, efetuou-se uma avaliação da mesma no Capítulo 6. Agora, resta retirar conclusões de todo este processo efetuado.

O objetivo principal desta dissertação é avaliar e otimizar uma aplicação desenvolvida em React. No entanto, para se alcançar este grande objetivo, deve-se efetivamente parti-lo em vários sub-objetivos, para que se compreendam as etapas a ultrapassar.

O primeiro sub-objetivo identificado pretende estudar os padrões e anti padrões existentes em React. Inicialmente, na Subsecção 5.1.1 faz-se um estudo intensivo sobre o ciclo de vida de um componente React. Percebe-se, por exemplo, em teoria o impacto que a função

shouldComponentUpdate tem na otimização de um componente e em maior plano numa aplicação. Um anti padrão identificado e que pode trazer problemas de desempenho é a atualização do estado na função componentDidMount apesar de o utilizador não sentir essa mesma atualização, de uma forma visual, a ser efetuada. Outro anti padrão que é habitualmente realizado é o de agrupar vários elementos dentro de um elemento HTML *div*. Com o padrão de fragmentos invocado na secção 5.1.4 deixa de ser necessário realizar este agrupamento, poupando no número de nós na DOM e por consequência nos recursos computacionais.

Para que se perceba onde existem problemas de desempenho de uma aplicação *web* é necessário encontrar uma forma de fazer auditoria a essa mesma aplicação. Assim, é necessário efetuar um estudo sobre as ferramentas de auditoria de desempenho em aplicações *web* que existem e determinar quais as que se adequam mais à realidade desta dissertação. Este estudo é realizado na secção 2.2.3., sendo posteriormente aplicado, nas secções 6.2.2. e 6.2.3., na avaliação de protótipos realizados de acordo com o pretendido nesta dissertação.

De forma a que se perceba quais os sintomas e causas de quebra de desempenho de uma aplicação existe um estudo aprofundado ligado, desde início, ao que são os elementos básicos de uma aplicação *web*, tal como demonstrado na Subsecção 2.1.1.1., seguindo para a enumeração de boas práticas, nomeadamente a redução do tamanho da DOM, como escrito na Subsecção 2.2.2.4. Por último, existem funções e práticas que não são apenas utilizadas quando se programa em React, mas sim em todo o ecossistema de JavaScript, pelo que são referidos igualmente na Subsecção 5.1.7. a 5.1.9, sendo a sua avaliação feito nas secções apropriadas do Capítulo 6.

Como último sub-objetivo, pretende-se demonstrar através de exemplos a transformação que é possível efetuar para que haja uma melhoria na globalidade da aplicação *web*. Assim, são mencionados na secção 5.2. componentes React que sofrem de melhorias significativas aplicadas com o conhecimento previamente adquirido através da conclusão dos sub-objetivos anteriores. De forma a que se compreenda o impacto que estas melhorias têm, realizam-se avaliações na secção 6.2.2 e 6.2.3. que deixam bem explícitos os ganhos ao se aplicar memorização em cache, virtualização de listas, componentes puros e a função shouldComponentUpdate.

Em suma, foram alcançados todos os objetivos a que esta dissertação se propôs na secção 1.3., demonstrando assim formas viáveis de se alcançar um bom desempenho no que toca a aplicação em React, tendo sempre como alicerce as ferramentas de auditoria.

7.3 Trabalho futuro

No mundo da tecnologia existem sempre tarefas por realizar, pelo que com os avanços da mesma é preciso estar em aprendizagem constante para que se possa tirar maior partido do que as tecnologias nos oferecem. Assim, existem algumas mais valias para serem tidas em atenção no futuro:

- Estudar a nova *contextAPI* oferecida pelo React, os seus pros e contras em relação ao Redux e se traz melhorias de desempenho acentuadas;
- Estudar os *hooks*, tendo sido estes introduzidos apenas para componentes funcionais, mudando assim um pouco o paradigma do React nesse tipo de componentes;
- Estudar o *webpack* e todas as suas valências, fazendo uma comparação entre minificadores, como por exemplo o *Terser* e o *UglifyJs*;
- Incluir nas *pipelines* testes de desempenho como o *Lighthouse* aqui estudado.

7.4 Apreciação final e pessoal

Para que uma dissertação esteja corretamente realizada não é necessário que todas as perguntas anteriormente levantadas estejam respondidas a cem por cento. Neste caso, todos os objetivos delineados foram alcançados, permitindo que houvesse uma melhoria significativa no desempenho da aplicação *web*, bem como a mitigação de problemas futuros em novas aplicações.

Esta dissertação teve como base as boas práticas de engenharia, suportando-se dos módulos lecionados no contexto da unidade curricular de TMDEI, com particular realce na pesquisa e escrita técnico-científica, análise de valor de negócio, bem como de experimentação e avaliação. Assim, para o futuro profissional e pessoal, existe uma exigência redobrada no momento de codificar, possuindo uma linha de pensamento mais abrangente.

Referências

Airbnb (2019) *Introduction · Enzyme*. Available at: https://airbnb.io/enzyme/ (Accessed: 12 August 2019).

Allee, V. (2006) 'What is ValueNet Works™ Analysis?', *Network*, pp. 2–5.

Appleseed, J. (2014) *E-Commerce Search Field Design and Its Implications - Articles - Baymard Institute*. Available at: https://baymard.com/blog/search-field-design (Accessed: 12 August 2019).

Arsenault, C. (2017) *Measuring Web Performance - Analyzing What Matters Most - KeyCDN*. Available at: https://www.keycdn.com/blog/measuring-web-performance (Accessed: 21 February 2019).

Arsenault, R. (2016) *Gone in Three Seconds: Your Customers Have No Patience for a Slow Website - Aberdeen*. Available at: https://www.aberdeen.com/techpro-essentials/gone-in-three-seconds-your-customers-have-no-patience-for-a-slow-website/ (Accessed: 18 February 2019).

Basques, K. (2019) *Get Started With Analyzing Runtime Performance*. Available at: https://developers.google.com/web/tools/chrome-devtools/evaluate-performance (Accessed: 11 February 2019).

Baymard (2019) *E-Commerce Product Lists & Filtering Usability: An Original Research Study - Product Lists & Filtering - Baymard Institute*. Available at: https://baymard.com/research/ecommerce-product-lists (Accessed: 12 August 2019).

Bohachenko, S. (2018) *How to design a product card in eCommerce*. Available at: https://lenal.eu/blog/How-to-design-a-product-card (Accessed: 12 August 2019).

Boldare (2019) *React Component Lifecycle*. Available at: https://facebook.github.io/react/docs/component-specs.html#lifecycle-methods (Accessed: 13 July 2019).

Buchanan, J. and Gardiner, L. (2003) 'A comparison of two reference point methods in multiple objective mathematical programming', *European Journal of Operational Research*. doi: 10.1016/S0377-2217(02)00487-3.

Bustos, L. (2016) *Product Lists: Pagination vs Infinite Scroll - Ecommerce Illustrated*. Available at: https://www.ecommerceillustrated.com/product-lists-pagination-vs-infinite-scroll/ (Accessed: 12 August 2019).

Costa, V. (2017) *A inteligência por trás do Atomic Design*. Available at: https://medium.com/dex01/inteligenciaportrasatomicdesign-1e405464ff5d (Accessed: 23 February 2019).

Cousins, C. (2017) *Why Does User Experience Matter?*, *Ceros Originals*. Available at: https://www.ceros.com/originals/why-does-user-experience-matter/ (Accessed: 15 January 2019).

Denysov, A. (2017) *Performance metrics. What's this all about?* Available at: https://codeburst.io/performance-metrics-whats-this-all-about-1128461ad6b (Accessed: 23 February 2019).

Einav, Y. and Shalom, N. (2012) *Amazon found every 100ms of latency cost them 1% in sales.*, *The GigaSpaves Technology Blog*. Available at: https://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/ (Accessed: 10 January 2019).

English, C. (2014) *Understanding Ecommerce Design, Part 3: Headers and Footers | Practical Ecommerce*. Available at: https://www.practicalecommerce.com/Understanding-Ecommerce-Design-Part-3-Headers-and-Footers (Accessed: 12 August 2019).

Facebook (2016) 'React - A declarative, efficient, and flexible JavaScript library for building user interfaces.' doi: 10.1038/nphys2993.

Facebook (2019) 'Optimizing performance'. Available at: https://reactjs.org/docs/optimizing-performance.html#shouldcomponentupdate-in-action (Accessed: 18 July 2019).

Farber, D. (2006) *Google´s Marissa Mayer: Speed wins*, *CNET News.com*. Available at: https://www.zdnet.com/article/googles-marissa-mayer-speed-wins/ (Accessed: 10 January 2019).

Fowler, M. (2005) *TestDrivenDevelopment*. Available at: https://martinfowler.com/bliki/TestDrivenDevelopment.html (Accessed: 20 January 2019).

Frost, B. (2016) *Atomic design*. Available at: http://atomicdesign.bradfrost.com/chapter-2/ (Accessed: 23 February 2019).

Gassmann, O. and Schweitzer, F. (2014) *Management of the Fuzzy front end of innovation*, *Management of the Fuzzy Front End of Innovation*. doi: 10.1007/978-3-319-01056-4.

Google (2018a) 'Aproveitar o processo de cache do navegador | PageSpeed Insights | Google Developers'. Available at: https://developers.google.com/speed/docs/insights/LeverageBrowserCaching (Accessed: 25 January 2019).

Google (2018b) 'Ativar a compactação | PageSpeed Insights | Google Developers'. Available at: https://developers.google.com/speed/docs/insights/EnableCompression (Accessed: 25 January 2019).

Google (2018c) *Reduzir recursos (HTML, CSS e JavaScript) | PageSpeed Insights | Google Developers*. Available at: https://developers.google.com/speed/docs/insights/MinifyResources (Accessed: 25 January 2019).

Google (2019a) *Auditar apps da Web com o Lighthouse | Tools for Web Developers*. Available at: https://developers.google.com/web/tools/lighthouse (Accessed: 11 February 2019).

Google (2019b) *GitHub - GoogleChrome/puppeteer: Headless Chrome Node.js API*. Available at: https://github.com/GoogleChrome/puppeteer (Accessed: 12 August 2019).

Google (2019c) *Uses An Excessive DOM Size | Tools for Web Developers | Google Developers*. Available at: https://developers.google.com/web/tools/lighthouse/audits/dom-size (Accessed: 25 January 2019).

Gupta, R., Kumar, M. and Rohit, B. (2016) *Data Compression - Lossless and Lossy Techniques*. Available at: www.ijaiem.org (Accessed: 21 February 2019).

Hogan, L. (2014) *Performance is User Experience | Designing for Performance*. Available at:

http://designingforperformance.com/performance-is-ux/ (Accessed: 20 January 2019).

Holst, C. (2018) *Drop-Down Usability: When You Should (and Shouldn't) Use Them - Articles - Baymard Institute*. Available at: https://baymard.com/blog/drop-down-usability (Accessed: 12 August 2019).

Iriondo, R. (2018) *Why Web Performance Matters (Small Businesses to Large Enterprises)*. Available at: https://medium.com/@robiriondo/why-web-performance-matters-small-businesses-to-large-enterprises-86efe13127b6 (Accessed: 20 January 2019).

Jackson, C. (2019) *Micro Frontends*. Available at: https://martinfowler.com/articles/micro-frontends.html (Accessed: 20 August 2019).

Jasmine (2019) *Jasmine Documentation*. Available at: https://jasmine.github.io/ (Accessed: 12 August 2019).

Jest (2019) *Jest · Delightful JavaScript Testing*. Available at: https://jestjs.io/ (Accessed: 12 August 2019).

Kirkpatrick, D. (2016) *Google: 53% of mobile users abandon sites that take over 3 seconds to load*, *MarketingDive*. Available at: https://www.marketingdive.com/news/google-53-of-mobile-users-abandon-sites-that-take-over-3-seconds-to-load/426070/ (Accessed: 18 February 2019).

Koen, P. a *et al.* (2002) 'Fuzzy Front End: Effective Methods, Tools, and Techniques', in *Bellviav, P., Griffin, A. and Somemeyer, S. (eds.) The PDMA Tool-Book for New Product Development*. doi: 10.1021/ed072p378.

Koussa, S. (2018) *What do SAST, DAST, IAST and RASP mean to developers?* Available at: https://www.softwaresecured.com/what-do-sast-dast-iast-and-rasp-mean-to-developers/ (Accessed: 12 August 2019).

Lechat, O. (2014) *Basic Structure of a Web Page*. Available at: https://www.sitepoint.com/basic-structure-of-a-web-page/ (Accessed: 21 February 2019).

Maldonado, L. (2018) *What's the Document Object Model, and why you should know how to use it.* Available at: https://medium.freecodecamp.org/whats-the-document-object-model-

and-why-you-should-know-how-to-use-it-1a2d0bc5429d (Accessed: 21 February 2019).

Marchese, L. (2016) *The Psychology of Checklists: Why Setting Small Goals Motivates Us to Accomplish Bigger Things*, *Trello*. Available at: https://blog.trello.com/the-psychology-of-checklists-why-setting-small-goals-motivates-us-to-accomplish-bigger-things (Accessed: 7 January 2019).

McCarthy, J. and Wright, P. (2004) 'Technology as experience', *interactions*. doi: 10.1145/1015530.1015549.

Mocha (2019) *Mocha - the fun, simple, flexible JavaScript test framework*. Available at: https://mochajs.org/ (Accessed: 12 August 2019).

Morar, D. (2013) 'An overview of the consumer value literature–perceived value, desired value', *Marketing From Information to Decision*.

Nair, J. (2018) *TDD vs BDD - What's the Difference Between TDD and BDD? - TestLodge Blog*. Available at: https://blog.testlodge.com/tdd-vs-bdd/ (Accessed: 12 August 2019).

Nguyen, H. H. and Rinard, M. (2007) 'Detecting and eliminating memory leaks using cyclic memory allocation', in *Proceedings of the 6th international symposium on Memory management - ISMM '07*. doi: 10.1145/1296907.1296912.

Nicola, S., Ferreira, E. P. and Ferreira, J. J. P. (2012) 'A NOVEL FRAMEWORK FOR MODELING VALUE FOR THE CUSTOMER, AN ESSAY ON NEGOTIATION', *International Journal of Information Technology & Decision Making*, 11(03), pp. 661–703. doi: 10.1142/S0219622012500162.

OpenJS (2019) *WebdriverIO · Next-gen WebDriver test framework for Node.js*. Available at: https://webdriver.io/ (Accessed: 12 August 2019).

Osmani, A. (2012) 'Learning JavaScript Design Patterns', *JavaScript*. doi: 10.1017/CBO9781107415324.004.

Osterwalder, A., Pigneur, Y. and Smith, A. (2010) *Business Model Generation*, *Booksgooglecom*. doi: 10.1523/JNEUROSCI.0307-10.2010.

OWASP (2018) *Cross-site Scripting (XSS) - OWASP*. Available at:

https://www.owasp.org/index.php/Cross-site_Scripting_(XSS) (Accessed: 12 August 2019).

Peffers, K. *et al.* (2007) 'A Design Science Research Methodology for Information Systems Research', *Journal of Management Information Systems*. doi: 10.2753/MIS0742-1222240302.

Portswigger (2019) *What is SQL Injection? Tutorial & Examples*. Available at: https://portswigger.net/web-security/sql-injection (Accessed: 12 August 2019).

Reimann, D. (2018) *Atomic Design is messy, here's what I prefer*. Available at: https://dennisreimann.de/articles/atomic-design-is-messy.html (Accessed: 23 February 2019).

Rouse, M. (2018) *What is cache (computing)?* Available at: https://searchstorage.techtarget.com/definition/cache (Accessed: 21 February 2019).

Saaty, T. L. (1990) 'How to make a decision: The analytic hierarchy process', *European Journal of Operational Research*. doi: 10.1016/0377-2217(90)90057-I.

Sailing, R. (2012) *The Importance of Buttons and Strong Calls to Action*. Available at: https://www.tipsandtricks-hq.com/the-importance-of-buttons-and-strong-calls-to-action-4326 (Accessed: 12 August 2019).

Salman, A. (2018) *How to NOT React: Common Anti-Patterns and Gotchas in React*. Available at: https://codeburst.io/how-to-not-react-common-anti-patterns-and-gotchas-in-react-40141fe0dcd (Accessed: 19 February 2019).

Scott, P. (2017) *Mono-repo or multi-repo? Why choose one, when you can have both?* Available at: https://medium.com/@patrickleet/mono-repo-or-multi-repo-why-choose-one-when-you-can-have-both-e9c77bd0c668 (Accessed: 10 February 2019).

Shneiderman, B. (1984) 'Response time and display rate in human performance with computers', *ACM Computing Surveys*, 16(3), pp. 265–285. doi: 10.1145/2514.2517.

Sirois, J.-P. and Hall, Z. (2019) *Lodash*. Available at: https://lodash.com/ (Accessed: 12 August 2019).

TestCafe (2019) *A node.js tool to automate end-to-end web testing | TestCafe*. Available at: https://devexpress.github.io/testcafe/ (Accessed: 12 August 2019).

Vaughn, B. (2018) *Introducing the React Profiler – React Blog*, *React*. Available at: https://reactjs.org/blog/2018/09/10/introducing-the-react-profiler.html#reading-performance-data (Accessed: 4 August 2019).

Vaughn, B. (2019) *GitHub - bvaughn/react-window: React components for efficiently rendering large lists and tabular data*. Available at: https://github.com/bvaughn/react-window (Accessed: 11 August 2019).

Vocke, H. (2018) *The Practical Test Pyramid*. Available at: https://martinfowler.com/articles/practical-test-pyramid.html#UnitTests (Accessed: 25 July 2019).

Wang, X. S. *et al.* (2013) 'Demystifying Page Load Performance with WProf', in *USENIX Symposium on Networked Systems Design and Implementation*, pp. 473–485. doi: 10.1016/j.prevetmed.2016.08.006.

WebPageTest (2019) *WebPageTest - Website Performance and Optimization Test*. Available at: https://www.webpagetest.org/ (Accessed: 11 February 2019).

Williams, S. (2019) *Can icons harm usability and when should you use them?* Available at: https://uxdesign.cc/when-should-i-be-using-icons-63e7448202c4 (Accessed: 12 August 2019).

Woodall, T. (2003) 'Conceptualising "Value for the Customer": An Attributional, Structural and Dispositional Analysis', *Academy of Marketing Science Review*. doi: 10.1007/s11661-004-0356-5.

Anexo A *Frameworks*

Uma *framework* é um pacote que tem como função dar suporte à construção de aplicações web dinâmicas. Esta ajuda o programador a desenvolver uma página web praticamente do nada. Atualmente no mercado existe variadíssimas opções pelas quais o programador pode optar, no entanto, nesta secção iremos comparar as três *frameworks* mais utilizadas do mercado.
O Angular é desenvolvido pela Google e foi disponibilizado ao público pela primeira vez em 2010. É o mais antigo dos três e sofreu modificações substanciais quando foi lançada a segunda versão desta *framework*. Tem como pontos fortes a ligação bidirecional dos dados, onde replica as alterações feitas no modelo para a secção de visualização de uma instantânea e eficiente. Além disto, tem uma comunidade bastante grande por detrás dela. Como pontos fracos, a *framework* possui imensas funcionalidades, pelo que nem sempre são necessárias para o programador, o que torna a aplicação demasiado pesada para projetos simples. Também, está sempre em constante evolução e a sua curva de aprendizagem é maior do que a das outras duas.

O React é desenvolvido pelo Facebook e é uma *framework* usada para criar componentes visuais que possuem um estado e que podem ser reutilizados. Permite que os programadores criem aplicações *web* que não precisam de ser recarregadas a cada mudança de dados. Tem como pontos fortes utilizar uma DOM virtual, ou seja, quando realiza uma comparação entre esta DOM e a que existe no navegador, apenas rerenderiza os nós que são necessários. Além disto, tem uma ligação unidirecional dos dados, dando maior controlo ao programador, bem como também é fácil de testar, permitindo que haja um processo de desenvolvimento mais seguro. Como pontos fracos, tem a necessidade de se ir acrescentando bibliotecas ao projeto para lidar com o estado e o modelo dos componentes e o facto de ter uma documentação não muito exaustiva, visto esta *framework* estar sempre em constante mudança.

O Vue é uma *framework* que não pertence a uma empresa, sendo também a mais recente das três estudadas. No geral, tenta reaproveitar algumas funcionalidades das restantes, no entanto, aproxima-se mais do React no que toca a possuir uma DOM virtual e em focar-se apenas nas operações mais importantes, delegando as restantes tarefas para outras bibliotecas. Como pontos fortes, possui um desempenho geral bastante bom, sendo fácil de usar e aprender. Além

disto, a sua curva de aprendizagem é menos acentuada que no Angular e possui melhor documentação que o React. Por outro lado, como pontos fracos tem uma pequena adoção por parte da comunidade, levando a que haja um mercado de trabalho bastante mais escasso.

Tabela 23 - Comparação entre *frameworks*.

Frameworks	**Pontos fortes**	**Pontos fracos**
Angular	• Ligação bidirecional dos dados; • Comunidade bastante grande; • Desenvolvido pela Google.	• Pesado para aplicações simples; • Em constante mudança; • Maior curva de aprendizagem grande.
React	• Ligação unidirecional dos dados; • Utilização de DOM virtual; • Fácil de testar; • Desenvolvido pelo Facebook.	• Pouca documentação; • Em constante mudança; • Acréscimo de bibliotecas externas.
Vue	• Bom desempenho; • Curva de aprendizagem pequena; • Boa documentação; • Utilização de DOM virtual.	• Pequena comunidade; • Mercado de trabalho diminuto; • Acréscimo de bibliotecas externas; • Não tem uma grande empresa por trás.

Com esta análise realizada é possível compreender que de entre as três *frameworks* estudadas, o Angular é a que traz menos benefícios para o estudo aliado ao desempenho de uma aplicação

web. A pequena comunidade e o mercado de trabalho escasso do Vue, além da já pequena familiaridade existente do aluno para com o React, torna esta última *Framework* a mais apropriada a ser estudada.

Anexo B *Vista de Processos*

Anexo B.1 *Versão aplicada*

A vista de processos pretende explicar os processos do sistema e como eles comunicam entre si. De forma a que esta comunicação seja explicitada de forma a que o leitor possa entender é efetuado um diagrama de sequência. Estes diagramas permitem que se perceba qual o caminho que percorre uma ação despoletada inicialmente por um utilizador junto da interface gráfica.

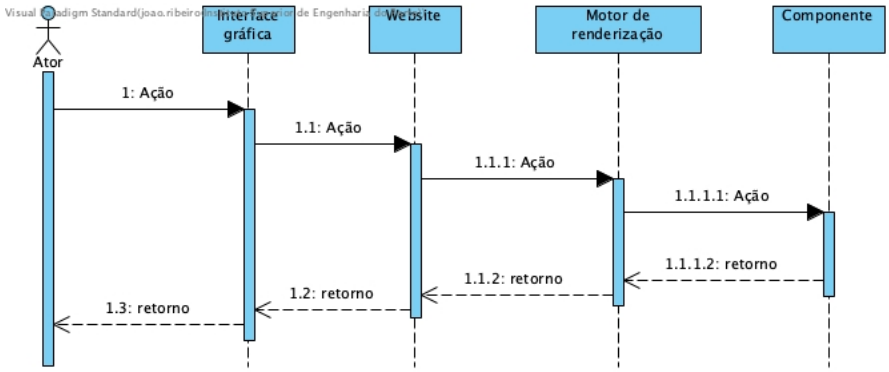

Figura 27 - Diagrama representativo da vista de processos.

Com a representação da Figura 27 pode-se perceber que uma ação despoletada por um ator junto da interface gráfica tem impacto logo de seguida no *website*. Este, após receber a instrução dada pela interface gráfica, realiza uma chamada ao motor de renderização para que seja efetuada a modificação pretendida pelo ator. De seguida, o motor de renderização ao ser invocado, percebe em qual dos componentes é necessário efetuar uma modificação, dirigindo a ação para o mesmo. O componente que se encontra envolvido nesta cadeia de ações tem o dever de se adaptar ao que o ator pretende, por exemplo, ao introduzir-se um valor no componente de texto, este guardar o valor introduzido em memória. Após efetuar as tarefas que lhe são devidas, o componente pode ter de comunicar com outros componentes ou então apenas retornar a informação de que a ação despoletada foi finalizada com sucesso para o motor de renderização, sendo essa informação passada até ao ator de forma visual.

Anexo B.2 *Versão alternativa*

No caso de se seguir o mesmo pensamento que foi feito na secção 4.2.2, onde se refere a existência de um *web service* e uma base de dados fora da aplicação *web*, então a Figura 28 vai de encontro ao mesmo pensamento. Assim, existe uma similaridade até ao momento em que se encontra no componente, no entanto, ação chega ao *web service*, sendo depois efetuada alguma ação na base de dados, por exemplo de consulta de informação, retornando essa mesma informação até à interface gráfica.

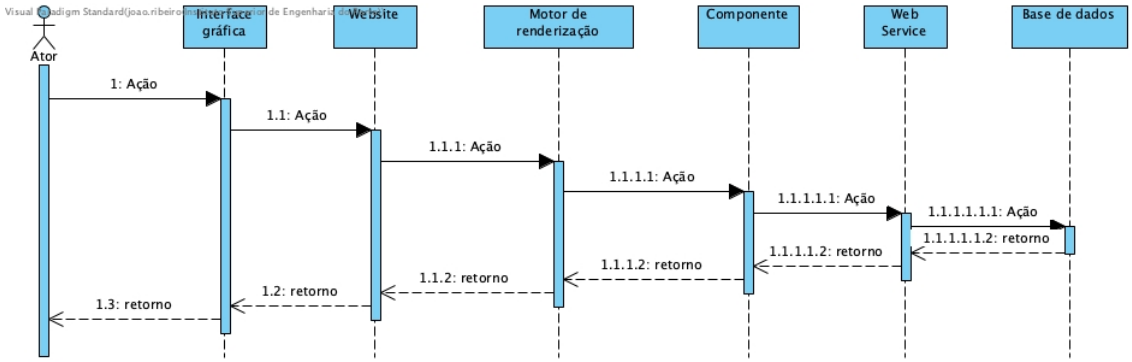

Figura 28 - Diagrama alternativo da vista de processos.

Anexo C *Vista de Implantação*

Anexo C.1 *Versão aplicada*

A visão de implantação descreve o sistema do ponto de vista de um engenheiro da especialidade. Este preocupa-se com a topologia dos componentes de *software* na camada física, tendo também em atenção as conexões entre esses mesmos componentes. Com o diagrama da Figura 29 pode-se perceber que toda a aplicação *web* estará alojada localmente num computador para que não haja interferências de rede no momento em que a aplicação *web* estiver em execução. Assim, permite-se que haja um maior foco na aplicação *web* e na maneira como esta foi desenvolvida para que se encontre melhorias no desempenho da mesma.

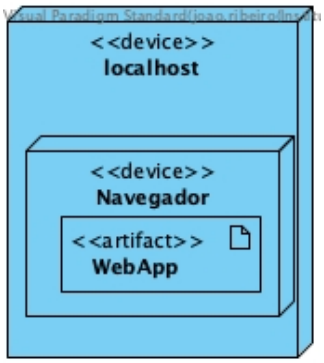

Figura 29 - Diagrama representativo da vista de implantação.

Anexo C.2 *Versão alternativa*

Como alternativa ao diagrama da Figura 29 existe a possibilidade de repartir a aplicação *web* em *web services* e base de dados externos, tal como já foi mencionado nas alternativas das vistas lógica e de processos. Com a Figura 30, existe o deslocamento do *web service* para um *web server* externo que comunica com o computador do cliente, que possuindo um navegador consegue utilizar a aplicação *web*. Por último, o *web service* também comunica com o servidor

de base de dados de forma a conseguir obter a informação que seja pedida pelo utilizador através da aplicação *web* no navegador.

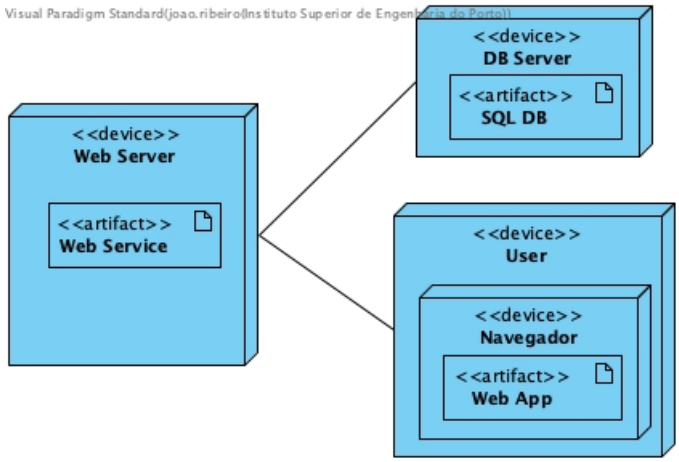

Figura 30 - Diagrama alternativo da vista de implantação.