

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Arquitetura Orientada a Componentes para uma Web Responsiva

Rui Tiago Bugalho Monteiro



Mestrado Integrado em Engenharia Informática e Computação

Orientador: Prof. João Correia Lopes

27 de Julho de 2015

Arquitetura Orientada a Componentes para uma Web Responsiva

Rui Tiago Bugalho Monteiro

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Prof. Cristina Ribeiro

Arguente: Prof. Benedita Malheiro

Vogal: Prof. João Correia Lopes

27 de Julho de 2015

Resumo

O desenvolvimento de *software* para a *World Wide Web* tem evoluído em função do crescimento do seu ecossistema de dispositivos. Se, nos seus primeiros anos, a *Web* era maioritariamente composta por páginas navegáveis informativas, que apenas podiam ser visualizadas em computadores pessoais, com ligações à Internet estáveis, hoje, uma aplicação *Web* pode ser usada para um número indeterminado de fins e pode ser corrida, tanto em computadores pessoais, como em *smartphones*, ou em *tablets*, *Smart TVs*, etc. e através de redes sem fios pouco estáveis ou até mesmo em locais em que não é possível conseguir-se, de todo, uma ligação à Internet. Por esta razão, é preciso criar novas formas para se desenvolverem as Aplicações *Web* dos dias de hoje, que permitam responder às necessidades criadas por todos estes novos paradigmas de utilização e que permitam aos programadores concentrarem-se na criação de aplicações capazes de oferecer boas experiências de utilização aos seus utilizadores e esquecerem a elevada complexidade inerente à implementação de aplicações com um espetro de dispositivos-alvo tão amplo.

Os conceitos *Web Components*, *Offline-first*, *Responsive Web Design* e *Single-Page Applications* introduzem ideias que dão resposta à elevada complexidade do desenvolvimento das Aplicações *Web* atuais e prometem ser eles os autores dos próximos livros de desenvolvimento de aplicações para a *Web*. Nesta dissertação é proposta uma arquitetura *front-end* para o desenvolvimento de Aplicações *Web* baseada nestes quatro conceitos, que será composta por uma seleção de tecnologias que já os implementam. São enumeradas as tecnologias que foram escolhidas para integrar essa arquitetura, explicadas as razões que justificaram cada escolha e descritos alguns dos detalhes de implementação mais importantes da arquitetura proposta.

Para validar a arquitetura *front-end* proposta foi desenvolvida, como prova de conceito, uma aplicação *Web* — Nomnow. A aplicação desenvolvida apresentou um código-fonte separado por componentes, intuitivo e de fácil manutenção, e uma experiência de utilização rica, com uma interface do utilizador responsiva e mantendo-se funcional mesmo sem uma ligação à Internet.

Confirmou-se, por fim, que as tecnologias selecionadas para integrar a arquitetura proposta por esta dissertação, e, por sua vez, os quatro conceitos explorados por essas tecnologias, oferecem efetivamente vantagens significativas na simplificação do processo de desenvolvimento de *software* para a *Web*.

Abstract

Software development for the World Wide Web has evolved with the growth of its ecosystem of devices. If, in its early days, the Web was mostly the place where people could get access to some simple websites, via desktop computers, with stable Internet connections, today, Web Applications can be created for a countless number of purposes and may be run not only by desktop computers, but also by smartphones, tablets, Smart TVs, etc. and accessed via wireless networks, which are so often unstable, or even in situations when it's not possible at all to get an Internet connection. Therefore, it's imperative that we find new ways to develop the Web Applications of today, so they can respond to the needs of all these new usage paradigms and allow developers to focus on creating software able to provide their users with good user experiences and to forget about the high complexity that comes with the development of software with such a broad spectrum of target devices.

The concepts Web Components, Offline-first, Responsive Web Design and Single-Page Applications propose new ideas which are mean to fight the high complexity of developing the Web Applications of today and vow to be the future authors of the next Web software development books. In this work, it is proposed a front-end architecture for Web development based on these four concepts, composed by a selection of technologies which already implement the ideas of these concepts. Furthermore, the technologies chosen to be part of the proposed architecture, as well as the reasons for choosing each one of them, and some of its most relevant implementation details are described.

To validate the proposed front-end architecture, a Web application was developed as proof of concept — Nomnow. The source code of the developed application was highly separated by components and it was intuitive and easy to maintain. Also, the application itself featured a responsive user interface and could be used even with no Internet connection.

Finally, it was concluded that the technologies chosen to be part of the proposed architecture, and, therefore, the four concepts implemented by those technologies, offer significant benefits in terms of simplifying the development process of software for the Web platform.

Agradecimentos

Ao professor João Correia Lopes, o meu muito obrigado por ter aceite o meu convite para orientar este trabalho e por ter desempenhado esse papel de forma exímia. Sem o seu empenho e a sua dedicação, não teria sido possível levar este projeto a bom porto.

Não menos importante, agradeço à Glazed Solutions, Lda., por me ter dado a oportunidade para desenvolver este projeto interessantíssimo num ambiente de trabalho extraordinário e, em especial, aos Engenheiros Luís Martinho e Pedro Melo Campos, que me ajudaram a ultrapassar grande parte das contrariedades que foram surgindo com a realização deste trabalho.

Este trabalho não teria sido ainda possível sem a ajuda da minha família, dos meus amigos e de todos aqueles que me acompanharam no meu percurso de formação académica e pessoal. Um forte abraço e um grande obrigado a todos.

Rui Tiago Bugalho Monteiro

*“There’s an old story about the person
who wished his computer were as easy to use as his telephone.
That wish has come true, since I no longer know how to use my telephone”*

Bjarne Stroustrup

Conteúdo

1	Introdução	1
1.1	Contexto	1
1.2	Motivação	1
1.3	Objetivos	2
1.4	Estrutura da Dissertação	2
2	Desenvolvimento de Aplicações Web	3
2.1	Introdução	3
2.2	Paradigmas de Utilização e Requisitos Emergentes	4
2.2.1	Criação de <i>Layouts</i> Dinâmicos	4
2.2.2	Imagens por Densidade de Ecrã	4
2.2.3	Reconhecimento de Novos Métodos de Entrada	5
2.2.4	Funcionamento <i>Offline</i>	5
2.3	Complexidade de Desenvolvimento	6
2.3.1	Elementos Nativos de HTML5	6
2.3.2	Regras de CSS	6
2.3.3	“Pyramid of Doom” do JavaScript	7
2.4	Conclusões	7
3	Tecnologias Emergentes	9
3.1	Introdução	9
3.2	<i>Web Components</i>	9
3.2.1	Linguagens Declarativas nas Aplicações <i>Web</i>	10
3.2.2	<i>Custom Elements</i>	10
3.2.3	<i>HTML Imports</i>	11
3.2.4	<i>HTML Templates</i>	11
3.2.5	<i>Shadow DOM</i>	12
3.2.6	Bibliotecas e <i>Frameworks</i> Disponíveis	12
3.3	<i>Offline-first</i>	13
3.3.1	Armazenamento de Dados	14
3.3.2	<i>Cache</i> do Código-fonte	14
3.3.3	<i>NoBackend</i>	15
3.3.4	<i>Frameworks</i>	15
3.4	<i>Responsive Web Design</i>	17
3.4.1	Princípios de Design	17
3.4.2	<i>Mobile first</i>	18
3.4.3	Material Design	18
3.4.4	Desenvolvimento de Interfaces do Utilizador	19

CONTEÚDO

3.4.5 <i>Web Animations API</i>	20
3.4.6 <i>Responsive Images</i>	21
3.5 <i>Single-Page Applications</i>	21
3.5.1 <i>Architectural Patterns</i>	22
3.5.2 <i>Design Patterns Emergentes</i>	23
3.5.3 <i>ECMAScript 6 e 7</i>	24
3.5.4 Testes Unitários	26
3.6 Conclusões	27
4 Proposta de Arquitetura Front-end	29
4.1 Tecnologias Selecionadas	29
4.1.1 Polymer para <i>Web Components</i>	30
4.1.2 <i>Offline-first</i> com Hoodie	30
4.1.3 Outros Componentes	31
4.2 Visão Geral da Arquitetura	31
4.3 Detalhes de Implementação	33
4.3.1 Declaração de um <i>Custom Element</i>	34
4.3.2 Comunicação entre Elementos	35
4.3.3 <i>Custom Element</i> para o Hoodie	36
4.3.4 Sessões, Coleções e <i>Queries</i>	36
4.3.5 <i>Custom Elements</i> para Plugins Hoodie	38
4.3.6 Interfaces Dinâmicas	39
4.3.7 <i>Layouts</i> Responsivos	40
4.3.8 Temas Globais	41
4.3.9 Testes Unitários	43
4.4 Ambiente de Desenvolvimento	43
4.5 Conclusões	45
5 Implementação de Prova de Conceito	47
5.1 Descrição da Aplicação	47
5.2 Especificação de Requisitos	48
5.2.1 Casos de Uso	48
5.2.2 Requisitos Não Funcionais	49
5.3 Detalhes de Implementação	49
5.3.1 Autenticação com Facebook	49
5.3.2 Integração da <i>Framework Parse</i>	50
5.3.3 Visão Geral da Arquitetura da Aplicação	51
5.3.4 Comportamento do <i>Service Worker</i>	51
5.3.5 Estrutura da Single-Page Application	52
5.3.6 Responsividade	54
5.3.7 Elementos Reutilizáveis	57
5.4 Resultados	58
5.4.1 Capturas de Ecrã	58
5.4.2 Critérios a Avaliar	60
5.4.3 Linhas de Código	60
5.4.4 Testes à Responsividade	61
5.4.5 Satisfação de Outros Requisitos	61
5.5 Conclusão	62

CONTEÚDO

6 Discussão de Resultados	63
6.1 Complexidade de Desenvolvimento	63
6.1.1 <i>Web Components</i> e ECMAScript 6 e 7	63
6.1.2 Comparação entre Aplicações Web e iOS	64
6.2 Responsividade	65
6.3 Funcionamento <i>Offline</i>	66
6.4 Limitações das Tecnologias Usadas	66
6.4.1 <i>Web Components</i>	66
6.4.2 Hoodie	67
6.5 Conclusões	67
7 Conclusões e Trabalho Futuro	69
7.1 Resumo	69
7.2 Contribuição Científica	70
7.3 Satisfação dos Objetivos	70
7.4 Trabalho Futuro	70
Referências	73

CONTEÚDO

Lista de Figuras

2.1	Interface da aplicação <i>Web</i> do The New York Times	4
2.2	Excerto de código <i>HTML</i> da aplicação <i>Web</i> atual do Facebook	7
2.3	Exemplo de <i>Pyramid of Doom</i> do Javascript	8
3.1	UI de aplicação <i>Web</i> com Material Design	19
3.2	Captura de ecrã da aplicação <i>Web</i> do serviço Gmail	22
3.3	Exemplo de <i>Arrow Function</i>	25
3.4	Exemplo de classe em ECMAScript 6	25
3.5	Sequência de operações assíncronas em JavaScript	26
3.6	Teste unitário implementado com Mocha e Chai	27
4.1	Diagrama da visão geral da arquitetura proposta	32
4.2	Estrutura base de uma SPA com <i>Custom Elements</i>	33
4.3	Declaração de <i>Custom Element</i> com Polymer	34
4.4	<i>Custom Element</i> para o Hoodie	36
4.5	Login com <i>Custom Element</i> Hoodie	37
4.6	<i>Query</i> com <i>Custom Element</i> Hoodie	38
4.7	Exemplo de uso de um <i>plugin</i> de Hoodie	39
4.8	Exemplo de uso de <i>template</i> condicional	40
4.9	Métodos do Polymer para subscrever alterações em propriedades	40
4.10	Exemplo de elemento responsivo com “core-media-query”	41
4.11	Exemplo de reutilização de um componente com estilos diferentes	42
4.12	<i>Binding</i> a código CSS	42
5.1	Casos de Utilização da aplicação Nomnow	48
5.2	Comunicação entre o Hoodie, o servidor de SocketIO e o Parse	51
5.3	Diagrama da visão geral da arquitetura da prova de conceito	52
5.4	Estrutura de <i>Custom Elements</i> da <i>Single-Page Application</i>	53
5.5	Excerto do código-fonte do componente “nomnow-responsive-layout”	55
5.6	Excerto do código de instância do componente “nomnow-responsive-layout”	56
5.7	Excerto da declaração do elemento “nomnow-phone-content”	56
5.8	Excerto da declaração do elemento “nomnow-tablet-content”	57
5.9	Interface do utilizador da aplicação para dispositivos móveis	59
5.10	Interface de conclusão de pedido de encomenda para dispositivos móveis	59
5.11	Interface do utilizador da aplicação para computadores pessoais	60

LISTA DE FIGURAS

Lista de Tabelas

5.1	Número de linhas de código da aplicação <i>Web Nomnow</i>	61
5.2	Resultado dos testes de responsividade	61
6.1	Número de linhas de código da aplicação de iOS	64
6.2	Número de linhas de código da aplicação <i>Web</i>	65

LISTA DE TABELAS

Abreviaturas e Símbolos

WWW	<i>World Wide Web</i>
AJAX	<i>Asynchronous JavaScript and XML</i>
CSS	<i>Cascading Style Sheets</i>
HTML	<i>HyperText Markup Language</i>
W3C	<i>World Wide Web Consortium</i>
XSS	<i>Cross-Site Scripting</i>
DOM	<i>Document Object Model</i>
UI	<i>User Interface</i>
MXML	<i>Magic eXtensible Markup Language</i>
XAML	<i>eXtensible Application Markup Language</i>
IDE	<i>Integrated Development Environment</i>
API	<i>Application Programming Interface</i>
HTTP	<i>Hypertext Transfer Protocol</i>
ES	<i>ECMAScript</i>
Sass	<i>Syntactically Awesome Style Sheets</i>
GPU	<i>Graphics Processing Unit</i>
CPU	<i>Central Processing Unit</i>
SPA	<i>Single-Page Application</i>

Capítulo 1

Introdução

1.1 Contexto

Este trabalho foi realizado em contexto empresarial, na Glazed Solutions, Lda., e apresentado como dissertação do Mestrado Integrado em Engenharia Informática e Computação, da Faculdade de Engenharia da Universidade do Porto. A Glazed Solutions, Lda. é uma empresa especializada em desenvolvimento de *software* para várias plataformas, das quais se destacam iOS, Android e *Web*, e que conta com uma lista de clientes que inclui várias empresas nacionais e internacionais líderes nos seus setores.

Este trabalho foca-se ainda na área do desenvolvimento de Aplicações *Web* e é contextualizado pelo aparecimento dos conceitos *Web Components*, *Offline-first*, *Responsive Web Design* e *Single-Page Applications*, introduzidos ao mundo da *World Wide Web* (WWW) nos últimos anos, que se comprometem a revolucionar o futuro do desenvolvimento de *software* para a *Web*.

1.2 Motivação

O aparecimento, nos últimos anos, de tecnologias que já implementam e se guiam pelos conceitos de *Web Components*, *Offline-first*, *Responsive Web Design* e *Single-Page Applications* motiva o estudo das suas vantagens e dos seus contratempos, bem como a descoberta de novas arquiteturas que suportem o desenvolvimento de Aplicações *Web* que tirem o máximo partido do seu uso conjunto.

É uma visão aliciante, pensar-se numa *Web* onde as aplicações sejam capazes de proporcionar uma experiência de utilização semelhante à das aplicações nativas, sejam mais fáceis de desenvolver e de manter e que, por não deixarem de ser Aplicações *Web*, continuem a ser compatíveis com todas as plataformas e dispositivos que tirem proveito do canal de distribuição universal que é a *World Wide Web*. A principal motivação desta dissertação é acreditar que as tecnologias que são abordadas por esta dissertação vão fazer parte do próximo passo nesse domínio.

1.3 Objetivos

O principal objetivo deste trabalho é propor uma arquitetura de *front-end* para o desenvolvimento de Aplicações *Web* que concilie os pontos fortes das tecnologias disponíveis que já implementem os conceitos *Web Components*, *Offline-first*, *Responsive Web Design* e *Single-Page Applications*. É ainda esperado que o desenvolvimento das aplicações que implementem a arquitetura proposta seja o mais simples possível e que as aplicações criadas ofereçam experiências de utilização ricas.

Porém, antes disto, é ainda importante explorar os quatro tópicos nos quais essa arquitetura se vai basear e avaliar as vantagens e desvantagens das tecnologias disponíveis.

1.4 Estrutura da Dissertação

Para além da introdução, esta dissertação contém mais 6 capítulos:

- No Capítulo 2 são descritos alguns dos principais problemas que se observam atualmente no desenvolvimento de aplicações para a *World Wide Web*.
- No Capítulo 3 é descrito o estado da arte das tecnologias *Web Components*, *Offline-first*, *Responsive Web Design* e *Single-Page Applications* e são abordadas e comparadas *frameworks* e bibliotecas que já implementam algumas destas tecnologias.
- No Capítulo 4 é descrita a arquitetura *front-end* proposta nesta dissertação e justificadas as escolhas para os seus componentes integrantes.
- No Capítulo 5 é descrita a implementação da aplicação desenvolvida como prova de conceito da arquitetura proposta no Capítulo 4.
- No Capítulo 6 é feita a discussão dos resultados obtidos com a implementação da prova de conceito da arquitetura proposta.
- Por fim, o Capítulo 7 — faz um resumo deste trabalho e da sua contribuição para a comunidade e enumera o que poderá ser feito, no futuro, para dar continuidade a este trabalho.

Capítulo 2

Desenvolvimento de Aplicações Web

Este capítulo introduz sucintamente o conceito de Aplicação *Web* e descreve alguns dos paradigmas de utilização mais visíveis atualmente neste tipo de aplicações. É ainda feito um levantamento de requisitos, que surgem como consequência desses paradigmas, bem como de algumas das principais dificuldades do desenvolvimento de aplicações que cumpram esses requisitos.

2.1 Introdução

O desenvolvimento de *software* para a *Web* evoluiu radicalmente ao longo dos anos, desde a sua origem. Nos primeiros anos, o seu curto portefólio de aplicações resumia-se a páginas *HyperText Markup Language* (HTML) estáticas, identificadas por um endereço único na rede, que permitiam a navegação para outras páginas através de hiperligações. Alguns anos depois, em 2005, com o aparecimento do *Asynchronous JavaScript and XML* (AJAX)¹, passou a ser possível fazer atualizações incrementais na Interface do Utilizador (UI) de páginas *Web* e começaram assim a ser criadas Aplicações *Web*, capazes de oferecer uma experiência de navegação mais próxima da oferecida pelas aplicações nativas, desenvolvidas para computadores pessoais. Este progresso, porém, deixou de ser suficiente, quando a teia da *World Wide Web* foi alargada ao ecossistema de dispositivos que a surgiram mais tarde.

O aparecimento, nos últimos anos, dos dispositivos móveis, das redes sem fios e a evolução das tecnologias *Web* têm vindo a tornar necessário pensar-se em novas formas de se desenvolverem Aplicações *Web*, adequadas ao contexto tecnológico e aos paradigmas de utilização da *Web* atuais.

¹Tecnologia que permite fazer trocas de dados com um servidor remoto assincronamente e fazer alterações dinamicamente a elementos do *Document Object Model* (DOM) de páginas *Web*.

2.2 Paradigmas de Utilização e Requisitos Emergentes

Hoje em dia, desenvolver *software* para a *Web* significa desenvolver *software* que esteja preparado para correr em diferentes sistemas operativos, tanto de computadores pessoais como de dispositivos móveis, e em ecrãs com resoluções e densidades diferentes. Deve significar ainda desenvolver *software* que não perca funcionalidade quando os métodos tradicionais de interação com o utilizador, como o rato e o teclado, são substituídos por outros, como o *touch* e o acelerómetro, ou quando a ligação à Internet é instável ou inexistente, como acontece frequentemente em *tablets* e *smartphones*.

2.2.1 Criação de *Layouts* Dinâmicos

Antes do aparecimento dos dispositivos móveis, desenvolver Aplicações *Web* capazes de proporcionar boas experiências de utilização em ecrãs de pequenas dimensões não era uma preocupação. As interfaces das Aplicações *Web* que já existiam estavam desenhadas para ecrãs de grandes dimensões como os dos computadores pessoais. Este tipo de interfaces é, no entanto, pouco adequado quando usado em *tablets* ou em *smartphones*. A Figura 2.1 ilustra a visualização de uma interface deste tipo num *smartphone*.



Figura 2.1: Interface da aplicação *Web* do The New York Times

A execução, em dispositivos com ecrãs de pequenas dimensões, de aplicações desenvolvidas com este tipo de interfaces torna o seu conteúdo pouco legível e obriga os seus utilizadores a recorrerem constantemente a funcionalidades de zoom.

2.2.2 Imagens por Densidade de Ecrã

Na altura, também não era preocupação disponibilização de imagens em vários tamanhos, antecipando a execução dessas aplicações em dispositivos com ecrãs tanto de altas como de baixas densidades. Era, por oposição, frequente as Aplicações *Web* disponibilizarem imagens que de tamanho único e que, por essa razão, acabassem por ser transferidas imagens com tamanhos

desnecessariamente grandes para os ecrãs pequenos e de baixa densidade ou imagens com pouca definição em dispositivos com ecrãs de grandes dimensões e com densidades altas.

A transferência de imagens desnecessariamente grandes usa uma largura de faixa superior à que seria usada caso a aplicação disponibilizasse imagens com tamanhos adequados ao dispositivo. Como consequência, a aplicação irá demorar mais tempo a carregar todos os seus recursos e perderá fluidez desnecessariamente. No caso oposto, quando as imagens disponibilizadas são demasiado pequenas para as propriedades do ecrã do dispositivo de visualização, as imagens são ampliadas e acabam por perder nitidez e prejudicar a experiência de utilização da aplicação.

2.2.3 Reconhecimento de Novos Métodos de Entrada

O desenho de interfaces do utilizador semelhantes à ilustrada pela Figura 2.1 dificulta a interação do utilizador com as Aplicações Web em dispositivos que usem, por exemplo, o *touch* como método de interação principal. Uma das razões mais comuns é os elementos de interação serem de pequenas dimensões e, por essa razão, de difícil seleção para o utilizador em dispositivos táteis. Para além disso, em dispositivos *touch*, não é possível oferecer um *feedback* intuitivo ao utilizador sobre os elementos da interface com os quais ele pode interagir como acontece, por exemplo, quando um elemento de texto é sublinhado quando é sobreposto pelo ponteiro do rato num computador pessoal.

Embora estas sejam duas das situações onde mais se evidencia a necessidade de se desenharem interfaces do utilizador para dispositivos móveis diferentes das que se desenhavam destinadas a dispositivos com rato e teclado, não são as únicas: grande parte dos dispositivos móveis, hoje, oferece vários outros métodos de *input*, alternativos aos tradicionais, que podem ser explorados como é o caso do giroscópio, do acelerómetro, do GPS, de sensores de proximidade, entre outros.

2.2.4 Funcionamento Offline

Sempre que, hoje em dia, abrimos ou atualizamos uma aplicação num navegador, o seu código-fonte e grande parte dos seus recursos são transferidos, mesmo que já o tenham sido anteriormente. Caso a ligação à Internet falhe, mesmo que o utilizador já esteja a navegar na aplicação, é comum a aplicação ficar completamente inutilizável ou até ser substituída por uma mensagem de erro do navegador, provocada pela falha na ligação. Em casos em que existam dados temporariamente guardados no *front-end* da aplicação, este tipo de falhas pode resultar na sua perda. Para além disso, a qualidade da ligação à rede tem, também, consequências diretas no desempenho das Aplicações Web e na qualidade da experiência de utilização que proporcionam ao utilizador final.

Em dispositivos móveis, a largura de faixa usada por uma aplicação é importante. É frequente os utilizadores destes dispositivos recorrerem a planos de dados móveis, que muitas vezes estão associados a um limite de tráfego por intervalo de tempo, para acederem às Aplicações Web. Para além da condicionante dos planos que impõem limites de tráfego ao utilizador, acresce a desvantagem de muitas ligações de rede sem fios serem instáveis e de baixa qualidade, por exemplo, quando sobrecarregadas com um grande número de utilizadores.

Em suma, os dispositivos móveis, sendo portáteis, deparam-se frequentemente com situações em que não é possível obter ligação à Internet, seja por dados móveis ou por outro tipo de rede sem fios. O funcionamento *offline* das Aplicações *Web* ganha portanto relevância com a introdução dos dispositivos móveis.

2.3 Complexidade de Desenvolvimento

Um problema atualmente bastante visível no desenvolvimento de Aplicações *Web* é a elevada complexidade, e consequente baixa legibilidade e difícil manutenção dos seus códigos-fonte.

O HTML é uma linguagem de anotação que permite obter resultados rápidos com pouco esforço na criação de aplicações que não necessitem de funcionalidades que não existam no seu catálogo de elementos nativos [RO15]. A customização das UI, feita com CSS, também pode ser uma tarefa simples para aplicações de baixa complexidade visual, mas criar fortes desafios, quando o objetivo é criar uma interface rica e modular e o código-fonte CSS precisar de ser gerido por uma equipa [Wal15].

Em suma, o JavaScript é uma linguagem dinâmica que oferece uma grande liberdade ao programador e que não impõe limites, dentro do contexto de uma Aplicação *Web*. No entanto, por ser uma linguagem extremamente tolerante a erros, é relativamente fácil o programador ser levado a situações de erro de difícil resolução ou criar respostas para problemas de programação que não sigam boas práticas de programação, resultando numa aplicação *Web* de difícil manutenção ou pouco escalável [Zak12].

2.3.1 Elementos Nativos de HTML5

É comum ver-se atualmente, no código-fonte de Aplicações *Web*, sequências de *tags* HTML cujo significado e a funcionalidade não são facilmente perceptíveis para o programador. Não o são, pelo menos, sem que o programador tenha um conhecimento prévio do código-fonte da aplicação ou que sem faça um estudo mais aprofundado desse código, na tentativa de deduzir a sua funcionalidade. A Figura 2.2 demonstra bem este problema na aplicação *Web* do Facebook.

A necessidade de implementar novas funcionalidades, que não as já asseguradas pelos elementos nativos da especificação do HTML5, leva o programador a este tipo de situações [RO15].

2.3.2 Regras de CSS

O Cascading Style Sheets (CSS) é global e as regras de estilo aplicadas a um elemento da UI de uma aplicação podem para afetar outros elementos de UI de diferentes componentes dessa aplicação.

Alterações feitas a *base rules*² e conflitos entre nomes de classes ou ids de CSS são dois exemplos de situações frequentes deste tipo. Para além disso, é ainda comum tentar contornar estes

²Regras de CSS constituídas por seletores de tipo, de atributos, de pseudo-classes, de filhos e de irmãos (não contêm classes nem IDs).

```
<html lang="en" id="facebook" class=" sidebarMode tinyViewport">
  <head>...</head>
  <body class="hasLeftCol home composerExpanded _5vb_ fbx _5p3y webkit safari mac x2 Locale_en_US" dir="ltr">
    <div id="pagelet_bluebar" role="banner" data-click="bluebar" data-click-phase="0">...</div>
    <div id="globalContainer" class="uiContextualLayerParent">
      <div class="fb_content clearfix" id="content" role="main">
        <div>
          <div id="toolbarContainer" class="hidden_elem"></div>
        <div id="mainContainer">
          <div id="leftCol">...</div>
          <div id="contentCol" class="clearfix hasRightCol homeWiderContent homeFixedLayout hasExpandedComposer newsFeedCompos...
            <div id="rightCol" role="complementary" aria-label="Reminders, people you may know, and ads">...</div>
            <div id="contentArea" role="main">
              <div id="stream_pagelet" data-referrer="stream_pagelet" data-click="Story" data-click-phase="0">
                <div id="pagelet_megaphone" data-referrer="pagelet_megaphone"></div>
                <div id="pagelet_composer" data-referrer="pagelet_composer" data-click="Composer" data-click-phase="1">
                  <div class="uiList _1ds1 _509- _4ki _6-h _6-j _6-i">
                    <li class="_4j _519b" data-ft="{"tn":"*t"}">
                      ...
                    </li>
                  </ul>
                </div>
              </div>
            </div>
          </div>
        </div>
      </div>
    </div>
  </body>
</html>
```

Figura 2.2: Excerto de código HTML da aplicação *Web* atual do Facebook

dois problemas recorrendo a regras com especificidades mais altas, com o objetivo de afetarem apenas elementos da UI que pertençam a um determinado ramo da árvore do *Document Object Model* (DOM). Porém, pode também acontecer que este tipo de regras acabem ainda por ter efeitos indesejados noutros elementos que pertençam ao mesmo ramo da árvore de DOM ou que dele descendam [Wal15].

Evitar este problema exige dos programadores um conhecimento profundo do código CSS desenvolvido e é uma tarefa ainda mais complicada na manutenção de uma aplicação de elevada complexidade desenvolvida por uma equipa com vários programadores ou *designers*.

2.3.3 “Pyramid of Doom” do JavaScript

O JavaScript é uma linguagem de programação que permite a implementação de funções *callback*³. É ainda possível recorrer ao uso de funções *callback* no corpo de outras funções *callback*. Por esta razão, são frequentes as situações em que várias camadas de funções deste tipo são usadas no código-fonte de Aplicações *Web*, dando origem a pirâmides de código JavaScript pouco legível e difícil de manter [Zim13]. A Figura 2.3 ilustra um exemplo de uma situação deste tipo, onde é feita uma interrogação a uma base de dados para se obterem as informações do país associado a um determinado utilizador.

2.4 Conclusões

As Aplicações *Web* tem vindo a evoluir no sentido de crescerem em número de funcionalidades e de serem capazes de dar à sua comunidade experiências de utilização mais ricas e mais próximas às oferecidas pelas aplicações nativas. No entanto, a sua evolução não tem acompanhado o rápido crescimento do seu ecossistema de dispositivos. Por esta razão, continuam a existir problemas sem

³Uma função *callback* é passada como argumento a outra função, que pode chamar, posteriormente, a função *callback*.

```
1 function getCountryByUserId(id, callback) {
2     openDatabase(function (db) {
3         db.getCollection('users', function (usersCollection) {
4             usersCollection.find({
5                 'id': id
6             }, function (userInfo) {
7                 db.getCollection('countries', function (countriesCollection) {
8                     countriesCollection.find({
9                         'countryId': userInfo.countryId
10                    }, function (countryInfo) {
11                        callback(countryInfo)
12                    })
13                })
14            })
15        })
16    })
17 }
```

Figura 2.3: Exemplo de *Pyramid of Doom* do Javascript

resolução, para quem pretende desenvolver Aplicações *Web* para os dispositivos mais recentes, que tirem o máximo partido das capacidades que oferecem.

O próximo capítulo descreverá o estado da arte de quatro tecnologias que se apresentam como propostas de resposta a alguns dos problemas que foram descritos neste capítulo.

Capítulo 3

Tecnologias Emergentes

Neste capítulo, será feita uma descrição do estado da arte de cada uma das tecnologias disponíveis que já implementam os conceitos *Web Components*, *Offline-first*, *Responsive Web Design* e *Single-Page Applications*.

3.1 Introdução

Nos últimos anos foram apresentadas várias tecnologias *Web* que oferecem respostas aos problemas descritos no capítulo anterior. Os *Web Components* prometem ser uma boa solução para combater a complexidade do código-fonte das Aplicações *Web* e estão já a ganhar forma nos padrões da *Web*, definidos pelo *World Wide Web Consortium*¹ (W3C); o conceito de *Offline-first* introduz a filosofia de que as Aplicações *Web* devem ser funcionais mesmo sem uma ligação constante à Internet e apresenta algumas tecnologias que contribuem para que o desenvolvimento para *Web* evolua nesse sentido; *Responsive Web Design* propõe respostas para os problemas relacionados com interfaces do utilizador; por último, as boas práticas de desenvolvimento de *Single-Page Applications* (SPA) comprometem-se a garantir uma melhor organização e qualidade global no código-fonte do *front-end* das Aplicações *Web*.

3.2 *Web Components*

O HTML é a linguagem de anotação que suporta a *World Wide Web*, desde o seu início. Um dos seus pontos fortes é ser uma linguagem declarativa e de uso simples. Porém, por oposição, uma das suas desvantagens é ser obrigada a servir-se de como o JavaScript para colmatar as suas limitações e conseguir desempenhar tarefas mais complexas.

¹Organização internacional responsável por definir as principais convenções que uniformizam a *World Wide Web*

Os *Web Components* procuram tirar proveito da simplicidade de desenvolvimento de aplicações com programação declarativa e trazer funcionalidades mais complexas para a parte declarativa do desenvolvimento das Aplicações *Web*.

3.2.1 Linguagens Declarativas nas Aplicações *Web*

As linguagens declarativas têm um papel importante na simplificação do processo de desenvolvimento de Aplicações *Web*. Na maior parte dos casos, este tipo de linguagens é usado para a construção das interfaces do utilizador, tanto em Aplicações *Web* como em aplicações nativas de outras plataformas — iOS e Android, por exemplo. O código das linguagens declarativas tem uma relação bastante próxima do seu resultado. No contexto de desenvolvimento de UI, o programador declara no código os componentes da UI que pretende usar na interface e o interpretador da linguagem é o responsável por perceber como há-de construir os resultados esperados para o código desenvolvido. Em contraste, o uso de linguagens imperativas para este fim implica que o programador especifique detalhadamente os passos necessários para que os componentes de UI sejam construídos [Rey14].

3.2.2 *Custom Elements*

Vários elementos nativos de HTML5 são dotados de funcionalidades, por oposição a apenas produzirem um resultado visual predefinido. É o caso do elemento “img”, que se encarrega de desenhar uma imagem numa Aplicação *Web*, indicada num atributo onde é definido o caminho para a localização na rede do recurso de imagem. Existem outros elementos que assumem as suas próprias funcionalidades, como os elementos “video”, “select” e “form”. Contudo, com a rápida expansão da *World Wide Web*, se antes existiam elementos dedicados à inserção de imagens, vídeos, tabelas, entre outras que implementavam funcionalidades comuns à grande parte das aplicações desenvolvidas na altura, hoje espera-se que as Aplicações *Web* sejam capazes de desempenhar funcionalidades mais complexas. Por esta razão, o catálogo de elementos nativos oferecidas pela especificação do HTML5 começa a não ser suficiente.

Com a introdução dos *Custom Elements*, passa a ser possível criar novos elementos HTML com uma *tag* e um comportamento próprio. Os novos elementos criados podem ainda basear-se nouros elementos, sejam eles nativos ou *Custom Elements*, de forma a herdarem as suas propriedades e funcionalidades.

Reutilização e Manutenção de Componentes

Os *Custom Elements* dão um forte contributo para a reutilização de código na *Web*. A declaração de um novo *Custom Element* só precisa de ser feita uma vez, momento a partir do qual podem ser criadas, declarativamente, instâncias no código-fonte da aplicação. Para além disso, essas instâncias irão respeitar o comportamento implementado na declaração do componente e as atualizações futuras serão refletidas em todas as suas instâncias. O resultado será uma manutenção

mais simples das funcionalidades acrescentadas a uma aplicação *Web* pelo componente hipotético em questão.

É ainda possível transmitir nos nomes das *tags* de novos componentes um valor semântico que identifique intuitivamente a funcionalidade desempenhada por cada componente. Desta forma, a utilização de *Web Components* pode ainda ser uma solução para melhorar a legibilidade de códigos-fonte como o ilustrado pela Figura 2.2 [Bid13a].

3.2.3 HTML Imports

Sem se recorrer a HTML *Imports*, não existem muitas formas de se reutilizar em HTML outras páginas HTML, não excluindo as suas dependências. Uma forma seria recorrer ao elemento de HTML “*iframe*”, que cria alguns obstáculos ao uso JavaScript e de CSS nas páginas importadas. Outra alternativa seria recorrer a AJAX que, embora resolvesse os problemas levantados pelo “*iframe*”, envolveria o recurso a JavaScript.

Os HTML *Imports* trazem à *Web* uma forma simples de se fazer a reutilização de páginas HTML. Para além disso, permitem ainda a reutilização de HTML proveniente de endereços externos que incluem a declaração de *Custom Elements*. Desta forma, o uso de HTML *Imports* complementam os *Custom Elements*, promovendo a reutilização de código no desenvolvimento de Aplicações *Web* [Bid13b].

3.2.4 HTML Templates

Antes do aparecimento dos *Web Components*, já vários *frameworks* recorriam ao uso de *templates* HTML. Recorrendo ao uso de *templates*, durante o desenvolvimento de uma aplicação *Web*, é possível declarar blocos de código HTML que podem, mais tarde, ser instanciados e reutilizados na aplicação. Ruby on Rails, Smarty e Django são alguns exemplos de *frameworks* que recorrem ao uso de *templates* HTML. Porém, atribuem ao servidor a responsabilidade de construir o resultado visual das aplicações e, cada vez mais, começam a ser vistas aplicações que seguem um paradigma diferente — a responsabilidade da criação da UI passa para o lado do cliente. O recurso ao uso de HTML Templates, processados no navegador, ganha assim mais relevância.

Alternativas

Já existiam soluções capazes de oferecer respostas a este problema antes dos HTML *Templates*. Uma delas seria declarar *templates* em HTML, dentro de uma *tag* “*div*”, invisível com o auxílio de código CSS, o que levaria a que o conteúdo do *template* declarado fosse carregado para o DOM mesmo em situações onde não estivesse a ser usado.

Outra opção seria recorrer a uma *tag* “*script*”. O conteúdo do *template* permaneceria invisível e seria interpretado pelo navegador como uma *string*, não sendo carregado desnecessariamente para o DOM. No entanto, embora o conteúdo do *template* não fosse carregado, a *tag* “*script*” que

o contivesse iria ser aceitável através do DOM e facilmente manipulável com o recurso a JavaScript, aumentando a probabilidade de existirem vulnerabilidades de *Cross-Site Scripting* (XSS) nas aplicações desenvolvidas recorrendo a esta técnica de *templating* [Bid13c].

Vantagens

Com o aparecimento dos HTML *Templates*, foi adicionada à especificação do HTML5 a tag “template”, que veio solucionar os problemas dos *hacks* anteriores. A renderização de um *HTML Template* só é feita no momento do seu uso e o acesso ao seu conteúdo só é possível se o *template* estiver a ser usado [Kit14].

3.2.5 Shadow DOM

O DOM de uma página *Web* está organizado hierarquicamente numa estrutura em árvore. O elemento “document” representa a raiz da árvore e todos os restantes elementos que constituem a página descendem desse elemento. As regras de estilo CSS usam seletores que especificam os elementos, pertencentes à árvore do DOM, aos quais os estilos definidos irão ser aplicados. Porém, os seletores apontam para um padrão na hierarquia dessa árvore e é frequente os padrões definidos nas regras verificarem-se em mais do que um sítio da árvore do DOM. Para além disso, as regras de CSS têm a característica de serem globais e de, por essa razão, poderem provocar alterações ao longo de toda a árvore DOM, alterações essas que, por vezes, podem ser indesejadas, subtis e difíceis de detetar.

Encapsulamento de Estilos

O *Shadow DOM* traz o conceito de encapsulamento para os *Web Components* e estabelece uma barreira dentro da hierarquia do DOM. Com o *Shadow DOM* passa a ser possível atribuir a elementos do DOM um contexto próprio que não seja afetado por regras de estilo que lhe sejam exteriores [Coo13].

Recorrendo às tecnologias *Custom Elements* e *HTML Imports* passa a ser relativamente simples importar elementos desenvolvidos por outros programadores. Esses componentes terão o seu próprio estilo encapsulado com *Shadow DOM* e não dependerão das regras de CSS impostas pelos restantes componentes da aplicação.

3.2.6 Bibliotecas e *Frameworks* Disponíveis

Existem vários *frameworks* que implementam algumas das ideias suportadas pelos *Web Components*. São dois exemplos o AngularJS 1.x², desenvolvido pela Google, e o React³, pela Facebook Inc. Porém, as suas implementações não têm ainda por base os padrões de *Web Components* especificados pelo W3C.

²<https://angularjs.org/>

³<http://facebook.github.io/react/>

Existe, no entanto, uma biblioteca que implementa as quatro tecnologias dos *Web Components* e que respeita os *standards* do W3C: a biblioteca Polymer⁴, da Google.

Polymer

A biblioteca Polymer foca-se na implementação de *Web Components* e é constituída por três camadas fundamentais:

1. *Web Components*: Nesta camada, o Polymer disponibiliza um conjunto de API — *polyfills* — que emulam o suporte para *Web Components* em navegadores que ainda não os suportam nativamente. Durante a execução da aplicação, o Polymer recorre à implementação nativa de *Web Components* nos navegadores que incluem o seu suporte ou recorre a esta camada de *polyfills*, caso o navegador não suporte estas tecnologias nativamente;
2. Biblioteca do Polymer: Esta biblioteca oferece uma sintaxe alternativa, mais simples e que envolve a criação de menos código, para a declaração de *Web Components*. Para além disso, são ainda implementadas, nesta camada, outras funcionalidades úteis no desenvolvimento do *front-end* de Aplicações *Web* como *data-binding* e o suporte a eventos *touch*;
3. *Elements*: O Polymer oferece uma camada com vários *Custom Elements* já implementados, que prevêm algumas das funcionalidades mais comuns nas Aplicações *Web*. Esta camada inclui a implementação dos *Core elements*, que permitem criar declarativamente desde pedidos HTTP a um servidor remoto, a animações e transições entre páginas e eventos de métodos de entrada como o *drag and drop*, e dos *Paper elements*, que incluem alguns elementos da interface do utilizador que seguem os princípios do Material Design (Secção 3.4.3) da Google [Pol14b].

3.3 Offline-first

Do ponto de vista do utilizador de um sistema hipotético, é importante que o sistema sirva o propósito para que foi criado e que satisfaça as necessidades do seu utilizador. No mundo da *Web*, a dificuldade de se estabelecer uma ligação à Internet de boa qualidade pode impedir a concretização dessas duas premissas. Este é um problema frequente para grande parte dos utilizadores da *Web* nos dias de hoje.

O conceito de *Offline-first* surge com o objetivo de dar resposta aos problemas relacionados com ligações à Internet de baixa qualidade e põe em primeiro plano as necessidades do utilizador. Espera-se que o tempo resposta do servidor deixe de ser um problema para o utilizador e que as Aplicações *Web* passem a poder ser usadas *offline*, à semelhança do que acontece com as aplicações nativas [Jak12].

⁴<https://www.polymer-project.org/0.5/>

3.3.1 Armazenamento de Dados

A existência de uma base de dados do lado do cliente é importante. A informação deve ser sempre guardada localmente para que, caso hajam falhas na ligação à Internet, a aplicação possa continuar a ser funcional e a informação retida possa continuar a estar acessível. Existem algumas APIs disponíveis que simplificam o processo de armazenamento local da informação: HTML5 *Web Storage*, IndexedDB e WebSQL são as principais. Porém, existem outras, que expandem a funcionalidade destas 3 e que abstraem a complexidade do seu uso em tarefas mais complexas, como é o caso das bases de dados de *front-end* PouchDB⁵ e Minimongo⁶.

Não deve, também, ser excluída a existência de um meio de armazenamento remoto, mantido em constante sincronização com o meio de armazenamento local. Já existem tecnologias que abstraem o programador da complexidade da sincronização de dados entre os dois locais de armazenamento [Wei15]. O PouchDB, por exemplo, foi desenvolvido com o objetivo de oferecer a sincronização *out of the box* de dados com a base de dados de *backend* CouchDB. Da mesma forma, o Minimongo foi desenhado com o objetivo de oferecer a mesma funcionalidade de sincronização, mas com a base de dados de *backend* MongoDB.

3.3.2 Cache do Código-fonte

Para que uma aplicação *Web* possa ser considerada uma aplicação *offline*, também os seus ficheiros executáveis devem ser armazenados *offline*, de forma a que a aplicação possa, em qualquer altura, executar sem a necessidade do utilizador estar ligado à rede. Existem duas tecnologias, que incorporam os *standards* da *Web*, que tornam isso possível: *Application Cache* e Service Worker.

Application Cache

HTML5 *Application Cache* é uma tecnologia que permite declarar facilmente quais os ficheiros que devem ser armazenados em *cache* no navegador. No entanto, a sua simplicidade vem associada a algumas limitações significativas:

1. Os recursos deixam de receber atualizações remotas a partir do momento em que são armazenados em *cache* — o que implica que uma aplicação deixará de receber atualizações a partir do momento em que é instalada no navegador;
2. A única forma de comunicar ao navegador alterações na aplicação é alterando o conteúdo do ficheiro onde é declarada a lista de recursos que devem ser guardados em *cache* — como consequência, a *cache* antiga será apagada e a aplicação será novamente transferida e armazenada integralmente [Fey13].

⁵<http://pouchdb.com/>

⁶<https://www.omniref.com/js/npm/minimongo/2.3.2>

Service Worker

O *Service Worker* foi criado com o objetivo de resolver os problemas do *HTML5 Application Cache* e de dar mais liberdade aos programadores sobre a forma como tratam a comunicação entre as Aplicações Web, a *cache* dos navegadores e a rede.

Os *Service Workers* são executados no navegador numa *thread* independente daquela em que corre a aplicação Web a que correspondem. Para além disso, funcionam como *proxys* de rede e permitem interceptar e manipular pedidos de rede feitos pela aplicação à rede remota. Desta forma, e por a sua funcionalidade ser da responsabilidade do programador, é possível implementar comportamentos como, por exemplo, especificar dinamicamente quais os recursos da aplicação que devem ser armazenados em *cache* na primeira vez em que a aplicação é executada ou declarar que pedidos feitos a um determinado domínio devem ser satisfeitos com uma resposta remota. É ainda possível implementar comportamentos mais complexos como, por exemplo, definir que pedidos devem ser satisfeitos com recursos armazenados em *cache* ou executados remotamente, se ainda não existirem em *cache*, e armazenar o seu resultado para dispensar o uso da rede em próximas ocasiões [RSA15].

3.3.3 NoBackend

O conceito de *Offline-first* surge muitas vezes a par com outro conceito novo na Web — *No-Backend* — embora os dois não se devam confundir.

Embora continuem a existir plataformas que exigem a implementação de um *backend* dedicado, muitas são as aplicações que implementam funcionalidades semelhantes ao nível do *frontend*: registo, autenticação, armazenamento de dados, *chat*, email, etc. O conceito de *NoBackend* vem tentar simplificar o processo de desenvolvimento de aplicações deste tipo, tirando a tarefa da implementação dos seus *backends* da responsabilidade dos programadores.

O objetivo desta abordagem é permitir aos engenheiros de *software* focarem-se ao máximo na funcionalidade e no resultado final esperado das aplicações, ignorando as exigências do ponto de vista do *backend*. Questões como a escalabilidade da aplicação ou a disponibilidade dos servidores ou até mesmo com questões de segurança deixam de ser preocupações para os programadores das Aplicações Web [Heh14].

O Hoodie e o Parse, descritos na Secção 3.3.4, são dois exemplos de *frameworks* que introduzem este conceito. Estas *frameworks* disponibilizam uma API em JavaScript que reduz todas as tarefas de *backend*, sejam elas a adição de dados, sincronização, entre outras, a funções de JavaScript de alto nível que permitem ao programador abstrair-se de todo o processo [Hem13].

3.3.4 Frameworks

Existem já *frameworks* que implementam as ideias do *Offline-first*. Nesta secção será feita uma descrição de três: Hoodie, Meteor e Parse.

Hoodie

O Hoodie é um *framework open source* dedicada ao desenvolvimento de Aplicações *Web* e móveis com uma arquitetura baseada nos conceitos de *NoBackend* e de *Offline-first*. Foi desenvolvida com JavaScript e NodeJS e recorre ao HTML5 *Web Storage* e à base de dados NoSQL CouchDB para armazenar dados localmente e no servidor, respetivamente.

Disponibilizando uma API em JavaScript de alto nível para gerir e simplificar todos os processos da lógica de negócio das Aplicações *Web* e móveis, o Hoodie permite criar aplicações *offline-first*, com sincronização com um *backend* remoto. Permite ainda a adição de *plugins* de NodeJS⁷ ao *backend*, que lhe conferem funcionalidades mais complexas como o envio de uma mensagem de correio eletrónico ou o processamento de uma transação monetária, por exemplo.

Quando uma ação é espoletada pelo utilizador, através de uma chamada da API do Hoodie, a ação vai ser processada pelo motor do Hoodie no *front-end* e armazenada na *Web Storage* do navegador. De seguida, é verificada a existência de uma ligação à Internet e, quando a ligação for bem sucedida, a ação é enviada pela rede ao *backend* do Hoodie e armazenada numa base de dados CouchDB. Caso não exista ligação, o Hoodie tentará fazer a sincronização noutra altura. Quando a informação estiver presente na base de dados do *backend*, caso deva ser processada por um *plugin* de NodeJS, a ação é transmitida pela base de dados ao *plugin* para ser concretizada [Hoo14b].

Meteor

O Meteor é um *framework full-stack*⁸ que, ao contrário do Hoodie — que ganha na simplicidade —, disponibiliza diversas APIs para a criação de *templates* HTML, para gerir sessões, para trabalhar com dados nas bases de dados locais e remotas e para outras necessidades das Aplicações *Web* dos dias de hoje.

Uma particularidade desta *framework* é a integração do conceito de “Programação Reativa”. O sistema de *templating* de *front-end* usado no Meteor estabelece um *binding* direto entre os elementos da UI das aplicações Meteor e os seus modelos de dados e, como consequência, alterações feitas ao nível do modelo de dados são refletidas na UI em tempo real.

Tal como o Hoodie, o Meteor serve-se de duas bases de dados NoSQL — MongoDB — para o lado do cliente (Minimongo) e para o *backend*. Tal como acontece com o Hoodie, a API disponibilizada para o tratamento de dados nas bases de dados é de alto nível, simples de usar e implementada em JavaScript. O Meteor não dispensa, no entanto, da implementação do *backend* das aplicações, embora as API isomórficas que são disponibilizadas pelo *framework*, que tornam possível que o mesmo código JavaScript possa ser corrido tanto no cliente como no servidor, tenham o propósito de ajudar na sua implementação [Met13b].

⁷<https://nodejs.org/>

⁸Framework que auxilia o desenvolvimento de uma aplicação em todas as suas camadas, desde a responsável pelo tratamento de dados até à da interface.

Parse

O Parse é um *framework* que faz parte do património da Facebook Inc. Insere-se num tipo diferente e mais específico de *frameworks NoBackend — Backend as a Service*. Funcionalidades como notificações Push e associar GeoPontos a dados guardados na base de dados são apenas algumas das funcionalidades que o Parse oferece. Para além disso, permite a implementação de funções ao nível do servidor — *cloud functions* — que podem ser usadas para processamento de tarefas que possam exigir demasiado da capacidade de processamento de um *smartphone* ou até de um computador pessoal. O Parse é, portanto, um *framework Backend as a Service* bastante completo.

O Parse não é um serviço gratuito. Os seus custos dependem do tipo de funcionalidades e da largura de faixa usadas. Contudo, também o desenvolvimento do *backend* para uma aplicação, seja ela *Web* ou não, tem custos que, em muitos casos, assumem dimensões que justificam o recurso a plataformas como o Parse [Par12b].

3.4 *Responsive Web Design*

Ethan Marcotte, um *web developer* e *web designer* com um interesse paralelo em arquitetura, descreve, num dos seus livros, um tipo de arquitetura — *Responsive Arquitecture* — que diz ter sido uma das inspirações para a sua introdução do conceito de *Responsive Web Design*. Edifícios com estruturas móveis e paredes com cores alternáveis foram alguns dos conceitos que despertaram o seu interesse e que o inspiraram a querer trazer esse conceito para o mundo da *Web*. A ideia seria oferecer aos utilizadores da *Web* um *design* adaptável ao contexto de visualização de cada utilizador [Mar10].

3.4.1 Princípios de Design

Alguns princípios de *design* associados ao desenvolvimento de aplicações para a *Web* que incorporem este conceito são:

- Criar *layouts* adaptáveis a ecrãs de todos os tamanhos, desde ecrãs *widescreen* de alta resolução a ecrãs de pequenas dimensões como os de grande parte dos *smartphones*;
- Redimensionar imagens de forma a manter a sua legibilidade em ecrãs pequenos [Mar11];
- Utilizar imagens, nos dispositivos móveis, que necessitem de uma largura de faixa menor [LeP14];
- Simplificar a estrutura das páginas nos dispositivos móveis;
- Omitir elementos desnecessários em ecrãs de pequenas dimensões;
- Aumentar o tamanho das componentes da UI nos dispositivos móveis para a navegação em dispositivos táteis ser mais prática [Yat12];

- Aproveitar algumas funcionalidades dos dispositivos móveis, como a Geolocalização e a orientação do dispositivo, para adequar a experiência de utilização das aplicações a cada contexto de utilização.

3.4.2 *Mobile first*

Mobile first é uma filosofia que se tem tornado popular nos últimos tempos, desde que o conceito de *Responsive Web Design* foi introduzido por Marcotte, em 2010. De uma forma suínta, consiste em desenvolver aplicações primeiro para os dispositivos móveis e só então fazer as adaptações necessárias para que elas corram também corretamente em dispositivos com ecrãs maiores [Yat12].

Os dispositivos móveis têm, de uma forma geral, menos capacidade de processamento e ecrãs com dimensões inferiores às dos computadores portátiles ou fixos e, como consequência, o *software* desenvolvido para esses dispositivos tem algumas limitações. Da mesma forma, a maioria dos navegadores disponíveis para as plataformas móveis não têm um leque de funcionalidades tão vasto e rico como os navegadores para computadores pessoais. Desta forma, se desenvolvemos uma aplicação a pensar só no seu resultado quando for utilizada através de um computador fixo e, mais tarde, a quisermos adaptar aos dispositivos móveis, é provável que sejamos obrigados a remover algumas funcionalidades e a ter o trabalho complicado.

Por oposição, focar o desenvolvimento das Aplicações *Web*, numa primeira fase, na sua correta visualização e numa boa experiência de utilização em dispositivos móveis, garantindo que apenas são usadas as tecnologias disponíveis e compatíveis com a maior parte dos navegadores mais usados nesses dispositivos, e só mais tarde fazer a sua adaptação a dispositivos de maiores dimensões, deverá resultar em aplicações com uma boa experiência de utilização em todas as plataformas [Mar10].

3.4.3 Material Design

O Material Design é um projeto, desenvolvido pela Google, que procura reunir um conjunto de boas práticas de *design*, respeitar os princípios do *Responsive Web Design* e ainda simplificar e unificar o processo de desenvolvimento da parte visual das Aplicações *Web* para todo o tipo de dispositivos. Na Figura 3.1 pode ver-se a UI de uma aplicação *Web* desenvolvida com Material Design, à esquerda vista num *smartphone* e à direita num computador pessoal.

O Material Design recorre a características como a luminosidade e sombras e a movimentações das superfícies que compõem a aplicação para ajudar o utilizador a perceber de que forma essas superfícies se relacionam entre si, num espaço tridimensional, e para evidenciar formas de interação entre o utilizador e a aplicação. Ainda no Material Design, elementos como a tipografia, grelhas, espaço em branco, dimensões, cores e imagens e vídeo são tidos em consideração no sentido de dar significado aos componentes visuais das Aplicações *Web*, de criar uma hierarquia e de manter o utilizador focado no que é essencial e, desta forma, tornar o uso destas aplicações o mais intuitivo possível [Mat14].

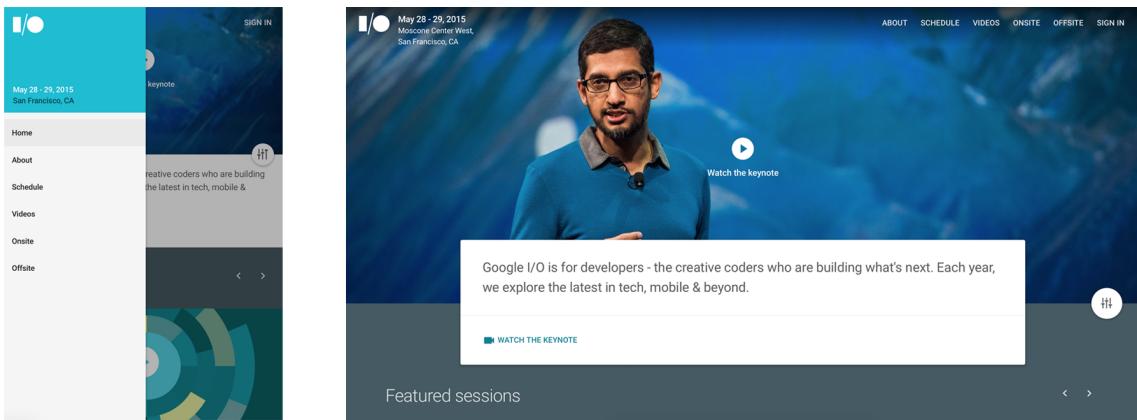


Figura 3.1: UI de aplicação Web com Material Design

3.4.4 Desenvolvimento de Interfaces do Utilizador

Desde o aparecimento do HTML, houve várias propostas de linguagens de anotação para personalizar o aspetto das páginas Web. As poucas alterações que podiam ser feitas nessa altura eram introduzidas diretamente no código HTML e eram bastante simples. Escolher os tipos de letra usados numa página ou alterar cores e tamanhos de letra são alguns exemplos do que podia ser feito. Porém, à medida que o HTML foi evoluindo, foi surgindo a necessidade de se criar um método melhor para desenvolver páginas Web com UI personalizadas e, em 1996, o *World Wide Web Consortium* acrescentou a linguagem CSS aos *standards* da Web. Com a introdução do CSS, passou a ser possível fazer esta personalização em ficheiros de código fonte independentes dos ficheiros HTML. Contudo, as alterações que podiam ser feitas com as primeiras versões do CSS eram ainda bastante limitadas [Fan11].

A versão 3 do CSS foi a primeira a considerar já o conceito de responsividade e a permitir criar *layouts* adaptáveis a diferentes resoluções e a ecrãs de diferentes dimensões, possibilitando a criação de grelhas, imagens flexíveis e *media queries*. Para além disto, a versão 3 do CSS passou a incluir algumas funcionalidades que viriam a melhorar consideravelmente a usabilidade das Aplicações Web como a introdução de sombras e de CSS Animations [Sam13].

Syntactically Awesome Style Sheets

As *Syntactically Awesome Style Sheets* (Sass) constituem uma linguagem que aumenta a funcionalidade do CSS, adicionando-lhe algumas propriedades que simplificam e que tornam mais organizado o processo de personalização visual das Aplicações Web. O código desenvolvido em Sass precisa, no entanto, de ser compilado para código CSS para poder ser usado na Web e reconhecido pelos navegadores. Em Sass existem variáveis; é possível usar seletores dentro de outros seletores — *nesting* —, evitando a necessidade de repetir o seletor mais genérico para cada regra aplicada a outro seletor que dele dependa; é possível criar ficheiros Sass parciais, com partes de código que podem ser posteriormente reutilizadas por ficheiros de código diferentes; existem *mixins*,

que permitem agrupar várias linhas de Sass numa para facilitar a sua reutilização; existe polimorfismo, sendo possível criar seletores que aumentam a funcionalidade doutros mais genéricos; e é possível ainda trabalhar com valores com os operadores matemáticos de adição, subtração, multiplicação, divisão e de cálculo do resto. Para além disso, o Sass pode ainda ser usado em conjunto com código CSS simples [Fra13].

Framework Compass

O Compass é um *framework* baseada em Sass que resolve alguns dos problemas mais comuns existentes no CSS3 e otimiza o desenvolvimento da componente visual das Aplicações Web. Tal como na linguagem Sass, o Compass permite usar variáveis, *mixins*, fazer *nesting* de seletores e realizar cálculos matemáticos. Quando usamos CSS3 ou Sass, é frequente sermos obrigados a escrever várias linhas de código que produzem o mesmo efeito visual, mas que se aplicam a navegadores diferentes. O Compass abstrai este problema e assegura que todo o código produzido é compatível com todas as plataformas desejadas. Para além disso, o Compass permite definir ainda propriedades como as cores de uma hiperligação, por exemplo, para todos os estados, recorrendo apenas a uma linha, ou definir as propriedades CSS de uma lista HTML usando, da mesma forma, apenas uma linha de código da API do Compass. É ainda possível usar *plugins* adicionais para usar funcionalidades que não são disponibilizadas por omissão na API da *framework* como sistemas de grelhas responsivas, *widgets*, esquemas de cores, entre outras [Fra13].

3.4.5 Web Animations API

Existem pelo menos 4 formas de se fazerem animações na Web:

- CSS *Transitions* e CSS *Animations* — embora consumam poucos recursos e sejam rápidas, construir animações através do CSS tem algumas limitações, não permitindo fazer animações mais complexas e sequenciais ou construir animações compostas por várias animações; para além disso, este tipo de animações têm de ser implementadas ao nível do CSS e não podem ser customizadas com JavaScript;
- CSS *Animations* — embora permita fazer animações com um maior nível de complexidade, as CSS *Animations* são pesadas para os navegadores, difíceis de construir e não permitem animar elementos HTML;
- O método “requestAnimationFrame” — exige o uso da *thread* principal onde correm os eventos de *front-end* e, caso essa *thread* esteja a ser usada, as animações não vão ser ativadas de imediato.

As *Web Animations* são uma alternativa e uma resposta às limitações destas 4 tecnologias já existentes.

De uma maneira geral, os navegadores mais recentes recorrem a 2 *threads* para fazerem o *render* de uma página Web. Uma delas, a *thread* principal, é normalmente a responsável por correr o

código JavaScript da página, por processar os estilos CSS associados a essa página, por converter o resultado visual dos vários componentes da página em imagens e por os enviar para a segunda *thread*, que se irá ocupar de apresentar essas imagens ao utilizador. Grande parte das tarefas encaimhadas para esta segunda *thread* são processadas em paralelo, ao nível do *Graphics Processing Unit* (GPU) e o seu desempenho não é afetado pelo que possa estar a acontecer no *Central Processing Unit* (CPU). Para além disso, as tarefas de desenhar imagens no ecrã, seja em diferentes sítios do ecrã, em diferentes ângulos ou em diferentes tamanhos, são extremamente rápidas, quando processadas ao nível da GPU. Tal como as CSS *Transitions* e as CSS *Animations*, também as *Web Animations* tiram partido do uso desta segunda *thread* e do poder de processamento da GPU para otimizarem as suas animações e, desta forma, permitem criar animações tão eficientes como as implementadas em CSS, exclusivamente com o recurso a JavaScript, e com um maior grau de complexidade [Max14].

3.4.6 *Responsive Images*

Já vimos que, com a introdução do CSS3, passou a ser possível redimensionar, reposicionar e ajustar os *layouts* de uma Aplicação Web a ecrãs de diferentes tamanhos. Porém, o conceito de *Responsive Web Design* não se limita aos *layouts*. É importante ter-se também em atenção as dimensões de conteúdo como imagens e vídeos nas Aplicações Web e tentar assegurar que os recursos usados numa aplicação num telemóvel têm as dimensões adequadas ao dispositivo. Da mesma forma, deve-se procurar garantir que uma imagem que deverá ocupar completamente um ecrã com uma resolução de alta definição não seja a mesma do que a carregada pela mesma aplicação num telemóvel. Estas considerações deverão ser tidas em conta, no sentido de se prevenir a deformação desses recursos em qualquer um desses monitores, bem como de garantir que recursos com definição demasiado alta não são carregados e não gastam largura de faixa desnecessária em dispositivos que também não o exigem. As *Responsive Images* surgem portanto como uma possível resposta a este problema.

3.5 *Single-Page Applications*

Com o aparecimento gradual de dispositivos com maior poder de processamento, do aumento da probabilidade de ocorrência de falhas nas ligações à Internet, com o aparecimento das redes sem fios e, mais tarde, com a explosão tecnológica que deu origem aos dispositivos móveis, passou não só a ser possível passar para o *front-end* das Aplicações Web algumas operações mais complexas de processamento, mas também a ser mais importante dar resposta às limitações impostas pela Web estática, pobre na interatividade e na experiência utilização que proporcionava ao utilizador final.

Aumentar o nível de interatividade, de cooperação e, como consequência, tornar a Web numa plataforma mais dinâmica foram algumas das ideias introduzidas com a Web 2.0. Foi também com o aparecimento da Web 2.0 que tecnologias como código JavaScript assíncrono e linguagens de anotação como o XML e o JSON começaram a ser cada vez mais faladas. O desenvolvimento deste

tipo de tecnologias foi o primeiro passo dado no sentido de possibilitar a criação de Aplicações Web dinâmicas, mais eficientes na troca de dados entre o *front-end* e o servidor, mais ricas no que toca à experiência de utilização proporcionada ao utilizador final e com mais funcionalidades e maior complexidade ao nível do *front-end*. Este tipo de aplicações são conhecidas por *Single-Page Applications* (SPA) ou *Single-Page Interfaces* e vieram revolucionar a forma como as Aplicações Web são estruturadas, tanto ao nível do *backend* e como do *front-end*.

Um exemplo bastante conhecido de uma aplicação deste tipo é a aplicação Web do serviço Gmail — Figura 3.2.

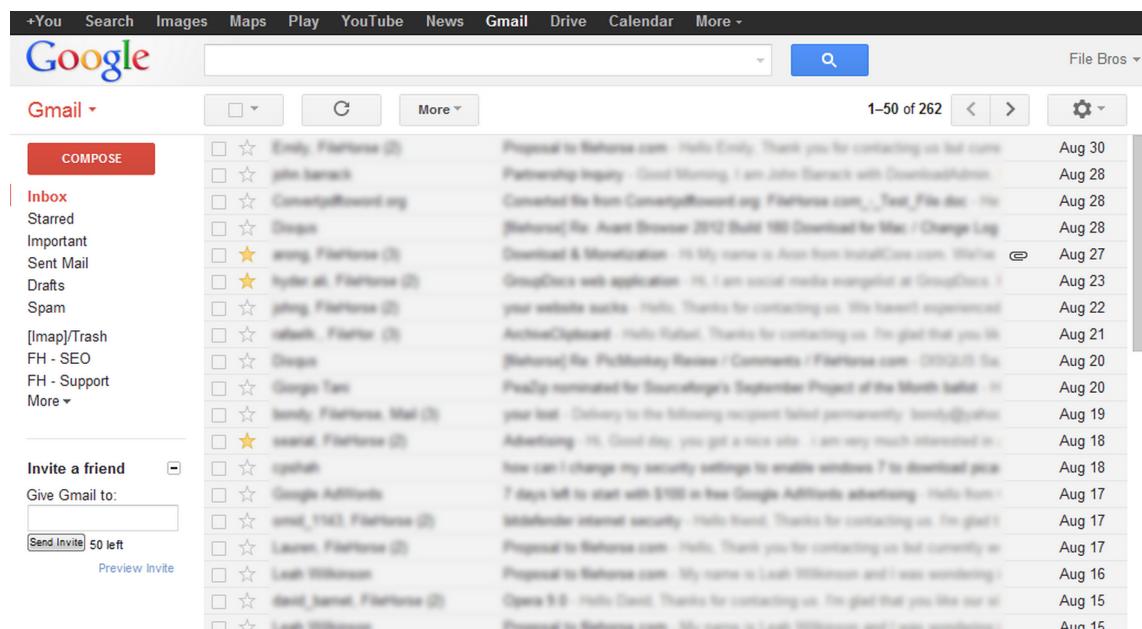


Figura 3.2: Captura de ecrã da aplicação Web do serviço Gmail

3.5.1 Architectural Patterns

A implementação de *Architectural Patterns* é naturalmente importante no desenvolvimento de *Single-Page Applications*, uma vez que promove a separação de código por partes lógicas. Em aplicações complexas, garantir uma estrutura modular e um alto nível de separação entre as componentes lógicas responsáveis pela UI, tratamento de dados e estado da aplicação e pela lógica de negócio facilita o desenvolvimento em equipa, a implementação de testes unitários a componentes distintos da aplicação e, mais tarde, a manutenção do código desenvolvido.

No entanto, é importante referir que um dos pontos-chave focados nesta dissertação é a implementação de *Single-Page Applications* aliadas a uma arquitetura estruturada por componentes — *Web Components*. Um dos principais objetivos do uso de *Web Components* é promover a reutilização de componentes na comunidade da Web. Para isto, é importante que os componentes desenvolvidos possam ser usados em qualquer contexto e que tenham as suas propriedades e a sua

funcionalidade encapsulada na sua declaração. Como consequência, é esperado que cada componente encapsule também todas as componentes lógicas que implementem a sua funcionalidade — UI, tratamento de dados, estado e lógica de negócio.

3.5.2 *Design Patterns Emergentes*

Design patterns são boas práticas de resolução de problemas que podem ser aplicadas a um grupo de problemas semelhantes, ao longo do desenvolvimento de um aplicação, e o seu uso recorrente resulta, normalmente, na produção de código mais robusto, organizado e legível.

Existe uma vasta lista de *design patterns* de JavaScript que podem ser usados e trazer benefícios ao desenvolvimento de Aplicações *Web* em diversas situações. No contexto das *Single-Page Applications*, porém, há alguns que têm particular interesse e que serão descritos a seguir [Osm12].

Module Pattern

O *Module Pattern* oferece uma forma de criar métodos públicos e privados na linguagem. Este é um *design pattern* importante na implementação de componentes em SPA, uma vez que, por trazer o conceito de encapsulamento de métodos para o JavaScript, impede que esses métodos entrem em conflito com métodos de outros componentes do sistema e concedem ao código um maior nível de organização, tornando possível incorporar funcionalidades semelhantes num mesmo módulo e esconder e separar toda a sua lógica dos restantes componentes da aplicação.

Prototype Pattern

É usado para implementar polimorfismo, em JavaScript. Permite, portanto, criar objetos com as mesmas propriedades de outros objetos e adicionar-lhes novas propriedades e funcionalidades. Todos os métodos implementados no objeto-pai são referenciados no objeto descendente, sendo que, quando é chamado um desses métodos no objeto-filho, é chamada a implementação desse método no objeto-pai.

Observer Pattern

O *Observer Pattern* é usado, no desenvolvimento de SPA, para implementar o conceito de *data-binding*. Na sua implementação, os objetos que dependam dos dados armazenados por outro objeto são adicionados a uma lista guardada neste último e, sempre que esses dados sofram alterações, esse objeto irá transmitir o novo estado a todos os elementos que constem nessa lista, em tempo real.

Composite Pattern

Um *design pattern* que declara que um grupo de objetos pode ser tratado de forma igual a qualquer um dos elementos que pertencem a esse grupo. É usado na criação de *Web Components*

e permite que se recorra a vários componentes que resolvem pequenos sub-problemas para criar um componente que resolva um problema maior [Mit13].

Mediator Pattern

Design pattern usado para implementar mecanismos de comunicação entre componentes sem que cada um dos intervenientes necessite de conhecer o funcionamento dos restantes. Num exemplo simples de comunicação entre 2 elementos A e B, é boa prática o elemento que inicia a comunicação, A, disparar um evento que é apanhado por um terceiro elemento, C — o mediador —, que terá a responsabilidade de o reencaminhar para o elemento de destino, B.

Facade Pattern

O objetivo deste *design pattern* é abstrair o programador de blocos de código longos e de maior complexidade, criando abstrações, simples e reutilizáveis, que permitam correr as funcionalidades desses blocos de código.

Singleton Pattern

O *Singleton Pattern* é usado para criar objetos que são instanciados apenas uma vez.

3.5.3 ECMAScript 6 e 7

ECMAScript é a linguagem de programação que acompanha o *standard* da Web para a programação imperativa — ECMA-262 —, proposto pela ECMA International e aprovado pelo W3C. A linguagem foi criada com o objetivo de uniformizar os vários dialetos de JavaScript que existiam quando a programação imperativa foi introduzida na Web, em 1995 [Kei05].

Hoje, as versões de JavaScript implementadas na maioria dos navegadores implementam a especificação proposta pelo *standard* de ECMAScript correspondem à sua versão 5.1. No entanto, o lançamento da versão 6 da linguagem está agendada para Junho de 2015 e já existem novas funcionalidades propostas para as próximas versões [Zak14].

As novas versões do ECMAScript trazem para a Web e, em particular, para o desenvolvimento de *Single-Page Applications*, algumas funcionalidades que devem ser mencionadas.

Arrow Functions

As *Arrow Functions* oferecem uma nova forma para se declararem funções anónimas. Para além de terem uma sintaxe diferente, têm ainda a particularidade de o valor do objeto “this” dentro do corpo da função anónima manter o mesmo valor que tinha fora da função, ao contrário do que acontecia com a declaração de funções anónimas já existente. A Figura 3.3 ilustra um exemplo de uma função deste tipo.

```

1 var imprimirValor = (valor) => {
2   console.log(valor)
3 }

```

Figura 3.3: Exemplo de *Arrow Function*

Let e Const

As palavras-chave “let” e “const” permitem instanciar variáveis e constantes, respectivamente, dentro de blocos de código como condições ou *loops*, que são destruídas depois dos blocos onde são instanciadas serem executados.

Classes

A versão 6 do ECMAScript introduz uma nova sintaxe que permite a implementação de classes, métodos estáticos e polimorfismo, como mostra o exemplo da Figura 3.4.

```

1 class MacbookPro extends Macbook {
2   constructor(ano, eRetina) {
3     super(ano)
4
5     this.ano = ano
6     this.eRetina = eRetina
7   }
8   metodo() {
9     //...
10    super.metodo();
11  }
12  static metodoEstatico() {
13    //...
14  }
15 }

```

Figura 3.4: Exemplo de classe em ECMAScript 6

Promises e Async Functions

As *Promises* apareceram com o objetivo de resolver o problema da *Pyramid of Doom* do JavaScript, oferecendo uma forma de escrever sequências de chamadas a funções que retornam resultados assincronamente mais fácil de ler. Porém, a proposta para a especificação da versão 7 do ECMAScript já inclui uma outra solução para este problema, que simplifica ainda mais o desenvolvimento de código assíncrono na Web — *Async Functions* [Arc14].

A Figura 3.5 mostra uma comparação entre uma mesma sequência de operações assíncronas desenvolvida com *callbacks*, com *Promises* e, por último, com *Async Functions*.

Transpiladores

Apesar de os navegadores não suportarem ainda grande parte das funcionalidades introduzidas nas versões mais recentes do ECMAScript, já é possível usar algumas destas funcionalidades na

```

1 // Código assíncrono com funções callback
2 function contexto() {
3     metodoAssincrono1(function () {
4         metodoAssincrono2(function () {
5             metodoAssincrono3(function () {
6                 console.log('concluído')
7             })
8         })
9     })
10 }

```

(a) Implementação com funções *callback*

```

1 // Código assíncrono com Promises
2 function contexto() {
3     metodoAssincrono1().then(function () {
4         return metodoAssincrono2()
5     }).then(function () {
6         return metodoAssincrono3()
7     }).then(function () {
8         console.log('concluído')
9     })
10 }

```

(b) Implementação com *Promises*

```

1 // Código assíncrono com funções Async
2 async function contexto() {
3     await metodoAssincrono1()
4     await metodoAssincrono2()
5     await metodoAssincrono3()
6     console.log('concluído')
7 }

```

(c) Implementação com *Async Functions*

Figura 3.5: Sequência de operações assíncronas em JavaScript

Web recorrendo a transpiladores⁹, que convertem o código-fonte desenvolvido em ECMAScript 6 e 7 para ESCMAScript 5.1, que é atualmente suportado pela maioria dos navegadores mais recentes. O Babel¹⁰ é um exemplo de uma ferramenta deste tipo.

3.5.4 Testes Unitários

Existem já algumas bibliotecas e *frameworks* que podem ser usadas em conjunto para se conseguir uma implementação completa de testes unitários ao *front-end* de Aplicações *Web*. Três exemplos são o Mocha, o Chai e o Sinon [Wei14].

Mocha

O Mocha¹¹ é uma *framework* que oferece uma API para a criação de testes unitários em JavaScript e um ambiente gráfico onde revela os resultados desses testes. Com o Mocha, é ainda possível avaliar o resultado de métodos assíncronos, como é o caso de uma função que executa um pedido a um servidor remoto e que fica à espera de uma resposta, enquanto a aplicação executa outras tarefas.

⁹Ferramenta que converte código entre duas linguagens semelhantes, que partilhem um nível de abstração semelhante.

¹⁰<https://babeljs.io/>

¹¹<http://mochajs.org/>

Chai

O Chai é a biblioteca de funções que permite fazer as comparações entre os resultados esperados e obtidos dos testes unitários declarados com a *framework* Mocha. A Figura 3.6 mostra um exemplo da implementação de um teste unitário simples com a *framework* Mocha e esta biblioteca.

```
var expect = chai.expect;

describe("Person", function() {
  describe("constructor", function() {
    it("should have a default name", function() {
      var person = new Person();
      expect(person.name).to.equal("Pessoa");
    });
  });
});
```

Figura 3.6: Teste unitário implementado com Mocha e Chai

No exemplo, é criado um ambiente de testes para os objetos criados a partir do objeto “Person” e é implementado teste ao método construtor que verifica que todos os objetos de “Person” instanciados com o construtor genérico têm o nome “Pessoa” [Wei14].

Sinon

O Sinon é uma biblioteca que simula um ambiente de execução real para os testes unitários a serem executados. O seu uso é útil em situações em que os métodos que pretendem ser testados afetam o estado do ambiente em que são executados, por oposição a retornarem diretamente os seus resultados [Sin10].

Web Component Tester

O Web Component Tester é uma ferramenta desenvolvida pela equipa responsável pelo Polymer (Secção 3.2.6) baseada em várias ferramentas de testes unitários de *front-end* — entre as quais as três descritas acima — que permite testar *Web Components* criados com a biblioteca Polymer [Osm14b].

3.6 Conclusões

Parte das tecnologias descritas neste capítulo ainda se encontram em fase experimental, enquanto que outras, embora já possam ser usadas em desenvolvimento num contexto de produção, são, não obstante, igualmente recentes e têm poucas provas práticas dadas no mundo do desenvolvimento para a *Web*.

Já foram debatidas as vantagens do uso de cada uma individualmente. Porém, esta dissertação compromete-se a estudar a possibilidade e os benefícios de se definir uma arquitetura *front-end* que conjugue as vantagens de cada uma e que possa, como resultado, dar um próximo passo no caminho da evolução da qualidade de desenvolvimento e na qualidade em si das Aplicações *Web*.

Tecnologias Emergentes

No próximo capítulo, será descrita a proposta de arquitetura defendida nesta dissertação.

Capítulo 4

Proposta de Arquitetura Front-end

Se cada uma das tecnologias descritas no capítulo anterior se focava na resolução de um problema, em particular, dos que foram inicialmente descritos no Capítulo 2, pretende-se agora encontrar uma forma de as unir, com o objetivo de solucionar os vários problemas descritos nesse capítulo.

No decorrer da realização desta dissertação, foram testadas e estudadas grande parte das tecnologias descritas no Capítulo 3. Desse estudo, resultou a proposta de uma nova arquitetura *front-end* para o desenvolvimento de Aplicações *Web* que reúne, na sua estrutura, os conceitos *Web Components*, *Offline-first*, *Responsive Web Design* e *Single-Page Applications*. Neste capítulo, serão revelados e justificadas as escolhas feitas para os vários componentes integrantes dessa arquitetura.

4.1 Tecnologias Selecionadas

Na sequência da experimentação feita às diversas tecnologias que foram faladas no Capítulo 3, foi feita então a escolha das tecnologias a integrar a proposta de arquitetura desta dissertação. É importante notar que as tecnologias foram selecionadas tendo por base os seguintes critérios:

1. Dar resposta ao máximo de problemas descritos no Capítulo 2;
2. Garantir um elevado nível de interoperabilidade com os restantes componentes da arquitetura;
3. Conceder à arquitetura final uma estrutura modelar, com o objetivo de facilitar a eventual troca futura de qualquer um dos seus componentes;
4. Promover tecnologias que se enquadrem com os *standards* da *Web* ditados pelo W3C.

4.1.1 Polymer para *Web Components*

Das alternativas exploradas — AngularJS e Polymer —, a biblioteca Polymer é a única ferramenta que implementa todas as tecnologias abrangidas pelo conceito de *Web Components* e igualmente a única que é fiel à especificação do W3C de *Web Components*.

Contudo, e como já foi dito, o Polymer é uma biblioteca que simplifica o desenvolvimento de Aplicações *Web* com *Web Components*. Outra alternativa seria, em todo o caso, usar apenas JavaScript e a especificação da API de *Web Components* do W3C para criar novos componentes. No entanto, seguindo esse caminho, estar-se-ia a afastar um dos objetivos-chave desta dissertação, que é o de simplificar o processo de desenvolvimento de Aplicações *Web*.

Deve ainda ser referido que, quando foi dado início à fase de desenvolvimento desta dissertação, a versão mais recente disponível do Polymer era a 0.5, que foi a que acabou por ser usada na arquitetura proposta. No entanto, já depois de definida a arquitetura descrita neste capítulo, foi lançada a versão 1.0, que trouxe mudanças significativas, em relação à versão anterior. Deve, contudo, ser tido em atenção que todas as referências feitas ao Polymer, ao longo deste documento, se referem à versão 0.5.

4.1.2 *Offline-first* com Hoodie

No âmbito do *Offline-first*, foram referidas, na Secção 3.3.4, três ferramentas — Hoodie, Meteor e Parse. Contudo, se o Hoodie era uma *framework NoBackend*, *Offline-first* e completamente gratuita, por ser *open source*, o mesmo não se pode dizer das restantes duas ferramentas.

O Meteor é uma *framework Offline-first* e *open source*, embora não seja *NoBackend*, enquanto que o Parse, por oposição, é um serviço pago, *NoBackend*, mas não *Offline-first*. Porém, o Parse já é um serviço com bastantes provas dadas em produção [Par12a]. Por analogia, também o Meteor é já uma *framework production-ready* bastante usada em produção [Met13a].

O Hoodie, pelo contrário, está ainda numa fase embrionária e apresenta várias limitações, frente aos seus concorrentes. Uma das suas desvantagens mais significativas é não recorrer a uma base de dados *front-end* para fazer o armazenamento de dados do lado do cliente. Como consequência de usar o HTML5 *Web Storage*, a sincronização de dados entre as bases de dados de *front-end* e de *backend* não tem, em algumas situações, o comportamento esperado e, ainda pela mesma razão, não existem coleções, permissões ou interrogações à base de dados do lado do cliente.

Das três hipóteses, o Parse, não obstante as suas vantagens, não pôde ser o escolhido, sendo o *Offline-first* um requisito para a arquitetura a propor. O Meteor, por outro lado, não é *NoBackend*, pelo que, por exclusão de partes, o escolhido acabou por ser o Hoodie, apesar das suas limitações.

Contudo, o fato de o Meteor não ser *NoBackend* não foi o único motivo para a escolha do Hoodie. O Hoodie ganhou ainda por se focar apenas na resolução dos problemas do âmbito do armazenamento de dados e por ser facilmente extensível, graças ao seu sistema de *plugins NodeJS*. Por essa razão, no futuro e se for necessário, será mais simples trocar o Hoodie por outro módulo

que se encarregue da sua funcionalidade do que seria fazer o mesmo com o Meteor, por ser uma *framework full-stack*.

4.1.3 Outros Componentes

Para além do Polymer e do Hoodie, foram ainda integrados na arquitetura as seguintes tecnologias:

- *Service Worker* — com o objetivo de conceder, às aplicações que implementem a arquitetura proposta, uma experiência de navegação *offline* semelhante à das aplicações nativas e de tornar possível acrescentar funcionalidades como *Push Notifications* e *Background Services*;
- *Web Animations* e *Responsive Images* — que permitem melhorar a experiência de utilização das aplicações desenvolvidas;
- Web Component Tester — para a implementação de testes unitários aos *Custom Elements* criados com Polymer;
- Mocha, Chai e Sinon — para a implementação de testes unitários aos *plugins* de Hoodie.

Foi ainda ponderada a integração de uma *framework* de CSS — Compass [3.4.4](#). No entanto, o uso de *Web Components* pressupõe a criação de componentes simplistas, direcionados para a resolução de um problema específico, como diz Taylor Savage — gestor de produto do Polymer.

“For just about any problem you might need to solve on the web, there’s an element for that.” [Sav15]

Para além disso, o Polymer já oferece tecnologias como o *Shadow DOM* ou ferramentas que facilitam a criação de *layouts* responsivos. É portanto suposto que a complexidade da UI das aplicações desenvolvidas com a arquitetura proposta esteja distribuída e encapsulada nos vários componentes da aplicação e que, por essa razão, não implique o desenvolvimento de um código CSS complexo, difícil de desenvolver e de manter.

Em suma, o uso de uma *framework* ou de uma linguagem alternativa ao CSS acrescentaria complexidade à arquitetura sem maiores benefícios [\[Med11\]](#). Para além disso, constituiria ainda um desvio em relação aos *standards* do *World Wide Web Consortium*.

4.2 Visão Geral da Arquitetura

Por estar a ser proposta uma arquitetura *front-end*, vai ser descrita apenas uma visão geral dos detalhes da sua estrutura do lado do *backend*, neste caso referentes aos mecanismos de sincronização de dados do Hoodie e da possível extensão da sua funcionalidade com *plugins* NodeJS.

A Figura 4.1 representa a forma como os principais componentes se distribuem pela arquitetura e se interligam. Como é visível na figura, o *front-end* é composto por uma *Single-Page*

Proposta de Arquitetura Front-end

Application que irá correr numa janela do navegador. Porém, ainda do lado do *front-end*, existe outro componente — *Service Worker* — que, embora também corra no navegador, será corrido noutra processo, em segundo plano, e continuará a correr mesmo depois da janela da aplicação a que estiver associado ser fechada.

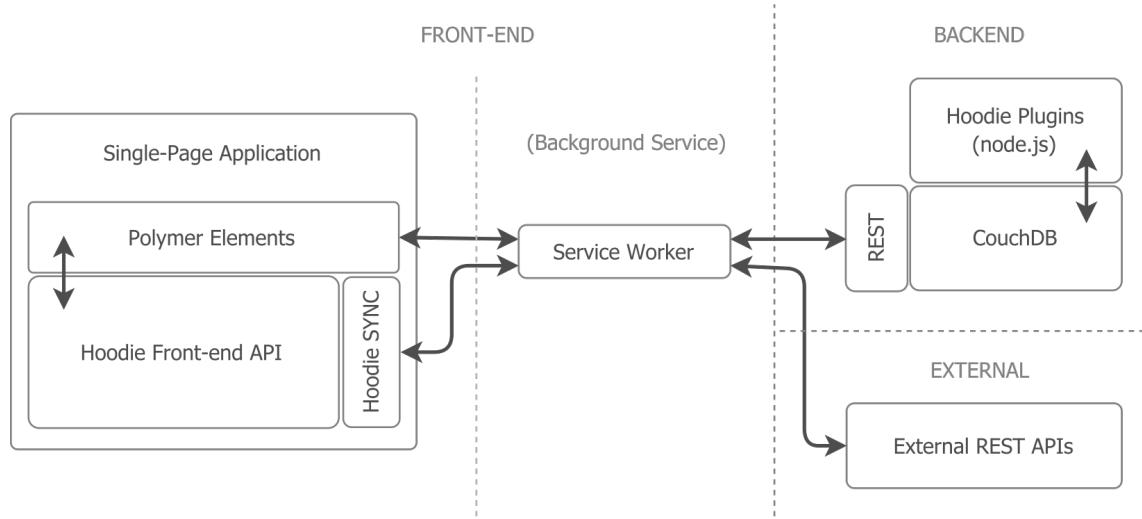


Figura 4.1: Diagrama da visão geral da arquitetura proposta

Service Worker

O *Service Worker* ficará responsável por interceptar e tratar os pedidos feitos pela aplicação, ou reencaminhando-os para o *backend* ou para uma API REST remota, ou respondendo com uma resposta anterior, armazenada em *cache*.

Single-Page Application

A *Single-Page Application* será, por sua vez, composta por *Web Components Polymer*, por uma API de JavaScript do Hoodie e por um componente de sincronização interno ao Hoodie — Hoodie SYNC — que ficará responsável por assegurar a persistência dos dados em *front-end*.

O componente Hoodie SYNC comunicará com uma API REST do Hoodie em *backend* (através do *Service Worker*), com o HTML5 *Web Storage* em *front-end* e com a API de JavaScript do Hoodie. O componente Hoodie SYNC encarregar-se-á ainda de manter os dados armazenados, local e remotamente, em sincronização e espoletará eventos sempre que existirem novas alterações de dados. A API de JavaScript do Hoodie permitirá então ao programador subscrever estes eventos e trabalhar os dados armazenados com as operações básicas de leitura e escrita [Hoo14a].

Plugins Hoodie

Do lado do servidor, o componente Hoodie Plugins, que representa na figura o mecanismo de *plugins* do Hoodie, receberá também notificações sobre alterações feitas aos dados na base de

dados CouchDB. Isto faz com que seja possível o programador espoletar uma ação de um *plugin* no *backend* criando um documento, com a API de JavaScript do Hoodie, em *front-end*, desde que o documento criado tenha um ID no formato que é reconhecido pelo Hoodie como o correspondente ao de uma *task* (exemplo: \$enviar-mensagem).

Custom Elements na SPA

A *Single-Page Application* será, toda ela, composta por *Custom Elements*. Deverá existir, portanto, um elemento que se encarregará de inicializar o *Service Worker*, bem como outro que desempenhe as tarefas necessárias para inicializar o Hoodie em *front-end*, de forma a que os dados armazenados através do Hoodie possam ser usados na aplicação através desse componente. A Figura 4.2 representa uma estrutura genérica base para uma *Single-Page Application* desenvolvida com esta arquitetura.

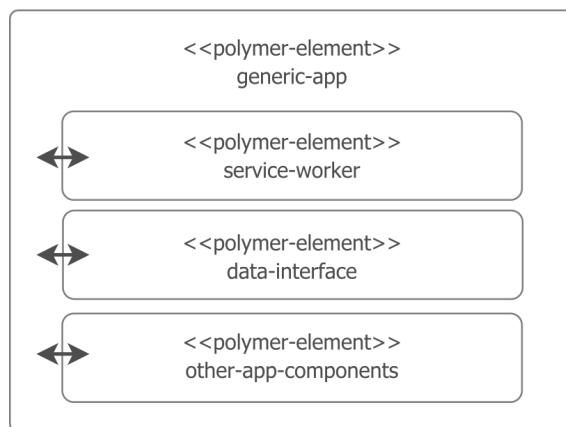


Figura 4.2: Estrutura base de uma SPA com *Custom Elements*

A comunicação entre os vários componentes que compõem a SPA será feita através do componente “generic-app” — o mediador no *Mediator Pattern*. Para além disso, cada componente deverá encapsular as suas próprias características — entenda-se o seu estado, os dados que usa e as suas operações de lógica de negócio. O objetivo é promover a criação de componentes que não dependam do contexto em que são usados, de forma a facilitar a sua reutilização noutros contextos. Assim, neste caso, o elemento “data-interface” deverá incluir as dependências de JavaScript do Hoodie e fornecer, na API de DOM do componente, uma interface que permita usufruir das operações facultadas pela API de JavaScript do Hoodie.

4.3 Detalhes de Implementação

Já foi dito que os *Custom Elements* serão os alicerces da estrutura da *Single-Page Application*, na arquitetura proposta. Desta forma, é importante perceber como funcionam internamente esses componentes e de que forma uma aplicação constituída somente por *Custom Elements* é capaz de

satisfazer todos os requisitos exigidos a uma *Single-Page Application* — desde funcionalidades que passem pela leitura e manipulação de dados à criação de interfaces dinâmicas, com animações ricas e *layouts responsivos*.

4.3.1 Declaração de um *Custom Element*

A Figura 4.3 ilustra um exemplo da declaração de um *Custom Element* genérico, com Polymer, onde se consegue perceber de que forma pode ser organizada a implementação das suas várias partes lógicas.

```

1 <polymer-element name="elemento-declarado" attributes="dadoUm">
2   <template>
3     <style>
4       /* Regras de estilo CSS */
5     </style>
6     <!-- Tag "content" permite selecionar HTML da propriedade "innerHTML"
7         de uma instância deste componente (opcional). O atributo "select"
8         recebe um seletor de CSS que especifica quais os elementos HTML
9         usados -->
10    <content select="componente-selecionado"></content>
11
12    <!-- Restantes elementos que compõem o componente (opcional) -->
13    <example-component dadoUsado="{{dadoUm}}"
14      on-evento-disparado="{{acaoUm}}></example-component>
15    <another-component id="outroComponente"></another-component>
16  </template>
17  <script>
18    // Construtor de Polymer Custom Element
19    Polymer({
20      // Propriedades e métodos
21      dadoUm: 0, // valor por defeito = 0
22      acaoUm: function(event) {
23        // Implementa uma ação que é realizada quando o evento
24        // "evento-disparado" da instância do componente
25        // "example-component", declarada neste componente, é
26        // despoletado. Neste caso específico, concretiza ainda
27        // a comunicação entre os componentes "example-component"
28        // e "another-component".
29
30        // O método "acao" pertence à API de DOM do componente
31        // com o id "outroComponente".
32        this.$.outroComponente.acao()
33      }
34    })
35  </script>
36 </polymer-element>
```

Figura 4.3: Declaração de *Custom Element* com Polymer

É importante notar que a sintaxe `{ {propriedade} }` é usada na biblioteca Polymer para definir e usar propriedades cujos valores serão atualizados através do mecanismo de *two-way data binding* interno ao Polymer. Para além disso, atributos que sigam o padrão `on-nome-do-evento` são usados para tratar eventos de DOM recebidos nos elementos a que são aplicados.

Dados

No atributo “`attributes`” é passada uma lista das propriedades que cada instância do elemento poderá receber. É através dessas propriedades que deve ser passado o modelo de dados usado pelo componente para desempenhar a sua funcionalidade. Essas propriedades podem, posteriormente, ser usadas diretamente no HTML da declaração do componente recorrendo à sintaxe `{propriedade}`, como acontece na Figura 4.3, na linha 13. Porém, com Polymer, é ainda possível definir facilmente valores pré-definidos para essas propriedades. Isto é feito ao nível do registo do elemento, em JavaScript, como acontece no exemplo da figura, na linha 17.

É ainda possível que o componente utilize dados fornecidos pelos elementos que o compõem, como se vai ver, mais à frente, nos componentes associados ao Hoodie e à manipulação de dados (Secção 4.3.4).

Layout

As regras de estilo CSS e o HTML que, juntos, constroem o resultado visual de um componente, devem ser implementados dentro da tag “`template`” — linha 2 do exemplo. É ainda possível selecionar HTML que seja aplicado dentro de uma instância do elemento declarado, como “`innerHTML`”. Para isso, pode ser usada a tag “`content`” — linha 8 —, que pode ser usada com o atributo “`selector`” (opcional) e que permite recorrer a um seletor de DOM para escolher os componentes a reproduzir.

Lógica de Negócio

Por fim, as operações de lógica de negócio de cada componente podem ser implementadas como métodos de JavaScript, no momento do registo do elemento Polymer. Contudo, podem recorrer a funcionalidades mais simples de outros componentes — como, por exemplo, os elementos “`other-component`” e “`another-component`” da figura — para comporem funcionalidades mais complexas, promovendo a reutilização de código.

4.3.2 Comunicação entre Elementos

A comunicação entre *Custom Elements* é feita recorrendo ao *Mediator Pattern*. No exemplo da Figura 4.3, o elemento declarado será o mediador da comunicação entre os componentes “`other-component`” e “`another-component`”.

A propagação de eventos na árvore do DOM é feita sempre no sentido das folhas da árvore para a sua raiz. Desta forma, no caso do exemplo, o elemento declarado receberá o evento “`evento-disparado`”, lançado pelo componente “`other-component`” e trata-lo-á no método “`acaoUm`” — linha 19. Posteriormente, o método “`acaoUm`”, irá chamar o método “`acao`” do elemento “`another-component`” de forma a concretizar a ação desejada. Assim, consegue-se fazer com que nenhum dos intervenientes na comunicação precise de conhecer detalhes da implementação dos restantes,

além do mediador, que, por encapsular os restantes, continua a poder ser aplicado a qualquer contexto.

4.3.3 *Custom Element* para o Hoodie

A declaração do *Custom Element* do Hoodie deverá incluir e inicializar a biblioteca de *front-end* do Hoodie e deverá implementar uma interface, na API de DOM do componente, que permita usar os métodos da API do Hoodie a partir de outros componentes da aplicação. Na Figura 4.4, é ainda representada a comunicação da API do Hoodie com a rede. Deve-se notar que essa comunicação dar-se-á recorrendo aos mecanismos descritos na visão geral da arquitetura (Secção 4.2).

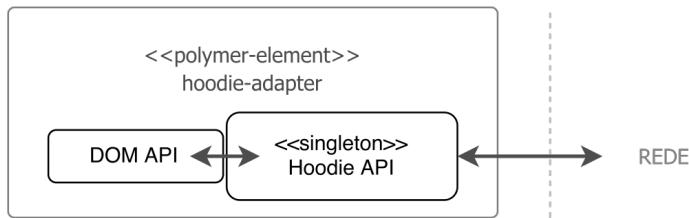


Figura 4.4: *Custom Element* para o Hoodie

É importante referir que é provável que o programador queira reutilizar o componente em várias partes da aplicação, onde queira ter acesso a dados armazenados com o Hoodie. Desta forma, deve apenas ser criada uma instância de Hoodie, em JavaScript, que deve ser partilhada por todas as instâncias do *Custom Element* de Hoodie ao longo da aplicação — *Singleton Pattern*. Caso contrário, seria criada uma nova ligação ao *backend* de Hoodie de cada vez que fosse criada uma nova instância do *Custom Element*.

4.3.4 Sessões, Coleções e *Queries*

Operações como criar e apagar sessões de utilizador, ou ler e alterar dados do utilizador podem ser feitas diretamente com a API de DOM do componente do Hoodie. Um exemplo de autenticação simples pode ser visto na Figura 4.5.

Operações relacionadas com a manipulação de dados, no entanto, são um bocado mais complexas. No caso prático de retornar todos os elementos de uma coleção genérica de Hoodie, por exemplo, é necessário criar dois *Custom Elements* adicionais — um que represente a coleção com que se pretende trabalhar e outro que use o primeiro e que permita ao programador ler dados dessa coleção.

Manipulação de Dados

Na Figura 4.6, é novamente instanciado um *Custom Element* de Hoodie — “hoodie-adapter” —, que é usado por outro componente — “data-collection”. O principal objetivo da criação deste último é fornecer ao programador uma API de DOM que lhe permita indicar apenas uma vez o

```

1 <polymer-element name="elemento-declarado">
2   <template>
3     <!-- Ligação entre o elemento e o Hoodie -->
4     <hoodie-adapter id="hoodie"></hoodie-adapter>
5   </template>
6   <script>
7     Polymer({
8       signIn: function(username, password) {
9         // É chamado o método "signIn" da API de DOM
10        // do componente do Hoodie
11        this.$.hoodie.signIn(username, password)
12      }
13    })
14  </script>
15 </polymer-element>

```

Figura 4.5: Login com *Custom Element* Hoodie

nome da coleção que pretende usar — quando cria uma nova instância do componente — e que, a partir desse momento, todas as operações feitas com essa instância do componente sejam aplicadas a essa coleção em específico.

Deve-se notar que a sintaxe “`$.idDoElemento`” é usada em Polymer para referenciar, através de um elemento, outros elementos de DOM que o componham.

O componente “data-collection” aparece, no entanto, também com o objetivo de contribuir para a modularidade da arquitetura proposta e de permitir que o *backend* de Hoodie, que neste momento é uma parte integrante da arquitetura, possa ser, no futuro, facilmente substituído. Para tornar isto possível, o elemento “data-collection” deve garantir que o objeto que lhe é passado através do atributo “adapter” tem sempre o mesmo comportamento, esteja a ser usado o Hoodie ou outra tecnologia de *backend* qualquer. Existem linguagens de programação que suportam uma funcionalidade que existe para tratar estes casos — *interface*. O JavaScript não é uma delas, mas existem técnicas que permitem alcançar resultados semelhantes. Uma relativamente simples e que serve as necessidades do caso em questão chama-se *duck typing*.

“If it walks like a duck and quacks like a duck, it must be a duck.” [OZPSG10]

Duck typing é uma técnica normalmente usada em linguagens dinâmicas para avaliar se um dado objeto é ou não uma instância de uma dada classe, baseada nos métodos que o objeto implementa [DH07]. No caso específico do componente “data-collection”, pode ser usado *duck typing* para obrigar a que o objeto passado no atributo “adapter” implemente os métodos de manipulação de dados usados internamente pelo “data-collection”.

Existe ainda um terceiro componente — “data-collection-query” —, que deverá receber no atributo “collection” a instância do elemento “hoodie-adapter” e que fará *binding* ao valor da propriedade “resultados” — que será criada no “elemento-declarado” - dos resultados da *query* feita internamente pelo componente “data-collection-query” ao Hoodie.

Contudo, é importante referir que, desta forma, o elemento “data-collection-query” assumirá que irá ser usado aliado a um *Custom Element* “data-collection”. Assim, por estes componentes

```

1 <!-- Atributo "noscript" dispensa o uso do construtor de JavaScript -->
2 <polymer-element name="elemento-declarado" noscript>
3 <template>
4   <hoodie-adapter id="hoodie"></hoodie-adapter>
5   <!-- Usa o componente do Hoodie para garantir o acesso
6       declarativo a uma coleção de dados do Hoodie -->
7   <data-collection id="colecaoUm" adapter="{{$.hoodie}}>
8     <!-- nome="colecaoUm"-->
9   <!-- Faz uma query à coleção declarada no elemento "data-collection"
10    e faz binding dos resultados à propriedade "resultados" do
11    componente "elemento-declarado" -->
12   <data-collection-query collection="{{$.colecaoUm}}"
13     results="{{resultados}}></data-collection-query>
14
15   <!-- Imprime iterativamente os resultados obtidos -->
16   <template repeat="{{elemento in resultados}}>
17     {{elemento}}
18   </template>
19 </template>
20 </polymer-element>

```

Figura 4.6: *Query com Custom Element Hoodie*

não poderem ser reutilizados individualmente, não podem ser usados em qualquer contexto. No entanto, têm a vantagem de dispensar, para os casos mais simples, a necessidade de criar código imperativo para realizar operações de manipulação e tratamento de dados.

Queries Dinâmicas

Internamente, o elemento “data-collection-query” deverá ainda subscrever as alterações feitas à coleção a que cada *query* é aplicada. Sempre que hajam novas alterações aos dados dessa coleção, a propriedade “results” do elemento “data-collection-query” (representado na Figura 4.6) deve ser atualizada de acordo com os novos dados armazenados na coleção.

4.3.5 Custom Elements para Plugins Hoodie

Os *plugins* de Hoodie fornecem, também eles, uma API de *front-end* que estende a funcionalidade do objeto de JavaScript do Hoodie. A forma mais direta de se conseguir acesso a esta API seria estender a API de DOM do *Custom Element* do Hoodie com os métodos das APIs dos *plugins* que se pretendessem usar. No entanto, cada aplicação terá requisitos diferentes, que implicarão o uso de *plugins* diferentes e, por consequência, a adaptação do *Custom Element* do Hoodie a cada caso. Para além de esta não ser uma solução cómoda para os programadores, está ainda em desacordo com um dos principais propósitos da arquitetura proposta, dificultando a fácil reutilização do *Custom Element* do Hoodie.

Dito isto e lembrando uma frase¹, da autoria de Taylor Savage, que já foi citado neste capítulo (Secção 4.1.3), foi decidido que deveria ser criado um novo *Custom Element* para cada *plugin* Hoodie. A Figura 4.7 representa a utilização de um componente de um *plugin* genérico.

¹ “For just about any problem you might need to solve on the web, there’s an element for that.” [Sav15]

```

1 <polymer-element name="elemento-declarado">
2   <template>
3     <hoodie-adapter id="hoodie">
4       <hoodie-plugin id="hoodiePlugin"></hoodie-plugin>
5     </hoodie-adapter>
6   </template>
7   <script>
8     Polymer({
9       usarPlugin: function() {
10         this.$.hoodiePlugin.acaoDoPlugin()
11       }
12     })
13   </script>
14 </polymer-element>

```

Figura 4.7: Exemplo de uso de um *plugin* de Hoodie

A comunicação entre o componente criado para o *plugin* e os restantes componentes integrantes do “elemento-declarado” continuará a funcionar como foi descrito anteriormente, implementando o *Mediator Design Pattern*.

4.3.6 Interfaces Dinâmicas

Favorecer a criação de aplicações com interfaces do utilizador dinâmicas, de acordo com o que é esperado de uma *Single-Page Application*, era outra característica esperada desta arquitetura. As aplicações desenvolvidas com a arquitetura proposta já são, no entanto e por omissão, *Single-Page Applications*.

Os *layouts* de cada *Custom Element* criado com Polymer são criados com *HTML Templates* e são preenchidos com os dados armazenados nas propriedades e calculados com os métodos do componente. Estes dados podem então ser usados para decidir quais os componentes que devem ser mostrados ao utilizador para um determinado estado da aplicação.

Templates Condicionais

O Polymer estende a funcionalidade dos *HTML Templates* e acrescenta algumas funcionalidades úteis que simplificam a vida ao programador. Uma dessas funcionalidades permite controlar o estado de um *template* declarativamente, adicionando à tag “template” o atributo “if”, que poderá receber o valor de uma propriedade do elemento em que o *template* é instanciado, por *data binding*.

Na Figura 4.8 é mostrado um exemplo simples em que apenas um dos dois *templates* declarados será instanciado, consoante o valor da propriedade “utilizadorAutenticado”.

Dados Assíncronos

É ainda importante notar que, em alguns casos — quando as fontes dos valores das propriedades dos *Custom Elements* são exteriores ao componente — esses dados são obtidos assincrona-

```

1 <template if="{{utilizadorAutenticado}}>
2   <!-- Layout 1 -->
3 </template>
4 <template if="{{!utilizadorAutenticado}}>
5   <!-- Layout 2 -->
6 </template>

```

Figura 4.8: Exemplo de uso de *template* condicional

mente e só depois de as instâncias do elemento serem carregadas para o DOM e desenhadas na interface do utilizador.

Em situações onde os dados obtidos, atualizados através do mecanismo de *two-way data binding* do Polymer, sejam aplicados diretamente na UI, o Polymer encarrega-se de fazer essa atualização internamente. Porém, em casos em que os dados precisem de ser tratados antes de poderem ser revelados ao utilizador, isso não acontece. O Polymer, não obstante, oferece formas de o programador subscrever essas alterações (Ver Figura ??).

```

1 Polymer('elemento-declarado', {
2   propriedade: '',
3   observe: {
4     propriedade: 'validate'
5   },
6   propriedadeChanged: function (valorAntigo, valorAtualizado) {
7     // TODO
8   },
9   validate: function (valorAntigo, valorAtualizado) {
10    // TODO
11  }
12 });

```

Figura 4.9: Métodos do Polymer para subscrever alterações em propriedades

Na declaração do componente, é possível definir a propriedade “observe”, que deverá ser um *object literal* de JavaScript. As chaves e os valores do conteúdo do *object literal* “observe” deverão ser, por sua vez, os nomes das propriedades cujas alterações se pretendem subscrever e o método correspondente, para cada uma, que se pretende chamar sempre que for feita uma alteração a essa propriedade, respetivamente. Para além disso, qualquer método do componente que tenha o nome terminado em “Changed”, precedido pelo nome de uma propriedade do componente, será igualmente chamado sempre que a respetiva propriedade for atualizada. Os dois casos estão exemplificados na Figura 4.9.

Assim, o programador poderá, recorrendo a um dos métodos descritos em cima, implementar as operações necessárias para preparar os dados no elemento até que estes estejam preparados para servirem a funcionalidade do componente.

4.3.7 *Layouts Responsivos*

A criação de *layouts* responsivos para as aplicações que implementem a arquitetura proposta deverá ser resolvida com *Custom Elements*. Por sua vez, o problema do uso de imagens que se ajustem à dimensão e à densidade de cada ecrã deverá ser resolvido recorrendo ao uso de

Responsive Images (Secção 3.4.6) — tag “picture” e atributo “srcset” da tag “img” dos *standards* do W3C.

A coleção de *Custom Elements* publicados pela equipa do Polymer disponibiliza um elemento “core-media-query” que recebe, num atributo, uma *media-query* e que devolve noutro uma propriedade atualizável, com um valor booleano, com o resultado dessa *media-query*. Este elemento, usado com *templates* condicionais, permite indicar facilmente que *layout* deverá ser usado na aplicação, dependendo do seu estado. A Figura 4.10 representa um exemplo de um elemento que pode ser renderizado com um de três *layouts* possíveis.

```

1 <polymer-element name="elemento-responsivo" noscript>
2   <template>
3     <core-media-query query="max-width: 640px"
4       queryMatches="{{telemovel}}"></core-media-query>
5     <core-media-query query="max-width: 1135px"
6       queryMatches="{{tablet}}"></core-media-query>
7     <template if="{{telemovel}}">
8       <!-- Layout de telemóvel -->
9     </template>
10    <template if="{{tablet && !telemovel}}">
11      <!-- Layout de tablet -->
12    </template>
13    <template if="{{!tablet && !telemovel}}">
14      <!-- Layout de desktop -->
15    </template>
16  </template>
17 </polymer-element>

```

Figura 4.10: Exemplo de elemento responsivo com “core-media-query”

4.3.8 Temas Globais

O *Shadow DOM* permite produzir código CSS mais fácil de manter, trazendo para o DOM o conceito de encapsulamento de estilos para a *Web*. No entanto, cria uma barreira ao desenvolvimento de temas para as aplicações. Se antes, as regras de estilo eram globais e estavam separadas da implementação da parte lógica das Aplicações *Web*, com o *Shadow DOM*, para se alterar o aspeto global de uma aplicação é necessário alterar o estilo de cada componente.

Além disso, mais uma vez, os *Custom Elements* desenvolvidos devem ser facilmente reutilizáveis em qualquer contexto, embora possa ser útil o mesmo elemento ter estilos diferentes em diferentes contextos. Dá-se o exemplo do componente de um botão que se pretenda instanciar para implementar dois botões numa aplicação: um com a cor de fundo vermelho e outro verde. Uma solução simples seria definir a cor de fundo de cada um no componente em que os dois são instanciados, como no exemplo da Figura 4.11.

Esta é uma boa prática quando se pretende customizar o aspeto de um componente reutilizado. Contudo, continua a não oferecer vantagens no que diz respeito à criação de temas globais para uma aplicação *Web*.

Proposta de Arquitetura Front-end

```
1 <polymer-element name="elemento-declarado" noscript>
2   <template>
3     <style>
4       .botao-vermelho {
5         background-color: red;
6       }
7       .botao-verde {
8         background-color: green;
9     }
10    </style>
11    <botao-generico class="botao-vermelho"></botao-generico>
12    <botao-generico class="botao-verde"></botao-generico>
13  </template>
14 </polymer-element>
```

Figura 4.11: Exemplo de reutilização de um componente com estilos diferentes

Com Polymer, é possível fazer *binding* de propriedades de um componente a código de CSS. Para além disso, é ainda possível fazer *binding* de código CSS a valores de propriedades de outros componentes, como acontece no exemplo da Figura 4.12.

```
5   <!-- Propriedade "tema" está declarada dentro do componente
6   "tema-claro" e contém um object literal de JavaScript
7   com os valores das cores do tema usado -->
8   <tema-claro valores="{{tema}}></tema-claro>
9   <style>
10    .botao-registo {
11      background-color: {{tema.cores.botaoRegistro}};
12    }
13    .botao-autenticacao {
14      background-color: {{tema.cores.botaoAutenticacao}};
15    }
16  </style>
17  <botao-generico class="botao-registo"></botao-generico>
18  <botao-generico class="botao-autenticacao"></botao-generico>
```

Figura 4.12: *Binding* a código CSS

Por outro lado, é simples criar um elemento que guarde, num objeto de JavaScript *singleton*, os principais valores para a implementação do tema global de uma aplicação Web, sejam eles cores, dimensões, endereços para os recursos usados pelo tema, entre outros. No exemplo da figura, o atributo “valores”, do elemento “tema-claro”, será uma propriedade *singleton* e conterá um *object literal* com a propriedade “cores”, que, por sua vez, conterá nas propriedades “botaoRegistro” e “botaoAutenticacao” os valores para as cores que são usadas nas duas regras de CSS declaradas.

Em suma, o desenvolvimento de temas para aplicações que implementem a arquitetura proposta deverá fazer uso da boa prática de atribuir estilos de CSS a componentes reutilizáveis fora da declaração desses componentes e, para casos em que se pretendam usar temas globais, poderá ser criado um elemento auxiliar que contenha, num objeto *singleton*, os valores para os estilos da aplicação, acessíveis em toda a aplicação através de instâncias desse componente. É ainda importante referir que, aplicando as duas técnicas descritas, nos casos em que não for definido, no tema

global, o valor para um determinado componente da aplicação, este continuará a apresentar-se de acordo com as regras de estilo definidas na declaração do componente.

4.3.9 Testes Unitários

Para a implementação de testes unitários em *front-end* — *Single-Page Application* — deve ser usada a biblioteca de testes disponibilizada pela equipa do Polymer — Web Component Tester (Secção 3.5.4). A biblioteca Web Component Tester permite implementar testes unitários ao comportamento da API de DOM de instâncias de componentes Polymer, incluindo a propriedades ou métodos que vejam o seu valor ser alterado assincronamente, e são executados no navegador, num ambiente de execução real.

A implementação de testes unitários para *plugins* de Hoodie poderá ser feita com as ferramentas Mocha, Chai e Sinon, embora não exista ainda disponibilizada qualquer documentação por parte da equipa do Hoodie que ajude os programadores neste sentido [Hoo14c]. Porém, existem já alguns *plugins* Hoodie publicados que seguem esta prática e que podem ser usados como exemplos. É o caso dos *plugins* “hoodie-plugin-stripe”² — um *plugin* de Hoodie genérico, composto apenas pela estrutura base de implementação de um *plugin* de Hoodie — e “hoodie-plugin-users”³ — *plugin* desenvolvido pela equipa responsável pelo Hoodie para gerir sessões de utilizador em Hoodie.

4.4 Ambiente de Desenvolvimento

As tecnologias já descritas neste capítulo são fundamentais para a concretização dos objetivos impostos à arquitetura proposta. Porém, existem outras ferramentas e tecnologias que, embora não sejam fundamentais, podem ser aliadas à arquitetura proposta para se conseguirem melhores resultados.

Utilizar algumas das funcionalidades propostas nas versões mais recentes de ECMAScript (Secção 3.5.3) pode ajudar a aumentar a produtividade dos programadores. O uso da ferramenta Yeoman (Secção 4.4) para gerar a estrutura base de um projeto para a arquitetura proposta ou para criar, a partir de um *template* genérico, novos componentes Polymer é outro exemplo de uma forma de se otimizar o desenvolvimento de aplicações com esta arquitetura. Esta dissertação propõe então o uso de um conjunto de ferramentas e tecnologias que definem um ambiente de desenvolvimento que deverá facilitar o processo de criação de uma aplicação *Web* que implemente a arquitetura proposta:

- **ECMAScript 6 e 7** — estende o potencial do JavaScript com as funcionalidades esperadas para integrarem as suas próximas versões, permitindo aos programadores usarem funcionalidades como *Async Functions*, classes, variáveis encapsuladas em blocos de código “for”

²<https://github.com/hoodiehq/hoodie-plugin-stripe>

³<https://github.com/hoodiehq/hoodie-plugin-users>

ou “if”, entre outras, de forma a conseguirem desenvolver códigos-fonte mais complexos e intuitivos com menos esforço.

- **Babel** — ferramenta utilizada para transpilar o código-fonte desenvolvido em ECMAScript (ES) 6 e 7 para JavaScript atualmente suportado pelos navegadores e para emular nos navegadores algumas das funcionalidades de ES 6 e 7.
- **Webpack⁴** — permite transpilar, com o Babel, o código desenvolvido em ES 6 e 7 sempre que forem feitas alterações aos ficheiros-fonte. Para além disso, o Webpack permite ainda concatenar num mesmo ficheiro de código a árvore de dependências de um ficheiro de JavaScript.
- **Vulcanize** — usado para concatenar a árvore de dependências *Custom Elements* de uma aplicação desenvolvida com Polymer [Osm13]. Para além disso, esta ferramenta permite ainda separar o código JavaScript associado aos componentes a concatenar e concatenar também esse código num ficheiro de JavaScript adicional. Esta funcionalidade, aliada ao Webpack e ao Babel, torna possível concatenar num único ficheiro HTML todo o código-fonte declarativo de uma aplicação desenvolvida com Polymer e num outro ficheiro o código desenvolvido em JavaScript ou em ES 6 e 7, que posteriormente poderá ser transpilado para ECMAScript 5.1 — versão suportada pela maioria dos navegadores atuais. Esta prática, em contexto de produção, permite diminuir a largura de faixa necessária para se transferir a aplicação para um navegador.
- **npm⁵** — gestor de pacotes que permite instalar facilmente algumas das restantes ferramentas usadas no ambiente de desenvolvimento, entre as quais o Babel, o Webpack, o Vulcanize, o Gulp, o Bower e o Yeoman.
- **Gulp⁶** — ferramenta que permite criar e correr tarefas implementadas em JavaScript [Sit14]. A criação das tarefas de compilação do código-fonte ES 6 e 7 para ES 5.1, recorrendo ao Babel e ao Webpack, e da criação de um ficheiro que contenha uma lista com os caminhos relativos para todos os ficheiros de código-fonte de um projeto, são dois exemplos de casos em que esta ferramenta se revela mais útil.
- **Bower⁷** — gestor de pacotes que permite transferir e instalar num projeto dependências de um projeto publicadas na plataforma GitHub através da linha de comandos.
- **Yeoman⁸** — ferramenta que permite criar ficheiros e estruturas de pastas através de *templates* pré-definidos. Esta ferramenta pode ser usada, por exemplo, para criar uma estrutura

⁴<http://webpack.github.io/>

⁵<https://www.npmjs.com/>

⁶<http://gulpjs.com/>

⁷<http://bower.io/>

⁸<http://yeoman.io/>

de pastas e de ficheiros genérica para novos projetos desenvolvidos com a arquitetura proposta de uma forma automatizada, ou para criar a estrutura base para novos componentes Polymer.

- **Git** — sistema de controlo de versões.

4.5 Conclusões

Neste capítulo foram reveladas as tecnologias que integram a proposta de arquitetura *front-end* defendida por esta dissertação, bem como as razões que levaram à escolha de cada uma delas. Foi ainda explicada a forma como os vários componentes da arquitetura se interligam e de que forma é esperado que sejam desenvolvidas *Single-Page Applications* com estruturas baseadas em *Web Components*. Por fim, foi feita uma listagem de algumas ferramentas e tecnologias que são sugeridas para integrarem o ambiente de desenvolvimento dos projetos que decidam implementar a arquitetura proposta que prometem otimizar o processo de desenvolvimento.

A arquitetura definida neste capítulo reflete os frutos do estudo e da experimentação que foram feitos sobre as tecnologias descritas no Capítulo 3. O próximo capítulo descreverá o desenvolvimento de uma prova de conceito que implementa a arquitetura proposta.

Proposta de Arquitetura Front-end

Capítulo 5

Implementação de Prova de Conceito

Depois de definida a proposta de arquitetura *front-end*, para o desenvolvimento de Aplicações *Web*, composta pelas tecnologias descritas no Capítulo 3, era importante aplicá-la a um contexto real e avaliar os seus resultados. Este capítulo descreve a implementação de uma aplicação desenvolvida como prova de conceito da arquitetura proposta e apresenta os resultados obtidos.

5.1 Descrição da Aplicação

A proposta para o projeto a desenvolver como prova de conceito foi feita pela Glazed Solutions, Lda. e consiste na criação de uma aplicação *Web* que permita fazer encomendas de refeições a restaurantes e acompanhar, em tempo-real, o estado dessas encomendas. Deve ainda ser referido que, durante o período em que foi feito este trabalho, estava a ser desenvolvida, na empresa, uma versão nativa da mesma aplicação para a plataforma iOS e que a data prevista para a sua conclusão se antecipava à data limite para entrega deste documento. Desta forma, uma comparação entre o código-fonte desenvolvido para as duas plataformas — iOS e *Web* — era vista como uma forma possível de testar os resultados obtidos com a implementação da arquitetura proposta.

“Nomnow” foi o nome que batizou a aplicação para iOS, que seria composta por duas partes funcionais:

1. Uma, a que qualquer utilizador registado na aplicação teria acesso e que incluiria no seu cardápio de funcionalidades uma listagem dos restaurantes registados no serviço, bem como os respetivos menus disponíveis, e que permitiria a criação de encomendas;
2. Outra, a que só teriam acesso alguns utilizadores, registados na aplicação com credenciais que lhes dariam o acesso a outro tipo de funcionalidades que não serão especificadas neste documento.

Dada a vasta extensão do projeto, tendo em conta o tempo disponível para a realização da dissertação, foi decidido que só iriam ser implementadas, para a aplicação de prova de conceito,

as funcionalidades correspondentes ao ponto 1.

Era também importante que as aplicações desenvolvidas para as duas plataformas partilhassem o mesmo modelo de dados, de forma a que um utilizador registado no serviço Nomnow pudesse usar tanto a aplicação nativa de iOS como a aplicação Web para usufruir do mesmo serviço. O modelo de dados usado pela aplicação de iOS já estava, no entanto, criado, no momento em que foi dado início ao desenvolvimento desta prova de conceito. Os dados usados pela aplicação nativa tinham sido alojados no serviço de *Backend as a Service* Parse (Secção 3.3.4), pelo que seria ainda necessário fazer a integração deste serviço na arquitetura base proposta. Para além disso, era ainda esperado que, tal como acontecia na aplicação de iOS, a autenticação na aplicação fosse feita com credenciais do Facebook.

5.2 Especificação de Requisitos

A primeira fase de desenvolvimento da prova de conceito consistiu na especificação de requisitos da aplicação a ser criada, onde foram definidos os casos de uso e os requisitos não funcionais técnicos que a aplicação deveria satisfazer.

5.2.1 Casos de Uso

A Figura 5.1 ilustra uma visão geral dos casos de uso especificados para a aplicação a desenvolver.

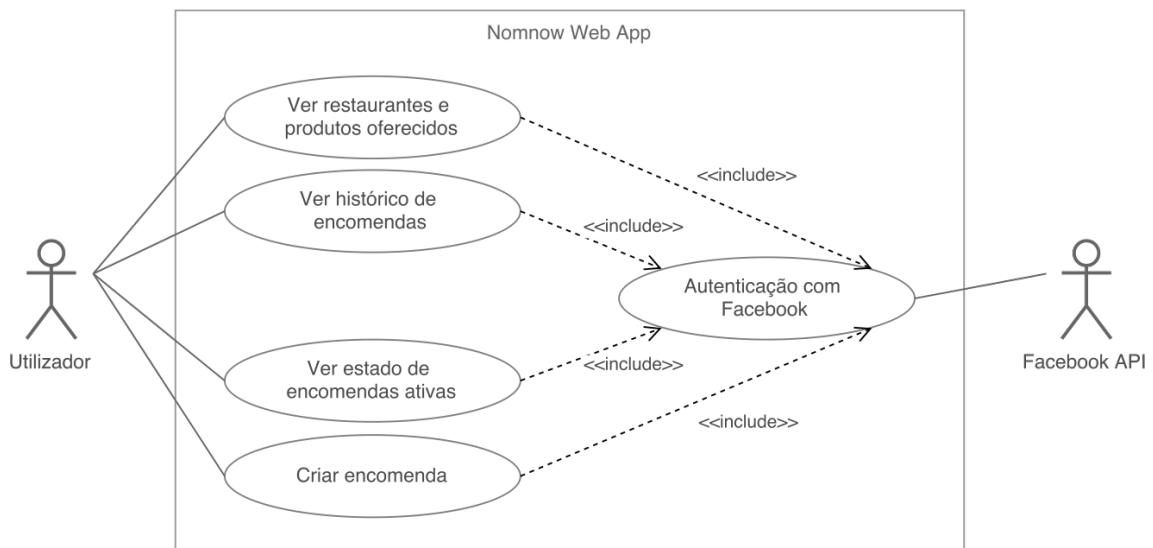


Figura 5.1: Casos de Utilização da aplicação Nomnow

5.2.2 Requisitos Não Funcionais

Como requisitos não funcionais da aplicação, foram definidos os seguintes:

- **Responsividade** — A aplicação deve oferecer uma interface do utilizador minimalista e intuitiva, adaptável aos tamanhos de ecrã de diferentes dispositivos. Deve ainda usar tamanhos de imagens diferentes, consoante as propriedades dos ecrãs em que a aplicação corra.
- **Funcionamento Offline** — A aplicação deve permitir ao utilizador usufruir, sem estar ligado à rede, de todas as suas funcionalidades, exceto as de autenticação com o Facebook e de criação de novas encomendas.
- **Real-time** — A aplicação deve garantir que os dados revelados na sua UI estão constantemente sincronizados com os dados armazenados no modelo de dados da aplicação.
- **Navegação Fluída** — A aplicação deve ser uma *Single-Page Application* e a sua navegação deve ser fluída e feita sem que haja a necessidade de que a aplicação seja recarregada, na integra, pelo navegador.
- **Segurança** — O sistema deve proteger a informação de acessos não autorizados através da utilização de um sistema de autenticação e verificação de privilégios.
- **Sincronização de Dados** — O sistema deve assegurar que os dados armazenados no serviço Parse estão, sempre que forem usados pela aplicação *Web*, sincronizados com os armazenados no *backend* de Hoodie.

5.3 Detalhes de Implementação

Concluída a fase de especificação de requisitos, foi então dado início à implementação da aplicação. Como já foi dito, a aplicação a ser desenvolvida deve usar o mesmo modelo de dados usado pela aplicação desenvolvida para iOS — alojado no serviço Parse — e permitir aos seus utilizadores autenticarem-se com credenciais de Facebook.

5.3.1 Autenticação com Facebook

A solução para a integração do Facebook na arquitetura da aplicação passa pela criação de um *plugin* de Hoodie que estenda, em *front-end*, a API de JavaScript da *framework* com métodos que, implementando o *design pattern Facade*, abstraiam as tarefas necessárias para gerir as sessões do utilizador do Facebook e do Hoodie, a última sendo necessária para que os utilizadores tenham permissões para acederem aos seus dados no *backend* de Hoodie. Para além disso, em *backend*, o *plugin* deve encarregar-se de validar os *tokens* de sessão de Facebook dos utilizadores da aplicação e de oferecer à API de *front-end* uma forma de criar sessões Hoodie sem que seja necessário o utilizador usar outros dados para além dos seus dados de sessão do Facebook.

Porém, a integração do Parse na arquitetura proposta, assegurando a sincronização entre os dados armazenados em Parse e no *backend* de Hoodie, é um problema mais complexo, que é descrito na secção seguinte.

5.3.2 Integração da *Framework* Parse

Parte da solução passou, à semelhança do que aconteceu com o Facebook, por criar um *plugin* de Hoodie que estabelecesse a ligação entre o novo componente a integrar — Parse — e os restantes componentes da arquitetura. Desta forma, foi possível implementar as tarefas que permitiam ter acesso aos dados armazenados no Parse e a replicá-los, a pedido do programador, na base de dados CouchDB. A replicação de novos dados adicionados ao Parse continuaria a ter de ser feita, no entanto, a pedido do utilizador, não acontecendo em tempo-real.

O Parse permitia, no entanto e como já foi revelado na descrição da *framework* (Secção 3.3.4), a criação de *Cloud Code*, implementado em JavaScript e corrido em *backend*. Para além disso, a API de JavaScript do Parse acessível pelas funções declaradas em *Cloud Code* permitia subscrever alterações feitas a coleções de dados Parse e interagir com APIs REST exteriores.

Parse e SocketIO

Foi implementado, em NodeJS, um servidor que disponibiliza uma API REST, usada no *Cloud Code* desenvolvido no Parse, para onde são enviadas notificações sempre que uma dada coleção do modelo de dados da aplicação sofre alterações. Para além disso, para que o *backend* de Hoodie da aplicação a desenvolver conseguisse receber estas notificações em tempo-real, foi ainda usada uma biblioteca de JavaScript — SocketIO¹ — que permitiu implementar, com poucas linhas de código, um mecanismo de comunicação por HTML5 *Web Sockets* entre o *plugin* Hoodie do Parse e o servidor criado.

Em suma, os nomes das coleções que sofrem alterações, armazenadas no serviço Parse, são assim comunicados ao servidor de NodeJS criado, através da sua API REST, e posteriormente publicados para um canal criado pela biblioteca SocketIO, subscrito pelo *plugin* Hoodie desenvolvido para integrar a *framework* Parse. Depois disto, o *plugin* Hoodie encarrega-se de solicitar ao Parse os dados que foram alterados nas coleções referidas na notificação desde a última vez em que a atualização desses dados foi feita. Esta troca de mensagens é ilustrada no diagrama UML de sequência da Figura 5.2.

Permissões

É importante adiantar que, no Hoodie, cada utilizador é proprietário de uma coleção de CouchDB, sincronizada automaticamente com o Web Storage, em *front-end*, e à qual só ele tem permissões de acesso. Para além disso, para dados que, no Parse, não tenham permissões de leitura públicas, são enviados, na notificação recebida pelo *plugin* de Hoodie, os IDs dos utilizadores

¹Socket.IO. Acedido a 2015-06-13. <http://socket.io/>.

Implementação de Prova de Conceito

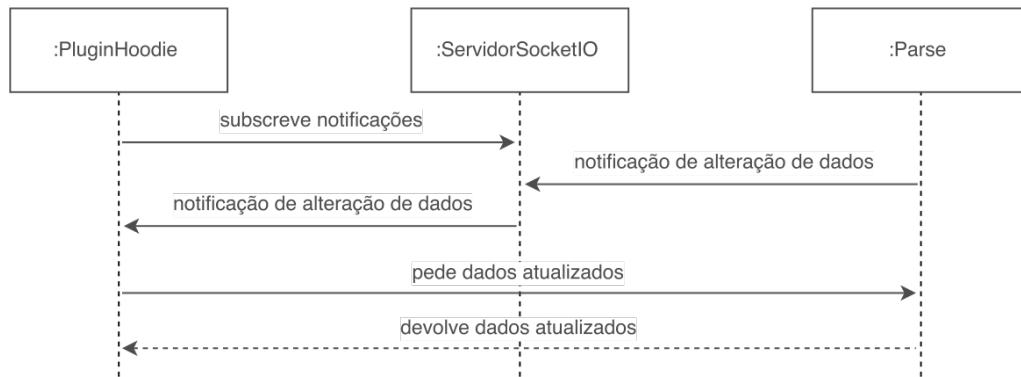


Figura 5.2: Comunicação entre o Hoodie, o servidor de SocketIO e o Parse

Parse que têm acesso aos dados atualizados. Desta forma, os novos dados recebidos pelo *plugin* são diretamente armazenados nas coleções de CouchDB correspondentes aos utilizadores que têm permissões para aceder à informação, garantindo-se assim que as permissões impostas ao modelo de dados do Parse se mantêm, quando os dados são replicados para o Hoodie.

Por outro lado, os dados sem restrições de acesso para leitura são guardados numa outra coleção de CouchDB com o mesmo tipo de permissões. Esta coleção pública está, no entanto, configurada para que os seus dados sejam replicados pelas bases de dados de todos os utilizadores Parse registados na aplicação. Assim, garante-se que todos os dados, armazenados em Parse, a que um utilizador da aplicação tem acesso são guardados na sua coleção de CouchDB, privada e constantemente sincronizada com a sua instalação da aplicação, em *front-end*.

5.3.3 Visão Geral da Arquitetura da Aplicação

A Figura 5.3 ilustra um diagrama da visão geral da arquitetura com os novos componentes.

5.3.4 Comportamento do *Service Worker*

Como já foi referido na Secção 4.2, um *Service Worker* permite intercetar e tratar pedidos feitos por uma aplicação Web ainda ao nível do navegador. Para o *Service Worker* implementado para a prova de conceito desta dissertação, foram definidas as seguintes regras, para o tratamento de pedidos de rede:

- Pedidos cujo endereço contenha a cadeia de caracteres “`api.parse.com`” — correspondendo a pedidos REST feitos diretamente à API REST do Parse, são feitos diretamente à rede.
- Pedidos feitos à API do REST do Hoodie que está ligada ao CouchDB — têm um comportamento semelhante aos do ponto anterior, não sendo tratados.

Implementação de Prova de Conceito

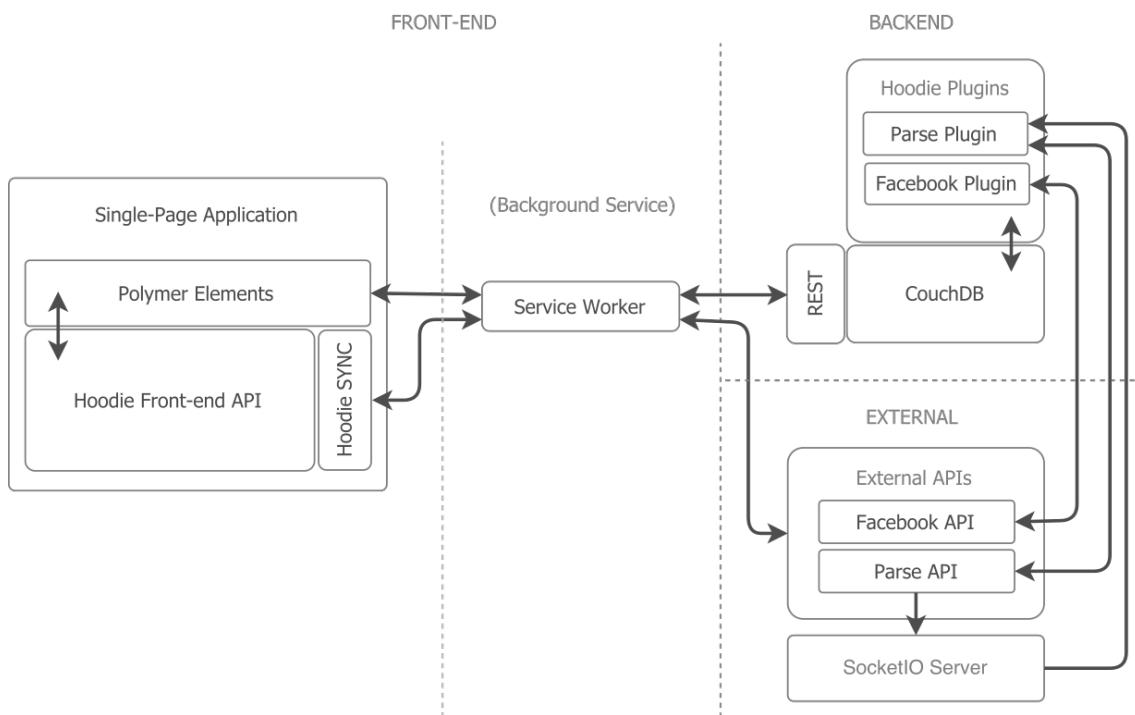


Figura 5.3: Diagrama da visão geral da arquitetura da prova de conceito

- Endereços com as cadeias de caracteres “`gstatic`” ou “`parsetfss`” — representam recursos estáticos, como imagens ou ficheiros, correspondentes ao serviço Google Maps (também usado na aplicação) e Parse, respetivamente. Pedidos deste tipo são respondidos com uma resposta armazenada em *cache*, se existir. Caso contrário, o pedido é reencaminhado para a rede e a resposta obtida é armazenada em *cache*.
- Pedidos de ficheiros do código-fonte da aplicação — com o auxílio da ferramenta Gulp, foi criada uma tarefa que gera uma lista com os ficheiros do código-fonte contidos na estrutura de pastas da aplicação. Esta lista é então usada pelo *Service Worker* para armazenar em *cache* as dependências da aplicação no momento da instalação do *Service Worker* (quando a página é carregada pela primeira vez). Desta forma, todos os pedidos de ficheiros do código-fonte da aplicação que sejam feitos depois da sua instalação são respondidos sempre com recursos armazenados em *cache*.

5.3.5 Estrutura da Single-Page Application

O diagrama da Figura 5.4 representa a estrutura, em árvore, dos principais componentes que compõem a *Single-Page Application*. Note-se que é incluído, no diagrama, um componente correspondente ao *plugin Hoodie* do Parse e que o mesmo não acontece para o *plugin Hoodie* criado para integrar o sistema de autenticação do Facebook na aplicação. Isto acontece, porque o *plugin* do Facebook está definido como uma dependência do *plugin* criado para o Parse e a API do último

Implementação de Prova de Conceito

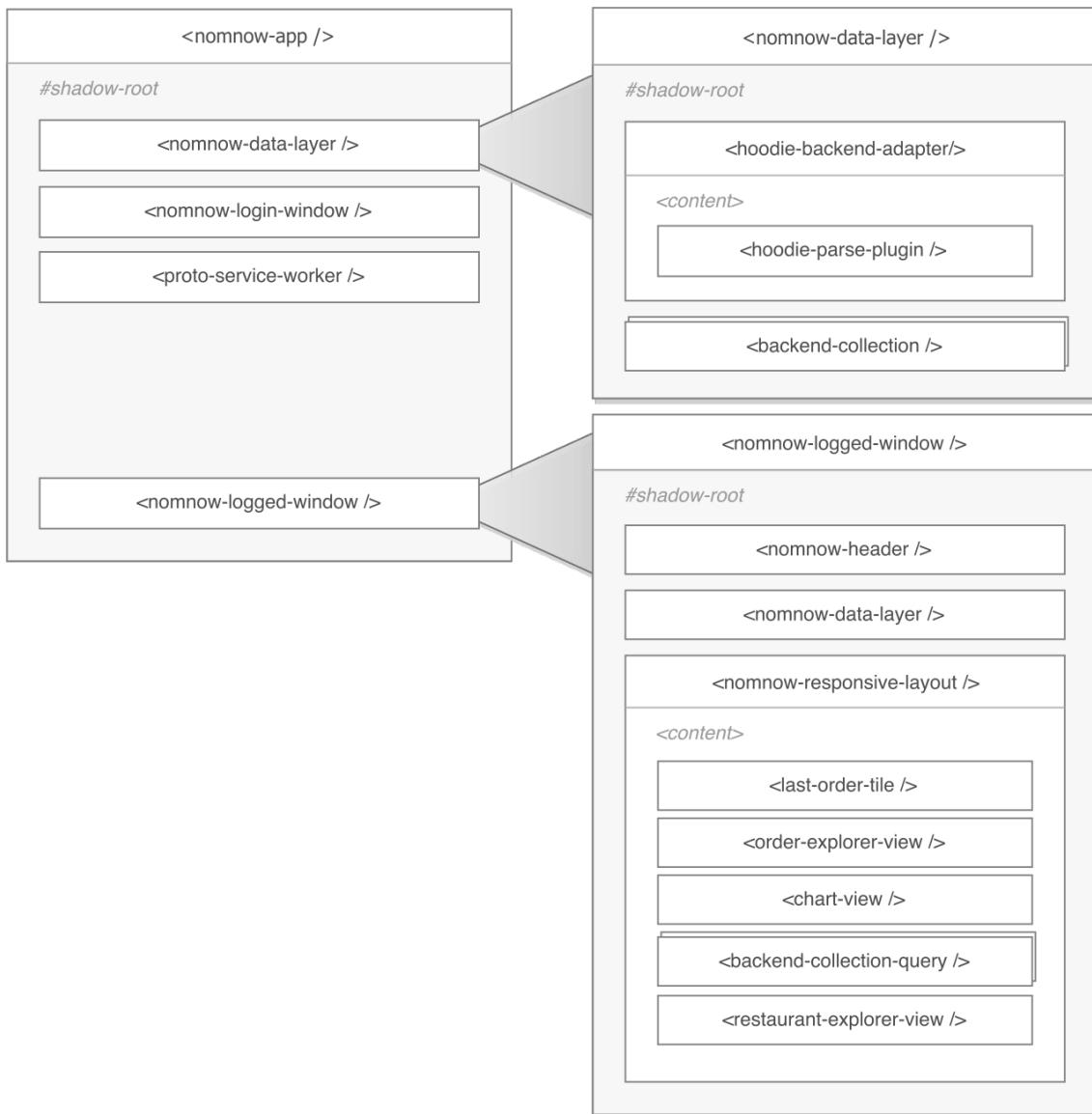


Figura 5.4: Estrutura de *Custom Elements* da *Single-Page Application*

abstrai as tarefas necessárias para que um utilizador se autentique no Parse com uma sessão do Facebook.

Ainda na Figura 5.4, existem dois tipos de relações de composição, entre os elementos representados. Nos casos em que, dentro de um componente, é introduzida a tag “`#shadow-root`”, deve ser entendido que a composição é feita ao nível da declaração do componente e que será, dessa forma, refletida em todas as suas instâncias. Por outro lado, nos casos em que seja usada a tag “`<content>`”, a composição é feita apenas para a instância do elemento considerada.

A aplicação representada na figura pelo componente “`nomnow-app`” é, assim, composta por 4 componentes principais:

- “nomnow-data-layer” — abstrai, em objetos *singleton* de JavaScript, os elementos correspondentes à instância de Hoodie usada na aplicação e aos *plugins* de Hoodie e coleções de dados usadas na aplicação. Note-se que outra instância deste componente é no *Custom element* “nomnow-logged-window”. Este elemento foi criado precisamente para que fosse possível ter-se acesso fácil ao modelo de dados da aplicação em qualquer parte do seu código-fonte, garantindo que, em qualquer instância sua, é sempre usada a mesma e única instância de Hoodie, de cada *plugin* Hoodie e de cada coleção de dados usados.
- “proto-service-worker” — faz o registo do *Service Worker*, com os parâmetros correspondentes a uma lista opcional de ficheiros para serem guardados em *cache* durante a instalação e às expressões regulares que filtrarão os pedidos de rede que serão feitos pela aplicação. É possível, assim, definir que pedidos feitos a endereços de rede que satisfaçam uma determinada expressão regular sejam sempre correspondidos com uma resposta da rede, da mesma forma que é possível indicar outros pedidos para serem satisfeitos com respostas previamente armazenadas em *cache*.
- “nomnow-login-window” — este elemento só é usado caso não exista ainda uma sessão de utilizador aberta na aplicação e é composto pela UI da janela de autenticação. A interação do utilizador com este componente espoleta eventos que serão, posteriormente, enviados para o elemento “nomnow-app”. Este, por sua vez e funcionando como mediador, encarregar-se-á de comunicar com a instância do componente “nomnow-data-layer”, que o compõe, e de processar as tarefas de autenticação correspondentes aos eventos que recebeu.
- “nomnow-logged-window” — elemento que será usado quando existir uma sessão de utilizador validada e que contém grande parte da lógica de negócios da aplicação. Este elemento é composto pela UI do cabeçalho da aplicação — “nomnow-header” — e por um elemento responsável por criar um *layout* responsivo — “nomnow-responsive-layout”. O último recebe ainda, como conteúdo, os elementos da aplicação responsáveis por listar o histórico de encomendas de um utilizador autenticado, as encomendas em progresso e o seu estado, os restaurantes disponíveis e os seus produtos e por efetuar novas encomendas.

5.3.6 Responsividade

Como já foi dito, o *Custom Element* “nomnow-responsive-layout” é o responsável por organizar, internamente, o conteúdo da aplicação num *layout* responsivo, que ofereça uma experiência de visualização e de utilização adequada aos vários dispositivos em que a aplicação possa ser corrida. A Figura 5.5 mostra um excerto do código-fonte deste elemento.

Internamente, o elemento é composto por duas instâncias do *Custom Element* “core-media-query”, desenvolvido pela equipa responsável pelo Polymer e por três outros componentes, que correspondem, cada um deles, aos *layouts* de ecrãs de *smartphones*, de *tablets* e de computadores pessoais, respectivamente. O valor de “queryMatches” é atualizado, através dos mecanismos de *data binding* internos do Polymer, sempre que a largura da janela do navegador varie entre um

Implementação de Prova de Conceito

```
1 <polymer-element name="nomnow-responsive-layout" noscript>
2   <template>
3     <style>
4       (...)
```

```
5     </style>
6     (...)
```

```
7     <core-media-query query="max-width: 640px"
```

```
8       queryMatches="{{phoneScreen}}"></core-media-query>
```

```
9     <core-media-query query="max-width: 1135px"
```

```
10    queryMatches="{{tabletScreen}}"></core-media-query>
```

```
11   <!-- Apenas um dos seguintes templates é usado, dependendo
12      da dimensão da janela da aplicação -->
```

```
13   <template if="{{phoneScreen}}">
```

```
14     <nomnow-phone-content>
```

```
15       <content></content>
```

```
16     </nomnow-phone-content>
```

```
17   </template>
```

```
18   <template if="{{tabletScreen && !phoneScreen}}">
```

```
19     <nomnow-tablet-content>
```

```
20       <content></content>
```

```
21     </nomnow-tablet-content>
```

```
22   </template>
```

```
23   <template if="{{!tabletScreen && !phoneScreen}}">
```

```
24     <nomnow-desktop-content>
```

```
25       <content></content>
```

```
26     </nomnow-desktop-content>
```

```
27   </template>
```

```
28 </template>
```

```
29 </polymer-element>
```

Figura 5.5: Excerto do código-fonte do componente “nomnow-responsive-layout”

intervalo de valores que contenha uma das seguintes medidas: 640 ou 1135 pixels. Assim, a propriedade “phoneScreen”, do componente “nomnow-responsive-layout”, assumirá o valor “true” quando a largura da janela do navegador tiver uma dimensão inferior a 640 pixels, ou o valor “false” caso contrário. O mesmo acontecerá para a propriedade “tabletScreen”, para a dimensão 1135 pixels. Como consequência, será feito *render* do *Custom Element* “nomnow-phone-content” quando a largura da janela da aplicação for inferior a 640 pixels, do elemento “nomnow-tablet-content” quando for superior 640 pixels mas inferior a 1135 e do “nomnow-desktop-content” para dimensões superiores. Para além disso, o conteúdo usado em instâncias do componente “nomnow-responsive-layout” é passado também como conteúdo para o componente que estiver a ser mostrado ao utilizador, seja o “nomnow-phone-content”, o “nomnow-tablet-content” ou o “nomnow-desktop-content”. A organização dos elementos da UI no *layout* passa assim a ser responsabilidade de cada um desses três componentes.

Na aplicação desenvolvida, os principais elementos de UI são implementados pelos componentes “last-order-tile”, “order-explorer-view”, “restaurant-view” e “chart-view” instanciados como conteúdo do elemento “nomnow-responsive-layout” (ver Figura 5.6), encarregue, como já foi dito, de fazer os organizar no *layout* da aplicação.

Implementação de Prova de Conceito

```
1 <nomnow-content-container>
2     (... )
3     <last-order-tile></last-order-tile>
4     <order-explorer-view></order-explorer-view>
5     <restaurant-explorer-view></restaurant-explorer-view>
6     <chart-view></chart-view>
7     (... )
8 </nomnow-content-container>
```

Figura 5.6: Excerto do código de instância do componente “nomnow-responsive-layout”

Layout para Smartphones

A interface do utilizador da aplicação em *smartphones*, para quando existe uma sessão de utilizador aberta, foi dividida em três páginas, navegáveis entre si através de gestos *touch*. O utilizador vê, quando abre a aplicação, uma página onde são listados os restaurantes e, através do gesto *swipe*, pode mudar para uma página à esquerda da inicial, onde é mostrado o seu histórico de encomendas, bem como as encomendas que tem ativas e o seu estado, ou para uma página à direita, onde lhe é mostrado o seu carrinho de compras e onde pode alterar ou concluir um pedido de encomenda.

Para tornar isto possível, foi usado um componente *open source* reutilizável que já implementava, internamente, a navegação por gestos *touch* entre vários ecrãs. A Figura 5.7 mostra como foi usado este componente.

```
1 <swipe-pages class="page-content-wrapper"
2     threshold="0.5" touch-action="none" selected="1">
3     <swipe-page>
4         <div class="page" touch-action="pan-y">
5             <content select="last-order-tile"></content>
6             <content select="order-explorer-view"></content>
7         </div>
8     </swipe-page>
9     <swipe-page>
10        <div class="page" touch-action="pan-y">
11            <content select="restaurant-explorer-view"></content>
12        </div>
13    </swipe-page>
14    <swipe-page>
15        <div class="page" touch-action="pan-y">
16            <content select="chart-view"></content>
17        </div>
18    </swipe-page>
19 </swipe-pages>
```

Figura 5.7: Excerto da declaração do elemento “nomnow-phone-content”

Note-se ainda que, no excerto de código da Figura 5.7, alguns dos componentes que foram passados como conteúdo na instância do elemento “nomnow-responsive-layout” da aplicação — ver Figura 5.4 — estão a ser usados e distribuídos pelas três páginas do *layout* criado.

Layout para Tablets

A criação da interface do utilizador para *tablets* foi feita recorrendo aos *layout attributes* do Polymer que, de uma forma sucinta, são um conjunto de atributos que podem ser usados em qualquer elemento HTML, numa aplicação desenvolvida com Polymer, e que abstraem um conjunto de regras de CSS Flexbox² [Pol14a] e, por consequência, facilitam a criação de *layouts* responsivos.

```
1 <div class="page-content-wrapper" horizontal layout>
2   <div id="left-container" flex>
3     <content select="last-order-tile"></content>
4     <content select="order-explorer-view"></content>
5   </div>
6   <div id="right-container" flex>
7     <content select="chart-view"></content>
8     <content select="restaurant-explorer-view"></content>
9   </div>
10 </div>
```

Figura 5.8: Excerto da declaração do elemento “nomnow-tablet-content”

Conforme é visível na Figura 5.8, foram usados dois elementos “div” com o atributo “flex”, pelos quais foram distribuídos os elementos da UI da aplicação. Esses dois elementos “div” compõem um terceiro, identificado com a classe de CSS “page-content-wrapper” e que ocupa todo o espaço da janela da aplicação, à exceção do ocupado pelo cabeçalho da aplicação. Por este elemento conter os atributos “horizontal” e “layout” e os seus dois descendentes conterem ambos o atributo “flex”, os dois últimos dividirão a largura da janela em igual proporção.

Em suma, a UI da aplicação em *tablets* mostrará, do lado esquerdo da janela, a lista e os detalhes das encomendas do utilizador e, do lado direito, o carrinho de compras, a lista de restaurantes e os seus respetivos produtos.

Layout para Computadores Pessoais

A implementação da interface do utilizador da aplicação para computadores pessoais, foi semelhante à implementada para *tablets*, com a única diferença do *layout* ter sido dividido em três partes, em vez de duas. Foi, portanto, criada uma divisão central, onde foi colocada a lista de restaurantes, antes posicionada à direita.

5.3.7 Elementos Reutilizáveis

No decorrer do desenvolvimento da prova de conceito, foram criados alguns elementos que podem ser facilmente reutilizados noutras projetos:

- “iso-date-formatter” — recebe uma cadeia de caracteres com uma data, no formato ISO, e imprime a mesma data, num outro formato, mais fácil de ler, para o utilizador.

²Sistema de *layouts* responsivos de CSS [MDN13].

- “hour-formatter” — recebe uma cadeia de caracteres com uma data, no formato ISO, e imprime apenas a hora e os minutos, no formato hh : mm.
- “dom-selector” — devolve num *array* todos os seus descendentes de primeira geração.
- “event-dispatcher” — permite tratar declarativamente eventos propagados pelo DOM.
- “open-closed-tag” — recebe duas cadeias de caracteres, correspondentes a datas no formato ISO, e imprime a palavra “open” caso a hora atual esteja entre o intervalo de horas das duas datas fornecidas, ou “closed”, caso contrário.
- “proto-loading” — animação contínua a ser mostrada ao utilizador quando há tarefas pendentes a serem executadas pela aplicação.
- “proto-service-worker” — componente encarregue de registar um *Service Worker* no navegador.
- “star-rating-bar” — barra de classificação por estrelas.
- “theme-style” — pode ser usado em vez do elemento nativo “style”, em navegadores que não suportem *Shadow DOM*.
- “clickable-list-item” — recebe um objeto de JavaScript como atributo e lança um evento, com esse objeto como conteúdo, sempre que for pressionado.
- “hoodie-backend-adapter” — elemento que cria uma instância do Hoodie e que abstrai, na sua API de DOM, os métodos das APIs do Hoodie de tratamento de dados e de gestão de sessões.
- “hoodie-parse-plugin” — elemento que abstrai a API do *plugin* Hoodie criado para o Parse.
- “backend-collection” — permite interagir declarativamente com coleções de dados de um serviço de *backend*.
- “backend-collection-query” — aliado ao elemento descrito no ponto anterior, permite fazer interrogações às coleções instanciadas.

5.4 Resultados

Nesta secção, são reveladas algumas imagens que ilustram o estado final da aplicação desenvolvida como prova de conceito e é feito um levantamento dos resultados obtidos com a implementação da arquitetura proposta.

5.4.1 Capturas de Ecrã

De seguida, são reveladas algumas imagens de ecrãs da aplicação desenvolvida.

Implementação de Prova de Conceito

(a) Vista em tablets

Order Details	Date	Time	Price
Lara	June 3, 2015	00:00 - 23:59	6.07 €
Lara	June 3, 2015	00:00 - 23:59	2.73 €
Lara	June 3, 2015	00:00 - 23:59	1.00 €
Lara	June 3, 2015	00:00 - 23:59	2.70 €
Lara	June 3, 2015	00:00 - 23:59	11.57 €
Lara	June 3, 2015	00:00 - 23:59	1.85 €
Lara			

(b) Vista em smartphones

Order Details	Date	Time	Price
Lara	June 3, 2015	00:00 - 23:59	6.07 €
Lara	June 3, 2015	00:00 - 23:59	2.73 €
Lara	June 3, 2015	00:00 - 23:59	1.00 €
Lara	June 3, 2015	00:00 - 23:59	2.70 €
Lara	June 3, 2015	00:00 - 23:59	11.57 €
Lara			

Figura 5.9: Interface do utilizador da aplicação para dispositivos móveis

Figura 5.10: Interface de conclusão de pedido de encomenda para dispositivos móveis

Implementação de Prova de Conceito

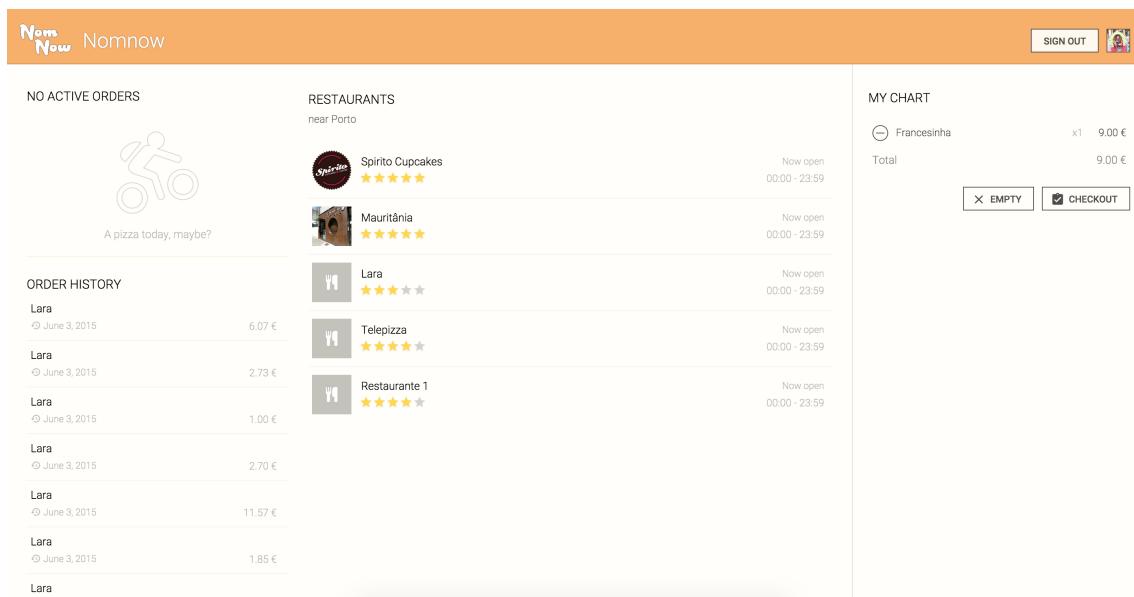


Figura 5.11: Interface do utilizador da aplicação para computadores pessoais

5.4.2 Critérios a Avaliar

Foram considerados, para avaliar a implementação desta prova de conceito, os seguintes critérios:

- Número total de linhas do código-fonte — para poder ser feita uma comparação entre a extensão dos códigos desenvolvidos para a aplicação *Web* e para a aplicação nativa de iOS já desenvolvida na Glazed Solutions, Lda. É ainda importante referir que não será contabilizado o código de bibliotecas exteriores usadas no desenvolvimento.
- Responsividade da aplicação — garantir que a aplicação se apresenta da forma esperada em ecrãs com dimensões e densidades diferentes.
- Satisfação de outros requisitos, como a navegação *offline*, navegação fluída e a sincronização de dados em tempo-real entre os serviços de *backend* Parse e Hoodie e a UI da aplicação.

5.4.3 Linhas de Código

Para contabilizar o número total de linhas do código-fonte da aplicação desenvolvida, foi usada a ferramenta CLOC³. Os resultados obtidos estão representados na Tabela 5.1.

³CLOC – Count Lines of Code. Acedido a 2015-06-13. <http://cloc.sourceforge.net/>.

Implementação de Prova de Conceito

Tabela 5.1: Número de linhas de código da aplicação *Web Nomnow*

Linguagem	Ficheiros	Linhas em Branco	Comentários	Linhas de Código
JavaScript	54	466	728	4574
HTML e CSS	55	103	83	2687
Total	109	569	811	7261

5.4.4 Testes à Responsividade

Para testar a responsividade da interface do utilizador, a aplicação desenvolvida foi testada em vários dispositivos, tendo-se comportado como esperado em todos eles. Os resultados são revelados pela Tabela 5.2.

Tabela 5.2: Resultado dos testes de responsividade

Dispositivo	Visualização Correta
Motorola Moto G 2014	Sim
Apple iPad	Sim
Apple iPhone 4	Sim
Apple iPhone 5	Sim
Apple iPhone 6	Sim
Apple iPhone 6 Plus	Sim
Google Nexus 4	Sim
Google Nexus 5	Sim
Google Nexus 7	Sim
Nokia Lumia 535	Sim
Samsung Galaxy Note II	Sim
Samsung Galaxy S4	Sim
Macbook Pro Retina 13”	Sim
Computador pessoal — resolução 1920x1080	Sim
Computador pessoal — resolução 1440x900	Sim

5.4.5 Satisfação de Outros Requisitos

Os resultados obtidos para os outros requisitos a considerar foram os seguintes:

- Navegação *offline* — Sim, embora apenas nos navegadores com suporte para *Service Workers*, ou seja, as versões mais recentes do Google Chrome e do Chrome for Android.
- Navegação fluída — Sim, embora com perdas de *performance* significativas em navegadores sem suporte nativo para *Web Components*.
- Sincronização de dados em tempo-real — Sim.

5.5 Conclusão

Neste capítulo foi descrita a aplicação implementada como prova de conceito para a arquitetura *front-end* proposta por este trabalho, bem como alguns detalhes da sua implementação que se destacaram durante a sua fase de desenvolvimento. A aplicação foi ainda testada, de acordo com os objetivos que se esperavam alcançar com a implementação da arquitetura proposta e com os requisitos definidos para a aplicação *Web* em particular. Por fim, foi feito um levantamento dos resultados obtidos. No próximo capítulo, serão discutidos estes resultados e estudadas as vantagens efetivas da implementação das ideias introduzidas com a arquitetura proposta e defendidas por esta dissertação.

Capítulo 6

Discussão de Resultados

Depois de implementada a prova de conceito da arquitetura proposta e feito o levantamento dos resultados obtidos com a aplicação desenvolvida, é agora oportuno discutir esses resultados e estudar o sucesso da arquitetura proposta no cumprimento dos objetivos que lhe foram impostos e nas respostas que foram dadas aos problemas da *Web* descritos no Capítulo 2. Esse estudo será feito neste capítulo.

6.1 Complexidade de Desenvolvimento

Como já foi descrito no Capítulo 2, o desenvolvimento de Aplicações *Web* pode ser uma tarefa complexa: a oferta limitada de elementos nativos de HTML leva frequentemente os programadores à produção de código pouco intuitivo para implementarem funcionalidades mais complexas; por ser global no contexto de uma aplicação, é difícil produzir código CSS de fácil manutenção, especialmente para projetos de maior dimensão, levados a cabo por equipas de vários elementos e, por fim, também a “Pyramid of Doom” do JavaScript, resultante do uso de sequências de funções *callback*, serve de modelo para situações em que o código produzido para Aplicações *Web* perde facilmente legibilidade e, por consequência, se torna difícil de manter.

6.1.1 *Web Components* e ECMAScript 6 e 7

Web Components foi a tecnologia integrada na arquitetura proposta para dar resposta à elevada complexidade do desenvolvimento para a *Web*, permitindo a criação de elementos HTML com regras de CSS encapsuladas — *Shadow DOM* — e com *tags* que expressem as funcionalidades que implementam. Também o uso de sequências de funções *callback*, em JavaScript e para o desenvolvimento de código assíncrono, foi provado desnecessário, com a introdução das funções *Async*, do ECMAScript 7. Para além disso, ambas as tecnologias permitem reduzir o número de linhas de código necessário para se implementarem as mesmas funcionalidades numa aplicação *Web* — *Web Components*, por contribuírem para a reutilização de código na *Web*, e as últimas

Discussão de Resultados

versões do ECMAScript, por oferecerem formas de se criar código com funcionalidades que já era possível criar antes, em JavaScript, mas agora com uma sintaxe mais compacta e intuitiva.

A seguir, será feita uma comparação entre a extensão dos códigos-fonte desenvolvidos para implementar a prova de conceito descrita no capítulo anterior e a versão da aplicação para a plataforma iOS. Deve ser tido em conta que, por ser uma aplicação *Web*, o código desenvolvido para a prova de conceito é compatível, não só também com a plataforma iOS, mas com todas as que disponibilizem um navegador que suporte as tecnologias *Web* que foram integradas na arquitetura proposta.

6.1.2 Comparação entre Aplicações *Web* e iOS

Como já foi dito no capítulo anterior, uma das razões para a escolha do desenvolvimento da aplicação Nomnow como prova de conceito da arquitetura proposta foi estar a ser desenvolvida, também na Glazed Solutions, Lda., uma versão nativa da mesma aplicação para a plataforma iOS. Desta forma, quando concluídas as duas aplicações, seria possível fazer uma comparação entre a extensão de código-fonte desenvolvido para as duas plataformas para implementar as mesmas funcionalidades. O número total de linhas do código-fonte da aplicação *Web* já foi divulgado na Secção 5.4.3 — 7261 linhas. Assim, é importante conhecer os valores da aplicação de iOS para poder ser feita a comparação.

A contabilização do número de linhas do código-fonte da aplicação de iOS foi feita com a ferramenta usada para fazer o mesmo estudo sobre a aplicação *Web* e apresenta os valores representados na Tabela 6.1.

Tabela 6.1: Número de linhas de código da aplicação de iOS

Linguagem	Ficheiros	Linhas em Branco	Comentários	Linhas de Código
Objective C	70	2735	644	3880
XIB e CSS	27	0	0	2593
C/C++ Header	70	391	491	730
Total	167	3126	1135	7203

Com um total de 7203 linhas de código desenvolvidas, em comparação com as 7261 da aplicação *Web*, as dimensões dos códigos-fonte das suas aplicações apresentam valores bastante próximos. No entanto, deve notar-se ainda que, no valor obtido para a aplicação *Web*, foi contabilizado todo o código desenvolvido para a sua implementação, incluindo os dois *plugins* Hoodie criados — para integrar o modelo de dados do Parse, usado pela aplicação de iOS, e o sistema de autenticação do Facebook —, os *Custom Elements* que tornaram possível fazer interrogações ao modelo de dados da aplicação declarativamente — “hoodie-backend-adapter”, “backend-collection” e “backend-collection-query”—, entre outros elementos reutilizáveis, que foram criados para esta aplicação em específico, mas que não dependem de um contexto e que, por isso, podem ser usados facilmente no contexto de qualquer outra aplicação baseada em *Web Components*.

Discussão de Resultados

Lembrando, mais uma vez, a frase de Taylor Savage (Secção 4.1.3) citada no Capítulo 4, é idealizado que, no futuro do desenvolvimento de Aplicações Web, cada problema da Web tenha uma resposta num *Web Component*. É, da mesma forma, esperado que o crescimento da comunidade de programadores Web que usem *Web Components* leve a que seja cada vez maior o catálogo de componentes *open source* já desenvolvidos, que resolvam grande parte dos problemas mais comuns a grande parte das Aplicações Web.

Se, por exemplo, existissem implementados alguns dos componentes que foram usados no desenvolvimento da aplicação Web criada, a extensão do código necessário para a sua implementação poderia ser significativamente menor. Os valores da Tabela 6.2 foram obtidos assumindo que os dois *plugins* Hoodie criados e os elementos “hoodie-backend-adapter”, “hoodie-parse-plugin”, “backend-collection” e “backend-collection-query” já existiam, quando a aplicação de prova de conceito foi desenvolvida, e aplicam-se a uma nova aplicação que possa ser desenvolvida no futuro com estas tecnologias.

Tabela 6.2: Número de linhas de código da aplicação Web

Linguagem	Ficheiros	Linhas em Branco	Comentários	Linhas de Código
JavaScript	45	396	598	3337
HTML e CSS	51	95	74	2655
Total	96	491	672	5992

Em suma, espera-se que o crescimento dos repositórios de *Web Components open source* na Web leve a que, no futuro, seja possível desenvolverem-se Aplicações Web cada vez mais complexas com menos código e em menos tempo.

6.2 Responsividade

Como é visível na Tabela 5.2, do capítulo anterior, a interface do utilizador responsiva, criada para a aplicação desenvolvida, comportou-se da forma esperada em todos os dispositivos em que foi testada. Deve-se destacar que, na lista de dispositivos usados no teste, foram incluídos dispositivos com diferentes métodos de entrada — *touch* para os *smartphones* e *tablets* usados e rato e teclado para os computadores pessoais — e que a aplicação adapta-se para permitir a navegação por gestos *touch* em *smartphones*.

É, contudo, importante referir que ainda nem todos os navegadores suportam o uso de *Responsive images*. Existe já, porém, um *polyfill* que pode ser usado para emular o seu suporte nesses navegadores — Picturefill¹.

¹Picturefill. Acedido a 2015-06-13. <http://scottjehl.github.io/picturefill/>.

6.3 Funcionamento *Offline*

À exceção das funcionalidades da aplicação que faziam sentido só poderem ser usadas com uma ligação à Internet, como é o caso da autenticação com credenciais de Facebook e da criação de novas encomendas, todas as outras funcionalidades podem ser usadas *offline*. Para além disso, os dados usados pela aplicação são armazenados localmente e mantidos atualizados, à responsabilidade do Hoodie, com os dados armazenados remotamente, na base de dados CouchDB alojada no servidor. Por esta razão e por não depender da comunicação com o *backend* de Hoodie, o acesso a dados na aplicação é rápido.

Só com o Hoodie, não era possível, no entanto, armazenar em *front-end* as dependências da aplicação. Assim, sempre que a janela do navegador em que a aplicação estivesse a ser corrida fosse atualizada sem haver uma ligação à Internet, o navegador iria tentar transferir novamente todas as dependências da aplicação, falhar, por não haver uma ligação à rede, e a aplicação não iria ser corrida. A integração do *Service Worker* na arquitetura da aplicação tornou possível armazenar essas dependências em *cache* e correr e usufruir da aplicação mesmo nessas situações.

Em suma, o Hoodie e o *Service Worker* foram os elementos-chave que permitiram conceder à aplicação desenvolvida uma experiência de utilização *offline* semelhante à que ainda só é visível em aplicações nativas. Deve ser acrescentado, no entanto, que o *Service Worker* é uma tecnologia recente, suportada ainda por um número bastante limitado de navegadores — Google Chrome, Opera e Chrome for Android.

6.4 Limitações das Tecnologias Usadas

Grande parte das tecnologias integradas na arquitetura proposta por esta dissertação são ainda relativamente recentes no mundo da *Web* e, por esta razão, apresentam bastantes limitações. Nesta secção são descritas algumas das que mais se evidenciaram.

6.4.1 *Web Components*

Uma das maiores limitações do desenvolvimento de Aplicações *Web* com *Web Components* é estes não serem ainda suportados nativamente por grande parte dos navegadores atuais. Usar *polyfills* para emular o seu suporte nesses navegadores é uma solução temporária para o problema. Porém, as aplicações desenvolvidas com *Web Components* têm uma *performance* significativamente superior quando corridas em navegadores com suporte nativo, em comparação com quando são corridas com suporte emulado por *polyfills*. Espera-se, no entanto, que, no futuro, o suporte para estas tecnologias seja implementado em grande parte dos navegadores mais recentes da *Web* e que passe a ser desnecessário o uso de *polyfills* no desenvolvimento de aplicações deste tipo.

6.4.2 Hoodie

O Hoodie foi a *framework* escolhida para trazer para a arquitetura proposta as ideias defendidas pelo conceito *Offline-first*, por ser a única atualmente disponível que permite armazenar dados em *front-end*, que abstrai o programador da sincronização desses dados com uma base de dados remota e que é ainda NoBackend e dispensa o programador da necessidade da implementação de código de servidor. No entanto, é ainda um projeto em desenvolvimento e com bastantes limitações.

Por vezes, por exemplo, é difícil encontrar documentação que corresponda ao seu estado atual de desenvolvimento. É também frequente encontrar-se documentação incompleta para funcionalidades que já foram implementadas na *framework*, ou funcionalidades já implementadas não terem o comportamento esperado, em algumas situações.

O fato de o Hoodie recorrer ao HTML5 *Web Storage* para armazenar dados em *front-end* é outra limitação do Hoodie que deve ser considerada. O HTML5 *Web Storage* foi uma tecnologia criada para armazenar, em *front-end*, pequenas quantidades de dados e de uma forma simples. Outras funcionalidades normalmente oferecidas por uma base de dados tradicional, como interrogações ou a criação de várias coleções e de definições de acesso personalizadas para determinados dados, exigem uma solução mais complexa. A equipa de desenvolvimento do Hoodie anunciou, no entanto, que está, já há algum tempo, a trabalhar numa alternativa melhor.

Para garantir a sincronização dos dados entre *front-end* e *backend*, o Hoodie recorre ainda à técnica de *long-polling*. Como consequência disto, uma aplicação desenvolvida com Hoodie estará constantemente a fazer pedidos de rede ao *backend* de Hoodie, de forma a que, quando os dados forem alterados no *backend*, as alterações sejam imediatamente enviadas numa resposta ao último pedido de *long-polling* feito pela aplicação [Sha13]. Todos os pedidos de rede feitos que, por não terem ocorrido alterações de dados em *backend*, não tiverem resposta consumirão, desnecessariamente, recursos de rede, da mesma forma que consumirão recursos de processamento nos dispositivos que corram a aplicação. Principalmente em dispositivos móveis, onde os baixos consumos de bateria e de recursos de rede são importantes, o uso de *long-polling* constitui uma limitação. Os HTML5 *Web Sockets* são uma tecnologia que permite resolver estes problemas [LG13], mas que ainda não está integrada na *framework* Hoodie.

6.5 Conclusões

Em suma, a análise dos resultados obtidos feita neste capítulo confirma que as tecnologias integradas na arquitetura proposta podem ser a solução para os problemas descritos no Capítulo 2. A aplicação desenvolvida mostrou-se responsiva, oferece uma experiência de navegação *offline* semelhante à de aplicações nativas e os *Web Components* e as novas versões da especificação do ECMAScript prometem trazer melhorias importantes para o processo de desenvolvimento de Aplicações *Web*. À luz destes resultados, é concluído que os objetivos impostos para a arquitetura proposta por esta dissertação forem cumpridos.

Discussão de Resultados

Capítulo 7

Conclusões e Trabalho Futuro

Os conceitos *Web Components*, *Offline-first*, *Responsive Web Design* e *Single-Page Applications* prometem vir revolucionar o futuro do desenvolvimento de aplicações para a *Web*. Este trabalho pretende contribuir nesse sentido e antecipar algumas das mudanças que poderão ser introduzidas com essa revolução.

7.1 Resumo

Desenvolver Aplicações *Web* de qualidade, que tirem proveito das particularidades de todos os dispositivos do ecossistema da *Web* e que não percam funcionalidade em função das limitações desses dispositivos, pode ser, atualmente e como foi visto nesta dissertação, um desafio grande. O estudo do estado da arte das tecnologias feito neste trabalho permitiu concluir que já existem várias tecnologias disponíveis que permitem simplificar esse desafio.

A arquitetura *front-end* proposta nesta dissertação mostrou que os conceitos *Web Components*, *Offline-first*, *Responsive Web Design* e *Single-Page Applications* podem ser aliados para simplificar o processo de desenvolvimento de Aplicações *Web*, brindando-nos ainda com produtos finais que se adaptam melhor às necessidades da *Web* atual e dos seus utilizadores. O sucesso no desenvolvimento da prova de conceito para a arquitetura proposta foi prova disso: a separação do código-fonte intuitivo e de fácil manutenção por componentes e a reutilização desses componentes na aplicação mostrou que as tecnologias que integram a arquitetura proposta oferecem vantagens significativas na simplificação do processo de desenvolvimento de Aplicações *Web*. Além disso, a aplicação criada mostrou-se responsiva nos testes e oferece uma experiência de utilização *offline* semelhante à de uma aplicação nativa.

Este pode ser, portanto, o caminho para se desenvolverem aplicações *Web* desenvolvidas que sejam capazes de responder às necessidades atuais da *Web*, do seu ecossistema de dispositivos e dos utilizadores.

7.2 Contribuição Científica

As escolhas das tecnologias a integrar na arquitetura *front-end* proposta basearam-se no estudo teórico e prático que foi feito e foram justificadas quando foi descrita a arquitetura. Este trabalho é ímpar, por ser o primeiro a estudar e propor uma forma de desenvolver Aplicações *Web* baseada nos conceitos *Web Components*, *Offline-first*, *Responsive Web Design* e *Single-Page Applications* e introduz várias ideias que terão interesse para quem, no futuro, pretenda desenvolver Aplicações *Web* que implementem esses conceitos.

Para além disso, os componentes reutilizáveis implementados para a prova de conceito da arquitetura proposta são *open source* e foram publicados com a licença MIT, podendo ser reutilizados livremente por programadores que desenvolvam aplicações com as tecnologias que foram integradas na arquitetura proposta.

7.3 Satisfação dos Objetivos

Considera-se que os principais objetivos definidos para esta dissertação foram concretizados. Foi possível selecionar, dos quatro conceitos explorados a integrar na arquitetura, um grupo de tecnologias que permitiu explorar as ideias e as vantagens mais importantes de cada um desses conceitos e que, por serem compatíveis entre si, puderam ser usadas para presentear a arquitetura proposta com os seus benefícios. Confirmou-se, assim, que o desenvolvimento de Aplicações *Web* pode, efetivamente, ser simplificado, recorrendo às tecnologias que integraram a arquitetura proposta, e que as aplicações criadas com essas tecnologias, não deixando de ser Aplicações *Web*, são capazes de oferecer experiências de utilização com características normalmente presentes apenas em aplicações nativas.

7.4 Trabalho Futuro

A implementação de testes unitários em *front-end* foi referida na definição da arquitetura *front-end* proposta, mas acabou por não ser aplicada à prova de conceito desenvolvida. Da mesma forma, também não foram implementados os testes unitários para os *plugins* de Hoodie criados. Implementar esses testes, recorrendo às ferramentas Mocha, Chai, Sinon e Web Component Tester, integradas na arquitetura proposta, seria assim uma prioridade para dar seguimento a este trabalho.

Para além disso, como já foi referido, embora, no momento em que este trabalho começou a ser desenvolvido, a versão do Polymer disponível fosse a 0.5, já foi lançada uma nova versão da biblioteca — Polymer 1.0 — e seria importante migrar os componentes Polymer desenvolvidos para a nova versão.

O Hoodie é, como já foi dito, uma *framework* ainda em fase de desenvolvimento e, por essa razão, têm ainda bastantes limitações. No entanto, espera-se que essas limitações sejam ultrapassadas num futuro próximo, quando a base de dados *front-end* PouchDB for integrada na *framework* e a comunicação entre o cliente e o servidor passar a ser feita recorrendo a HTML5 Web Sockets.

Conclusões e Trabalho Futuro

Assim, dar seguimento a este trabalho passaria ainda por integrar uma nova versão do Hoodie na arquitetura proposta.

A comunicação entre os vários *Custom Elements* na arquitetura proposta é feita através de eventos de DOM, que se propagam sempre no sentido dos nós da árvore de DOM mais afastados para os mais próximos da sua raiz. Este tipo de comunicação permite que os elementos que recebem esses eventos os tratem sem precisarem de conhecer o funcionamento interno dos componentes que os lançaram. No entanto, na arquitetura proposta, quando um componente-pai espoleta uma ação componente-filho, são chamados diretamente os métodos da API de DOM do componente-filho. Desta forma, o componente-pai precisa de conhecer os métodos da API de DOM do componente-filho e, caso esta seja alterada no futuro, é provável que a comunicação deixe de funcionar. O ideal seria usar também eventos para fazer a comunicação entre componentes no sentido dos componentes mais próximos da raiz da árvore de DOM para os mais afastados. Existe já uma tecnologia que permite criar canais e trocar eventos bidirecionais através desses canais entre componentes — *Communicating Sequential Processes* [Osm14a]. Embora o tempo disponível para a realização deste trabalho não tenha sido suficiente para explorar o interesse e a possibilidade de se integrar esta tecnologia na arquitetura proposta, é uma questão com interesse para dar continuidade a este trabalho.

Conclusões e Trabalho Futuro

Referências

- [Arc14] Archibald, Jake. ES7 async functions. *JakeArchibald.com*, 2014. URL: <http://jakearchibald.com/2014/es7-async-functions/> [último acesso em 2015-06-13].
- [Bid13a] Eric Bidelman. Custom Elements: defining new elements in HTML. *HTML5 Rocks*, 2013. URL: <http://www.html5rocks.com/en/tutorials/webcomponents/customelements/> [último acesso em 2014-11-18].
- [Bid13b] Eric Bidelman. HTML Imports: #include for the web. *HTML5 Rocks*, 2013. URL: <http://www.html5rocks.com/en/tutorials/webcomponents/imports/> [último acesso em 2014-11-18].
- [Bid13c] Eric Bidelman. HTML's New Template Tag: standardizing client-side templating. *HTML5 Rocks*, 2013. URL: <http://www.html5rocks.com/en/tutorials/webcomponents/template/> [último acesso em 2014-11-18].
- [Coo13] Dominic Cooney. Shadow DOM 101. *HTML5 Rocks*, 2013. URL: <http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom/> [último acesso em 2014-11-18].
- [DH07] D Diaz e R Harmes. *Pro JavaScript Design Patterns*. The Expert's voice in Web development. Apress, 2007. URL: <https://books.google.pt/books?id=za3vlnlWxb0C>.
- [Fan11] Yu Fan. Cascading style sheets. Master's thesis, Kemi-Tornio University of Applied Sciences, 2011.
- [Fey13] Alex Feyerke. Designing Offline-First Web Apps. *A List Apart*, 2013. URL: <http://alistapart.com/article/offline-first> [último acesso em 2014-11-30].
- [Fra13] B. Frain. *Sass and Compass for Designers*. Packt Publishing, 2013. URL: <http://www.google.hr/books?id=shJEwk7y80oC>.
- [Heh14] Nick Hehr. Getting Started with noBackend. *noBackend Blog*, 2014. URL: <http://nobackend.org/2014/05/getting-started-with-noBackend.html> [último acesso em 2014-11-28].
- [Hem13] Zef Hemel. NoBackend: Front-End First Web Development. *InfoQ*, 2013. URL: <http://www.infoq.com/news/2013/05/nobackend> [último acesso em 2014-11-28].

REFERÊNCIAS

- [Hoo14a] How Hoodie Works. Hoodie Docs, 2014. URL: <http://docs.hoodie.ie/en/hoodieverse/how-hoodie-works.html> [último acesso em 2015-06-11].
- [Hoo14b] Intro. hoodie, 2014. URL: <http://hood.ie/intro/> [último acesso em 2014-11-28].
- [Hoo14c] Plugins. Hoodie Docs, 2014. URL: <http://docs.hoodie.ie/en/plugins/tutorial.html> [último acesso em 2015-06-13].
- [Jak12] Jake Archibald. Application Cache is a Douchebag. *A List Apart*, 2012. URL: <http://alistapart.com/article/application-cache-is-a-douchebag> [último acesso em 2015-06-10].
- [Kei05] Jeremy Keith. DOM Scripting. *Friends of ED*, 2005.
- [Kit14] Eiji Kitamura. Introduction to the template elements. *Webcomponents.org*, 2014. URL: <http://webcomponents.org/articles/introduction-to-template-element/> [último acesso em 2014-11-18].
- [LeP14] Pete LePage. Images in Markup. Web Fundamentals. Google Developers, 2014. URL: <https://developers.google.com/web/fundamentals/media/images/images-in-markup?hl=en#use-relative-sizes-for-images> [último acesso em 2015-06-26].
- [LG13] Peter Lubbers e Frank. WebSocket.org Greco. The Benefits of WebSocket. *WebSocket.org*, 2013. URL: <https://www.websocket.org/quantum.html> [último acesso em 2015-06-25].
- [Mar10] Ethan Marcotte. Responsive Web Design. *A List Apart*, 2010. URL: <http://alistapart.com/article/responsive-web-design> [último acesso em 2014-12-08].
- [Mar11] Ethan Marcotte. *Responsive Web Design*. Editions Eyrolles, 2011.
- [Mat14] Introduction - Material design. Google design guidelines, 2014. URL: <http://www.google.com/design/spec/material-design/introduction.html> [último acesso em 2014-12-08].
- [Max14] Max Vujovic. CSS animations and transitions performance: looking inside the browser. *Web Platform Team Blog*, 2014. URL: <http://blogs.adobe.com/webplatform/2014/03/18/css-animations-and-transitions-performance/> [último acesso em 2015-01-05].
- [MDN13] Using CSS flexible boxes. Web developer guide. MDN, 2013. URL: https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Flexible_boxes [último acesso em 2015-06-23].
- [Med11] Miller Medeiros. The problem with CSS pre-processors, 2011. URL: <http://blog.millermedeiros.com/the-problem-with-css-pre-processors/> [último acesso em 2015-06-11].
- [Met13a] Professional services. Meteor, 2013. URL: <https://www.meteor.com/professional-services> [último acesso em 2015-06-11].

REFERÊNCIAS

- [Met13b] The Meteor Manual. Meteor, 2013. URL: <http://manual.meteor.com/> [último acesso em 2014-11-28].
- [Mit13] Tilo Mitra. Understanding Design Patterns in JavaScript, 2013. URL: <http://code.tutsplus.com/tutorials/understanding-design-patterns-in-javascript--net-25930> [último acesso em 2015-06-10].
- [Osm12] Addy Osmani. *Learning JavaScript Design Patterns*. O'Reilly Media, Inc., 2012.
- [Osm13] Addy Osmani. Concatenating Web Components with Vulcanize. Polymer, 2013. URL: [https://www.polymer-project.org/0.5/articles\(concatenating-web-components.html](https://www.polymer-project.org/0.5/articles(concatenating-web-components.html) [último acesso em 2015-06-14].
- [Osm14a] Addy Osmani. JavaScript Application Architecture On The Road To 2015 — Google Developers — Medium, 2014. URL: <https://medium.com/google-developers/javascript-application-architecture-on-the-road-to-2015-d8125811101b> [último acesso em 2015-04-5].
- [Osm14b] Addy Osmani. Unit Testing Polymer Elements. Polymer, 2014. URL: <https://www.polymer-project.org/0.5/articles/unit-testing-elements.html> [último acesso em 2015-06-10].
- [OZPSG10] Francisco Ortin, Daniel Zapico, J Baltasar García Pérez-Schofield e Miguel Garcia. Including both static and dynamic typing in the same programming language. *IET software*, 2010.
- [Par12a] Customers. Parse, 2012. URL: <https://parse.com/customers> [último acesso em 2015-06-11].
- [Par12b] Docs Overview. Parse, 2012. URL: <https://www.parse.com/docs> [último acesso em 2014-11-28].
- [Pol14a] Layout attributes. Polymer, 2014. URL: <https://www.polymer-project.org/0.5/docs/polymer/layout-attrs.html> [último acesso em 2015-06-23].
- [Pol14b] Understanding Polymer. Polymer, 2014. URL: <https://www.polymer-project.org/docs/start/everything.html> [último acesso em 2015-02-13].
- [Rey14] Scott Reynen. Declarative Programming And The Web. *Smashing Magazine*, 2014. URL: <http://www.smashingmagazine.com/2014/07/30/declarative-programming/> [último acesso em 2014-11-17].
- [RO15] Zeno Rocha e Addy Osmani. Why Web Components? *Webcomponents.org*, 2015. URL: <http://webcomponents.org/articles/why-web-components/> [último acesso em 2015-06-26].
- [RSA15] Alex Russell, Jungkee Song e Jake Archibald. Service Workers. *World Wide Web Consortium*, 2015. URL: <http://www.w3.org/TR/service-workers/> [último acesso em 2015-02-12].

REFERÊNCIAS

- [Sam13] Ana Isabel Sampaio. Responsive web design. Master's thesis, Universidade do Minho, 2013. URL: http://repository.sdm.uminho.pt/bitstream/1822/27902/1/eeum_di_dissertacao_pg20190.pdf.
- [Sav15] Taylor Savage. Polymer 1.0 Released! *Google Developers Blog*, 2015. URL: <http://googledevelopers.blogspot.pt/2015/05/polymer-10-released.html> [último acesso em 2015-06-11].
- [Sha13] Nikhar Sharma. Push Technology–Long Polling. *International Journal of Computer Science and Management Research*, 2013. URL: <http://www.ijcsmr.org/vol2issue5/paper398.pdf>.
- [Sin10] Documentation. SinonJS, 2010. URL: <http://sinonjs.org/docs/> [último acesso em 2015-02-13].
- [Sit14] Vendor-specific Properties. SitePoint, 2014. URL: <http://www.sitepoint.com/web-foundations/vendor-specific-properties/> [último acesso em 2015-02-15].
- [Wal15] Philip Walton. Side Effects in CSS. *PhilipWalton.com*, 2015. URL: <http://philipwalton.com/articles/side-effects-in-css/> [último acesso em 2015-06-10].
- [Wei14] Manuel Weiss. Using Mocha JS, Chai JS and Sinon JS to Test your Frontend JavaScript Code. *Codeship*, 2014. URL: <http://blog.codeship.com/mocha-js-chai-sinon-frontend-javascript-code-testing-tutorial/> [último acesso em 2015-02-13].
- [Wei15] Assaf Weinberg. Offline Data In The Browser. 2015. URL: <http://www.levvel.io/blog/offline-data-in-the-browser> [último acesso em 2015-05-28].
- [Yat12] Ian Yates. A Simple, Responsive, Mobile First Navigation. *Tuts+*, 2012. URL: <http://webdesign.tutsplus.com/articles/a-simple-responsive-mobile-first-navigation--webdesign-6074> [último acesso em 2015-06-26].
- [Zak12] Nicholas C Zakas. *Maintainable JavaScript*. O'Reilly Media, Inc., 2012.
- [Zak14] Nicholas C. Zakas. *Understanding ECMAScript 6*. Leanpub, 2014. URL: <https://leanpub.com/understandinges6/read/>.
- [Zim13] Joe Zimmerman. What's The Point Of Promises? 2013. URL: <http://www.telerik.com/blogs/what-is-the-point-of-promises> [último acesso em 2015-06-26].